International Journal of Computing and Digital Systems
ISSN (2210-142X)
Int. J. Com. Dig. Sys. , No. (Mon-20..)

http://dx.doi.org/10.12785/ijcds/XXXXXX

Graph Processing Systems: A Comprehensive Review with Emphasis on Graph Mutations

Soukaina Firmli[1] **and Dalila Chiadmi**[1]

[1]*Mohammed V University in Rabat , Ecole Mohammadia d'Ingénieurs , SIP Research Team , Morocco,*

Received Mon. 20, Revised Mon. 20, Accepted Mon. 20, Published Mon. 20

Abstract: The growing need for managing extensive dynamic datasets has propelled graph processing and streaming to the forefront of the data processing community. Coping with the sheer volume, intricacy, and continual evolution of data poses a challenge to graph data processing systems, necessitating the development of innovative techniques for effective handling and analysis. This paper underscores two pivotal aspects within Graph Processing Systems (GPS) that significantly impact overall system performance: 1) the graph representation, encompassing the data structures storing vertices and edges, and 2) graph mutation protocols, outlining approaches for assimilating and storing new graph updates, such as additions of edges and vertices. Given the irregularity of graph workloads and the large scale of real-world graphs, face numerous challenges, requiring adept solutions to ensure both efficient storage and update protocols. This dual imperative enables rapid analytics and streaming capabilities. Our paper aims to furnish a comprehensive overview of diverse methodologies employed by researchers to surmount performance challenges inherent in the design GPS and streaming. Our review highlights challenges linked to graph representation and update protocols, offering insight into researchers' efforts to tackle these issues. Conclusively, we identify research gaps in the domains of graph representation and mutation, emphasizing areas that remain unexplored and unresolved.

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

1. INTRODUCTION

Big data refers to extensive and complex datasets collected and analyzed by organizations to gain insights and make informed decisions. This data originates from diverse sources, including social media, e-commerce transactions, and sensor data. Graphs, in particular, stand out as a prominent data model in the current era of big data and data deluge [1], with the analysis of large-scale graphs becoming increasingly significant. Examples include examining the architecture of the Internet [2], exploring social or neurological networks [3], and recording the activity of proteins [4]. Efficiently processing these big data graphs poses challenges. Firstly, these networks are highly complex, featuring thousands of nodes and millions or even billions of edges [5]. Secondly, the relevant graphs inherently change over time as new social relationships form, novel linkages are established, or protein interactions evolve. To address these challenges, graph processing and streaming systems are developed for processing and analyzing massive dynamic graphs and networks [6]. These systems aim to scale horizontally by dispersing graph data and computation among a cluster of devices or to scale vertically by utilizing parallelism in multi-core machines. However, the software design and implementation of these systems need optimization for different workloads, as design choices may create challenges by hindering the computational capabilities of servers. Two vital software design elements of these systems, crucial for overall performance, are: 1) the graph data structure, storing vertices and edges, and 2) graph mutation protocols, the approaches for ingesting and storing new graph updates, such as new edges and vertices. Ideally, a graph processing system should offer excellent analytics performance, fast mutations (i.e., vertex or edge insertions and deletions), and low memory consumption with or without mutations. However, the large scale of graph data and complex algorithms pose computational and management challenges, requiring high-performance computing models as referenced by Tulasi et al. [7]. Furthermore, the rapid and continuous generation of new streams in real-world graphs may delay analytics and cause a substantial increase in memory consumption in graph streaming systems. These challenges have prompted researchers to explore ways to enhance classic graph data structures and update protocols to enable low-latency graph analytics and high update throughput. Techniques include in-place update techniques [8], [9], batching techniques [10], [11], and changeset-based updates with delta maps [12] and multi-versioning [13] to improve the update-friendliness of Compressed Sparse Row (CSR). In this paper, we present a review focusing on the performance challenges encountered by researchers when

E-mail address: soukaina.firmli@gmail.com, chiadmi@emi.ac.ma

designing graph processing and streaming solutions for large-scale dynamic graphs. We examine how researchers attempt to represent graphs in memory and design update protocols for efficient graph analytics, queries, and streaming, closely evaluating the performance implications of different techniques for storing and updating graphs in memory. Additionally, we identify gaps in research on graph representation and mutation that have yet to be addressed.

In this paper, we present a review that focuses on the performance challenges that researchers face when designing graph processing and streaming solutions for large scale dynamic graphs. Essentially, we review how they attempt to represent graphs in memory and design update protocols for efficient graph analytics, queries and streaming, by taking a close look at the performance implications of different techniques for storing and updating graphs in memory. Moreover, we identify gaps in research of graph representation and mutation that haven't been addressed.

The paper is organized as the following. First 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 conclude in 8.

2. BACKGROUND

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 [14], that come from storing and processing large graph data, pose unique challenges in the field of computer science and data processing.

First, 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 [15]. Within a single graph, vertex degrees can vary widely due to the power-law distribution [16], 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, it's possible that additional memory and processing power (RAM, CPU) will be needed to process vertices with higher degrees [17].

In addition to the skew of real world graph data sets that graph processing systems need to ingest and store, the large size of graph data and access patterns of graph algorithms are important factors that highly impact the performance of these systems in the context of graph processing and streaming [13]. 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 [13] 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 that occur when the CPU tries to access data that is not already stored in the cache [18], causing more slowdown.

Second, 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 [15]. For example, in a social network, graph streaming can be used to detect new connections between people in real-time. Mutations in graph streaming refer to the process to add, modify, or delete vertices and edges, allowing to maintain the graph as it changes over time.

Finally, to address these challenges of storing, analyzing and streaming big graph data, graph technology, including graph processing systems and graph databases like Neo4j [19], emerges as a suitable approach 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 [20]. 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 the hardware resources such as parallelism and Distributed machines to address the scalability challenge of processing and streaming large graphs efficiently. Moreover, they use techniques to optimize classical data structures and graph updates by improving their cache performance, using techniques such as compression[21], partitioning[22], and memory allocators[23].

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. In this context, we discuss and analyze in this paper the growing interest in research to achieve high performance by finding new ways to incorporate these techniques in the graph processing systems and streaming.

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 on the different techniques used by researchers to improve the performance of graph processing systems and streaming. In an attempt to give this overview, we narrow 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 [19]). 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 a result of this methodology, we break down these two elements and specify the research questions of our interest as the following:

1) Q1. What are the most used graph data structures used to represent graphs and how do researchers employ them to support large graphs and frequent updates while providing high performance and low memory consumption?
2) Q2. What are the methods for improving the cache performance of graph structures and algorithms?
3) Q3. How do researchers minimize the memory footprint of the massive dynamic graphs to fit in memory?
4) Q4. What are the main protocols for supporting graph updates and how do they impact the performance of graph analytics and memory?
5) Q5. What are the techniques used by researchers to improve the performance of graph mutations?

We extracted data relevant to the research questions and we performed a synthesis as shown in Table I.

4. TECHNIQUES FOR EFFICIENT GRAPH REPRESENTATION

In this section, we give an overview of how to store graph data elements efficiently by reviewing and analyzing different techniques in the literature. These techniques allow to tackle the challenges related to optimizing classical graph data structures for high analytic and update performance as well as minimising 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 (Sec. 4-B), compression (Sec. 4-C), using memory allocator software (Sec.4-D) and partitioning (Sec. 4-E). We discuss how they fine-tune the systems for graph processing and streaming using these classical techniques to achieve high performance. Finally, we present a summary in Table III.

A. Representative Graph Containers

The following is a list of some widely used graph representations. We provide descriptions of the 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 signify 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 neighbors. 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 [8] or utilize linked lists with fixed-size edge-containing buckets[24].

This schema stores vertices in a node list where each element points to a list of neighbors. This is less memory consuming than adjacency matrix since only the existing edges are stored. The common format uses linked lists for adjacencies, but there exist other more cache-friendly variants such as Blocked Adjacency Lists, where adjacencies are represented by simple arrays [8] or with linked lists of buckets containing a fixed size of edges [24].

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 neighbors' list. To find the degree of node i, we compute Node Array[i+1] minus 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++ [17][25][26]—aiming to enhance support for quicker structural updates. Further details on this topic are discussed in subsequent sections.

B. Memory layout

A good memory layout for a graph representation refers to the optimal way to arrange the graph data in memory to enhance its performance [18]. This means that the data structure layout in memory should allow to take advantage

TABLE I. 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, RQ5	Sec. 4, 5
Graph Mutation	The implementation of graph mutations: in-place, delta maps, snaphsots	RQ4	Sec. 5
Performance Optimization	The process of modifying a system to improve its functionality, thus making it more efficient in read and update workloads	RQ2, RQ3, RQ5	Sec. 4, 5
Parallelism & Hardware Resources	The underlying architecture of systems such as multi-core, CPU Cache and Distributed Systems	RQ2, RQ3, RQ5	Sec. 4, 5

TABLE II. Space and time complexityy for different topological operations on classic graph data structures. We annotate N: Number of vertices in a graph, M: Number of edges, Deg(V): Degree of a vertex V, and C: The cost of cache misses due to the access to non-contiguous memory.

Operation	Adjacency Matrix	Adjacency List	CSR	Edge List
Iteration over neighbors	O(n)	O(deg (v)) + C	O(deg(v))	O(m)
Vertex Insertion	$O(n^2)$	O(1)	O(n)	-
Edge Insertion	O(1)	O(1)	O(m)	O(1)
Edge Deletion	O(1)	O(deg (v))	O(m)	O(m)
Systems	[27]	[24][28][29][8][17]	[25][17][26][30][31][32]	[29][26]

of 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 [13], [25], [17], [30], hence better performance.

By taking these factors into consideration when selecting and implementing data structures, researchers in the graph community propose new variants of data structures that allow for better cache performance, and that by changing memory layouts of classical data structures [33]. Essentially, to remediate to the poor cache locality, when storing edges, many researchers [24], [34], [35] use *bucketing* technique where buckets are used to group edges from the same source vertex together, or using linked lists to group edges from different source vertices together. These buckets shouldn't be too huge, as it would slow down update performance, or too small, as that would cause cache misses. Therefore, making the memory layout of the AL, more cache-friendly while still maintaining good update performance at no memory cost.

Furthermore, since CSR is known for its high cache performance and slow update performance, many researchers [36] [27][13] [9] 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 *space-optimized* (SO) and *performance-optimized* (PO) structures. The *performance optimized* keeps a complete list of edges of the same vertex in each version of the graph. This variant of CSR is very close to the actual immutable CSR for static graphs and gives almost the same performance even when the graph is being updated multiple times and even for executing algorithms that require many random accesses to memory. However, since the edge list is copied for every updated vertex in the new version, the memory consumption grows steeply which is not ideal for real-world commodity hardware [17].

On the other hand, the *space-optimized* variant stores fragments of the edge lists for every vertex, meaning it stores only the new edges in the version. The performance of this variant is slower in principle since the edge lists are not stored contiguously, and the system needs to reconstruct the full adjacency of a vertex, which incurs many cache misses. However, in some cases such as PageRank where vertices are accesses in order, the space-optimized CSR design performs better.

C. Compression

Compression allows to reduce the amount of memory needed to store data while maintaining its essential properties and functionality[21]. Compression can be useful in situations where storage space is limited or expensive, but 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.

Recently, there have been quite an interest in using compression for graph representations [34], [37]. For instance, Aspen [34] 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 [37], 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. Unfortunately, to run graph algorithms in SSTGraph on a given data set, one should make sure the graph is symmetric, or pre-process the data set to make the data set reflect a symmetric graph, which in practice is cumbersome. Moreover, the interface is tightly dependent on the Ligra APIs [31] and lacks of some basic graph primitives such as methods for getting the out degree of vertices in a directed graph.

D. Memory Allocators

A memory allocator is a software component that manages the allocation and deallocation of memory in a computer program [23]. The primary responsibility of allocators is to handle requests from the program for memory, and to keep track of which parts of memory are currently being used and which parts are free and available for allocation. Many GPS use memory allocators to achieve a good performance in terms of read, updates and memory consumption. For instance, memory allocators such as Jemalloc [38] and TCMalloc [39], 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 [40], which leads to better cache locality and lower memory footprint.

Another approach used by some systems [24], [35], [17] is developing built-in memory managers that facilitate the speedy allocation of memory needed for applying mutations. For instance, in order to efficiently perform memory reclamation [41] and manage space, Hornet's [35] internal memory management uses a B+ tree for insertions and deletions to keep track of the available blocks of edges. When data is duplicated, the system uses a load-balancing mechanism to locate the freed memory for later usage. This restriction applies to both Hornet and STINGER, whose great performance is due to their memory allocation algorithms.

Another approach used by systems takes advantage of the persistent storage of machines to allocate large graphs. For instance, Metall [42], a permanent memory allocator that uses the copy-on-write technique for graph workloads, stores and manipulate enormously huge graphs at the exascale (billions of billions of operations per second), by employing smart allocation algorithms like those found in SuperMalloc [43]. 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, in order 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. Even while *reflink* helps save memory when making new snapshots, not all filesystems have built-in support for it. If it isn't available, Metall will instead resort to making full copies, which dramatically increases the memory footprint.

E. Partitioning

Graph data partitioning is the process of dividing a large graph into smaller subgraphs [22], called partitions, in order to enable parallel processing of the graph on multiple machines or processors. Graph partitioning is important for large-scale graph processing, as it allows us to distribute the computation and storage of the graph across multiple machines or processors, and to perform computations on each partition independently and in parallel.

Graph data partitioning is typically performed by dividing the vertices of the graph into partitions, such that the edges of the graph are distributed evenly among the partitions, while minimizing the number of edges that cross between partitions[32]. This is done in order to minimize the amount of communication required between the machines or processors during computation, as communication can be a bottleneck for large-scale graph processing. There are several algorithms and techniques for graph data partitioning, such as recursive spectral bisection, multilevel k-way partitioning, and label propagation [44]. The choice of partitioning algorithm depends on the specific characteristics of the graph and the system architecture being used.

Some GPS systems [45], [46] use partitioning for optimizing 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 very small number of articles address the challenges that come with it.

5. UPDATE PROTOCOLS FOR EFFICIENT GRAPH STREAMING

In addition to the representation of graphs in memory for graph queries and analytics, a wide range of GPS [24], [13], [50], [34] support graph mutation by allowing modifications to the graph's topology by adding or removing edges and vertices. Essentially, to achieve that, GPS implement different update protocols, which are different approaches for the ingestion and the storage of new incoming graph data. Specifically, update ingestions refer to the processing of the stream of update operations, either in bulk or concurrently with analytics before storing the updates in the final graph. On the other hand, update storage refers to the protocol of storing the updates either by directly appending them to the graph storage (a.k.a., in-place updates), or using additional data structures to store the new updates (a.k.a., delta stores). To put this in the context of our study, the main challenge for GPS is to design these graph update protocols, to efficiently process large volumes of data updates while yet providing optimal analytical performance and low memory footprint. As a result, in attempt to answer our research question, we review the different techniques in the literature for implementing graph updates, and discuss their performance implications. This ultimately allows us to find out more about this performance trade-off challenges for GPS and the research gaps.Finally, we present a summary in Table IV.

A. Update Protocols

In the following section, we review the different approaches for ingesting updates in graph processing and streaming.

a) Update Ingestion

Single update queries refer to the insertion or removal of single edge or vertex at a time, while *batch updates* refer to the grouping of the updates in a batch before applying them all at once. Essentially, the pre-processing on update batches before applying them (e.g., reordering, sorting, and partitioning), allows to reveal their inherent parallelism. For instance, the sorting allows grouping all the edge updates of a specific vertex together, separating deletions from insertions, which allows to run edge updates in parallel for separate vertices which improves the rate of update ingestions [11] [9] [51].

There exists two modes 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. The approach used by systems [35][51][8] , 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 [52], 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.

b) Update Storage

When applying the mutations, changes can be applied in-place (i.e., incorporated into the main structure) or stored in additional data structures called deltas [53]. In the following we review how graph processing and streaming systems implement update storage using these two approaches. In-place Updates: 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.

B. Single vs Batch Updates

Essentially, single updates are challenging to support for two main reasons. First, in most cases [51], [17] , and especially in deletion workloads, the system needs to perform a search over the neighbors of a vertex upon every edge insertion, which is not possible to do in parallel. This make system's performance very slow. Second, depending on the availability of memory, systems need to allocates new blocks to store the new edges [17], [30]. 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 [17], [26], [34], [50], opt for batch updates where the batch of edge updates is pre-processed. 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 [12]. 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 [30], [26], [17], 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.

C. Bulk vs Concurrent Updates

Systems [17][25][51] 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 [24] 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, systems that employ the bulk mode

TABLE III. Summary of the characterization of systems included in this study based their techniques for efficient graph storage.

Dimension	Impact on Perf.	Systems
Memory Layout	Cache-friendly data structures	[47][48]
Data Compression	Small memory footprint and better cache locality	[34][45]
Partitioning	Distributed processing and load balancing	[32][49][46]
Memory Allocators	Parallel allocation, low memory fragmentation, high cache performance	[17] [42][26][24][35]

have limited usage, since in real-world scenarios, graph users are constantly updating and running analytic queries concurrently. In the case where update/read happen 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. Essentially, updates modify the data, while queries retrieve information from the data [13]. However, if updates and queries are allowed to execute concurrently without any synchronization or control mechanisms, queries may observe partial or inconsistent states of the data, leading to incorrect or outdated results. Therefore, maintaining data integrity while permitting high update throughput is a major challenge for systems designed to support concurrent updates and queries [54] [13][30][34].

In the following, we review different approaches used in practice, to allow concurrent updates and queries and observe that a lot of systems [13][29][30][50][34][42] implement protocols to maintain data coherence and consistency between multiple readers and writers through different isolation levels [52], which is similar to traditional database systems.

1) Hybrid Store

One way to achieve concurrent analytic workloads and update workloads, is by creating a hybrid graph representation that uses separate data structures: one for the incoming updates and another another structure optimized for reads 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 [13], which creates a new delta, a.k.a., snapshot, every time the user runs a batch of updates. This technique enables readers to have parallel access to the previously created snapshots and run analytics and queries in the read-optimized store without interfering with the newer update queries performed in a separate structure called the write-optimized store. However as shown in recent work [30], [17], the continuous creation of new snapshots leads to a steep increase in memory. Another example is GraphOne [29], which implements a hybrid store for snapshots using adjacency lists (AL) store and an edge list (EL). the AL keeps track of a linked list of vertex degrees at various points in time using timestamps. However, the performance of GraphOne suffers by the indirection layer and the multiple levels of data in adjacency list as [30] points out, making it not reliable for read-intensive workloads.

2) Concurrency Control

Another notable way used in literature is to use concurrency control models to achieve concurrent reads and writes in graph processing and streaming. For instance, a popular model is the Multiversion Concurrency Control (MVCC) [55] used by transactional systems such as Teseo[30] to achieve Snapshot Isolation. This model prevents data corruption by updating graph data in-place, hence preserving the contiguous memory placement of data, and by utilizing a transaction-based validation method. Essentially, these systems use 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 pictures 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). The primary benefit of this method is that it enables scan performance similar to that of single-version systems while also providing fine-grain transaction scheduling. However, garbage collection, which must be conducted without interrupting the queries, adds an additional expense. Actually, both the performance and the resources needed to execute the version pruning may be slowed down due to the update granularity and update frequency.

D. In-place Updates

This means that, most systems [17][9][30] 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.

First, this can be done using growable dynamic arrays [8], [17], [11], where edge insertions are performed by directly storing the new edges in dynamic growable edge arrays and reallocating twice the initial array size if there is no memory space for the new edge. For instance, NetworKit [8] performs edge insertions by directly storing the new

edges in dynamic growable edge arrays and reallocating twice the initial array size if there is no memory space for the new edge. The same method is employed by the Madduri et. al. [11], with the exception that the size of the new edge array is defined in terms of a customizable factor rather than a fixed 2. Subsequently, when employing dynamic arrays, the amortized 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. Second, Packed Memory Arrays (PMAs) are another classic technique in data structures that allows in-place updates by maintains 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 to keeps track of gaps within regions of the array. For instance, [30], [25] develop a variant of CSR based on PMAs called PCSR, that provides efficient single-threaded mutations by leaving space at the end of each adjacency list. Moreover, 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 slowdown the update performance and delay the analytic workloads.

Finally, regarding the deletions of graph entities, there are two main approaches: physical deletions and logical deletions. In the latter, systems [17] enhance vertex and edge data with flags that can be set in the event that 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 [26].

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 [29][26][30] 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, therefore better read performance.

E. Delta Updates

As previously mentioned, data structures such as CSR are optimized for scans and need to be rebuilt to support even singular updates, which require O(n+m) space and time. One way to extend CSR to support fast updates is by employing *deltas*, by allocating a separate structure to store only the new changes in delta maps or vertex/edge logs. For instance, edge lists [29] 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 other they arrive. In practice, to support deltas, systems [26][29] tend to design separate structures: one for the original graph, usually referred to as the read optimized store (usually CSR based) and another structure (usually edge lists) for the new updates called write-optimized store which practically refers to a delta. Subsequently, by using deltas, systems [29][13][30] can run analytical workloads on different static versions (snapshots) of the changing graph over time, while applying updates. However, the downside of storing updates in delta maps or edge lists is two-fold. First, maintaining separate structures for each batch of update 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 [17] which can be fatal for real time systems.

Finally, to remediate this problem, these system perform compaction on the frequently created data structure into one main structure. For this purpose, several authors [29][34][30] 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." To illustrate this compaction, we cite GraphOne which stores newly-added edges in a circular log and uses adjacency lists as main read optimized store, which provide a coarse-grained snapshot method. First, GraphOne establishes an edge log cutoff for the "transfer" of the edges from the buffer to the base structure in order to preserve the separate store for the update operations and achieve good analytic performance. Essentially, when the number of newly created edges in the log grows too large, the system *archives* the older entries by copying them to a more permanent location, which is the adjacency list. 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. In fact, it is shown in [30] that GraphOne's iterator architecture is to blame for its subpar performance in read workloads. Point look-ups 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 neighbors into an intermediate buffer.

In summary, compaction is computationally demanding. Compaction can become very expensive, often zeroing the mutability performance benefits of these structures. For users that 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 an alternative to achieving better analytics and update performance with lower memory

TABLE IV. Summary of the characterization of systems included in this study based their techniques for efficient graph updates.

Technique	Category	Advantages	Systems	Systems
Single Upd.	Ingestion	Fine-granularity of updates	Requires more CPU and memory	[17][24][25]
Batch Upd.	Ingestion	Fast update, load balancing	Requires extra pre-processing	[17][26][34][30][29]
Upd. in Bulk	Ingestion	Fast update throughput, fast analytics	No concurrency	[17][25][24][35][11]
Concurrent Upd.	Ingestion	Parallel updates and reads	High memory footprint, slow updates	[26][34][30][29]
In-place Upd.	Storage	Low memory footprint, fast analytics	Slow updates	[17][25]
Delta Upd.	Storage	Multi-versioning, less resources	Slow analytics, high memory footprint slow updates	[56][57][50][30][29]

requirements.

6. DISCUSSION & SUMMARY

In our attempt to answer our research questions, we have discussed the techniques for graph processing and streaming, by giving an overview of representative graph data structures then the protocols and algorithms for graph updates. Table V presents a classification of the most popular graph systems reviewed in our paper, based on the dimensions discussed previously. In this section, we summarize key insights about the research on high efficiency graph representation and update protocols.

First, from Table V, we conclude that a lot of systems rely significantly on CSR, because it is cache-friendly and uses less memory. However, they use optimizations such as hardware/distributed systems [49][32] as well as extra data structures to store the updates. We also have discussed a trend in using multiple representations for the same graphs in memory, though GPS still offer very limited *configurability* for their data structures.

Second, we point out the large number of shared-memory systems, indicating the continued need for enhancements in that area to achieve the same level of performance as distributed systems while making better use of available hardware. In the same context, we note that there is very limited research work on the graph streaming in Distributed systems, especially the lack of protocols for update ingestion and storage designed specifically for Distributed infrastructures. This also implies that even though the studied systems guarantee performance for a certain hardware configuration, they do not ensure whether or not their data structures are portable to different infrastructures. Thus, there is a need for "portability" of solutions which have high-performance design of a graph representation and update protocol, and that can be easily implemented in single machines as well as in distributed environments.

Moreover, we have shown that the choice of the optimization techniques, in either graph representation or updates depend on the specific characteristics of the graphs, the types of workloads, and the constraints of the application environment. However, there is still ongoing research to achieve high performance, and as discussed and highlighted in Table V, most systems tend to improve on an aspect of performance, either read performance, updates throughput or memory consumption, and to find the barely attempt to provide 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. Finally, there is a high emphasis on protocols for updating the topology of graphs, but it is rare when other graph data such as graph properties, reverse edges or labels, are implemented and thoroughly evaluated for performance.

7. RELATED WORK

The prevalence of graph processing and streaming in various domains have prompted researchers to conduct reviews to understand how graphs are used in practice [15] [20]. Joaquim et. al. [58] review graph systems and classifies them based on their infrastructure. The review expands on specific abstractions such as programming models for distributed systems, and some of them are built on top of distributed dataflow frameworks such as MapReduce and Pregel.

Furthermore, Wheatman et al. [25] 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 literature aim to optimize the performance of these graph data structures.

Finally, [15] present a general survey of graph processing systems and streaming, where they define the landscape and taxonomy of the graph technology. Their work provides descriptions and analysis of different approaches for representing graphs in a streaming context. However they do not clearly analyze the performance trade-offs of different approaches for graph storage and mutation.

8. CONCLUSION

Streaming and dynamic graph processing is an important research field. It is used to maintain numerous dynamic graph datasets, simultaneously ensuring high-performance graph updates, queries, and analytics workloads. Many graph processing and streaming systems have been developed and use different data representations for fast parallel ingestion of updates and a plethora of graph queries and workloads. In this paper, we present insights on the growing interest in literature in the optimization of classical graph data structures for high analytic and update performance while minimizing the memory footprint. We give an overview of techniques available for researchers and developers, namely optimizing the memory layout of data structures, compression, using memory allocator software, and partitioning, and discuss how they're used to enhance graph processing and streaming. Moreover, we present the update protocols and the performance implications of different designs. Finally, we conclude that there is still vast ongoing research for optimal GPS that gives the best trade-off between the three important aspects of performance, namely the read, update and memory consumption.

REFERENCES

[1] "Gartner top 10 data and analytics trends for 2019," https://www.gartner.com/smarterwithgartner/gartner-top-10-data-analytics-trends/.

[2] P. Boldi and S. Vigna, "The webgraph framework i: Compression techniques," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 595–602.

[3] T. Ben-Nun, M. Besta, S. Huber, A. N. Ziogas, D. Peter, and T. Hoefler, "A modular benchmarking infrastructure for high-performance and reproducible deep learning," 2019.

[4] L. Di Paola, M. De Ruvo, P. Paci, D. Santoni, and A. Giuliani, "Protein contact networks: an emerging paradigm in chemistry," *Chemical reviews*, vol. 113, no. 3, pp. 1598–1613, 2013.

[5] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proc. VLDB Endow.*, vol. 8, no. 12, p. 1804–1815, aug 2015. [Online]. Available: https://doi.org/10.14778/2824032.2824077

[6] M. E. Coimbra, A. P. Francisco, and L. Veiga, "An analysis of the graph processing landscape," *journal of Big Data*, vol. 8, no. 1, pp. 1–41, 2021.

[7] B. Tulasi, R. S. Wagh, and S. Balaji, "High performance computing and big data analytics a [euro]" paradigms and challenges," *International Journal of Computer Applications*, vol. 116, no. 2, 2015.

[8] C. L. Staudt, A. Sazonovs, and H. Meyerhenke, "NetworKit: a tool suite for large-scale complex network analysis," *Network Science*, vol. 4, no. 4, p. 508–530, 2016.

[9] B. Wheatman and H. Xu, "Packed compressed sparse row: a dynamic graph representation," in *HPEC*, 2018.

[10] R. Cheng, E. Chen, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, and F. Zhao, "Kineograph: taking the pulse of a fast-changing and connected world," in *EuroSys*, 2012.

[11] K. Madduri and D. A. Bader, "Compact graph representations and parallel connectivity algorithms for massive dynamic network analysis," in *IPDPS*, 2009.

[12] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: large-scale graph computation on just a PC," in *OSDI*, 2012.

[13] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "LLAMA: efficient graph analytics using large multiversioned arrays," in *ICDE*, 2015.

[14] P. V and J. Jayakumar, "Challenges and opportunities with big data," *International Journal of Engineering Research And*, vol. V6, 2017.

[15] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, "Practice of streaming and dynamic graphs: Concepts, models, systems, and parallelism," *arXiv*, pp. 1912–12 740, 2020.

[16] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world'networks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[17] S. Firmli, V. Trigonakis, J.-P. Lozi, I. Psaroudakis, A. Weld, D. Chiadmi, S. Hong, and H. Chafi, "Csr++: A fast, scalable, update-friendly graph data structure," in *24th International Conference on Principles of Distributed Systems (OPODIS'20)*, 2020.

[18] U. Drepper, "What every programmer should know about memory," *Red Hat, Inc*, vol. 11, no. 2007, p. 2007, 2007.

[19] "Neo4j," http://www.neo4j.org.

[20] S. Firmli and 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 3*, pp. 214–223, 2020.

[21] J. A. Storer, *Data compression: methods and theory*. Computer Science Press, Inc., 1987.

[22] K. Lee and L. Liu, "Efficient data partitioning model for heterogeneous graphs in the cloud," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[23] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," *ACM Sigplan Notices*, vol. 35, no. 11, pp. 117–128, 2000.

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

[25] B. Wheatman and H. Xu, *A Parallel Packed Memory Array to Store Dynamic Graphs*, pp. 31–45. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611976472.3

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

[27] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and

TABLE V. 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 multiversioning; 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.	DS	SU	Batch	MV	Compact.	Up. Store	Scans	Mem	Up. Perf.
BGL	SM	AL	+	-	-	+	IP	-	-	+
PGX.SM	SM	CSR	-	+	+	-	D	+	+	-
Ligra	SM	CSR	-	-	-	-	X	+	+	-
GraphLab	Dist	CSR	-	-	-	-	X	-	+	-
PGX.D	Dist	CSR	-	+	+	-	D	+	+	-
STINGER	SM/Dist	AL	+	+	-	-	IP	-	-	+
Hornet	GPU	AL	+	+	-	-	IP	-	+	+
Compact	SM	AL	-	+	-	+	IP	-	-	+
PCSR	SM	CSR	+	-	-	+	IP	+	-	+
LLAMA	SM	CSR	-	-	+	+	D	+	-	-
Metall	SM	AL	-	+	+	+	D	-	+	+
GraphOne	SM	AL + EL	+	+	+	+	D	-	+	+
Aspen	SM	Tree	-	+	+	+	IP	-	+	+
Teseo	SM	Tree	+	+	+	+	IP	+	-	+

J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[28] "Boost adjacency list documentation," https://www.boost.org/doc/libs/1_67_0//libs/graph/doc/adjacency_list.html.

[29] P. Kumar and H. H. Huang, "GraphOne: a data store for real-time analytics on evolving graphs," *ACM Trans. Storage*, vol. 15, no. 4, 2020. [Online]. Available: https://doi.org/10.1145/3364180

[30] D. De Leo and P. Boncz, "Teseo and the analysis of structural dynamic graphs," *Proc. VLDB Endow.*, vol. 14, no. 6, p. 1053–1066, 2021. [Online]. Available: https://doi.org/10.14778/3447689.3447708

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

[32] S. Hong, S. Depner, T. Manhardt, J. van der Lugt, M. Verstraaten, and H. Chafi, "PGX.D: a fast distributed graph processing engine," in *SC*, 2015.

[33] M. A. Bender and H. Hu, "An adaptive packed-memory array," *ACM Trans. Database Syst.*, vol. 32, no. 4, p. 26–es, nov 2007. [Online]. Available: https://doi.org/10.1145/1292609.1292616

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

[35] F. Busato, O. Green, N. Bombieri, and D. A. Bader, "Hornet: an efficient data structure for dynamic sparse graphs and matrices on GPUs," in *HPEC*, 2018.

[36] N. P. Roth, V. Trigonakis, S. Hong, H. Chafi, A. Potter, B. Motik, and I. Horrocks, "PGX.D/Async: a scalable distributed graph pattern matching engine," in *GRADES*, 2017.

[37] B. Wheatman and R. Burns, "Streaming sparse graphs using efficient dynamic sets," in *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, 2021, pp. 284–294.

[38] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proc. of the bsdcan conference, ottawa, canada*, 2006.

[39] A. H. Hunter, C. Kennelly, D. Gove, P. Ranganathan, P. J. Turner, and T. J. Moseley, "Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator," in *OSDI*, 2021.

[40] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: Solved?" *ACM Sigplan Notices*, vol. 34, no. 3, pp. 26–36, 1998.

[41] D. Alistarh, W. Leiserson, A. Matveev, and N. Shavit, "Threadscan: Automatic and scalable memory reclamation," *ACM Transactions on Parallel Computing (TOPC)*, vol. 4, no. 4, pp. 1–18, 2018.

[42] K. Iwabuchi, K. Youssef, K. Velusamy, M. Gokhale, and R. Pearce, "Metall: A persistent memory allocator for data-centric analytics," *Parallel Computing*, vol. 111, p. 102905, 2022.

[43] B. C. Kuszmaul, "Supermalloc: A super fast multithreaded malloc for 64-bit machines," in *Proceedings of the 2015 International Symposium on Memory Management*, 2015, pp. 41–55.

[44] S. Phansalkar and S. Ahirrao, "Survey of data partitioning algorithms for big data stores," in *2016 Fourth International Conference*

on Parallel, Distributed and Grid Computing (PDGC). IEEE, 2016, pp. 163–168.

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

[46] A. Kyrola, G. Blelloch, and C. Guestrin, "{GraphChi}:{Large-Scale} graph computation on just a {PC}," in *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, 2012, pp. 31–46.

[47] "Oracle Parallel Graph AnalytiX (PGX)," https://www.oracle.com/middleware/technologies/parallel-graph-analytix.html.

[48] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi, "PGQL: a property graph query language," in *GRADES*, 2016.

[49] A. P. Iyer, Q. Pu, K. Patel, J. E. Gonzalez, and I. Stoica, "TEGRA: Efficient Ad-Hoc analytics on evolving graphs," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 337–355.

[50] M. Haubenschild, M. Then, S. Hong, and H. Chafi, "Asgraph: a mutable multi-versioned graph container with high analytical performance," in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, 2016, pp. 1–6.

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

[52] A. Adya, B. Liskov, and P. O'Neil, "Generalized isolation level definitions," in *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*. IEEE, 2000, pp. 67–78.

[53] S. Firmli and D. Chiadmi, "A review of engines for graph storage and mutations," in *EMENA-ISTL*, 2020.

[54] M. Haubenschild, M. Then, S. Hong, and H. Chafi, "ASGraph: a mutable multi-versioned graph container with high analytical performance," in *GRADES*, 2016.

[55] J. Böttcher, V. Leis, T. Neumann, and A. Kemper, "Scalable garbage collection for in-memory mvcc systems," *Proceedings of the VLDB Endowment*, vol. 13, no. 2, pp. 128–141, 2019.

[56] "LLAMA code," https://github.com/goatdb/llama.

[57] R. Raman, O. van Rest, S. Hong, Z. Wu, H. Chafi, and J. Banerjee, "PGX.ISO: parallel and efficient in-memory engine for subgraph isomorphism," in *GRADES*, 2014.

[58] P. Joaquim, "Hourglass-incremental graph processing on heterogeneous infrastructures."

Soukaina Firmli is a PhD student at University Mohammed V in the SIP Research Team, under the joint supervision of Professor Dalila Chiadmi and Vasileios Trigonakis. She worked as a Research Assistant in Oracle Labs within The PGX team and her research interest include Data Structures, Graph Databases and Parallel Processing. Previously to her PhD studies, she obtained a Diplome D'Ingenieur from Mohammadia School of Engineers, equivalent to a master's degree. Soukaina enjoys reading, volleyball and music in her free time.

Dalila Chiadmi is a Full professor at Mohammadia School of Engineers -EMI-, Mohammed V University in Rabat, Morocco. She got her PhD degree in computer science from EMI in 1999. Her main research interests focus on the field of data (Open, linked and Big Data, Big graphs, etc.). She is currently leading the SIP Team Research at EMI and she was appointed leader of the Moroccan Chapter of the Arab Women in Computing Anita Borg Institute Systers' community in 2012. She has co-authored a book and over 75 refereed book chapters, conference and journal publications. She has served in many conferences as chair, member of the program committee, or reviewer.