A Configurable Framework for High-Performance Graph Storage and Mutation

Soukaina Firmli, Dalila Chiadmi, Kawtar Younsi Dahbi
SIP Research Team-Rabat IT Center-EMI, Mohammed V University in Rabat, Morocco

***Abstract*—In the realm of graph processing, efficient storage and update mechanisms are crucial due to the large volume of graphs and their dynamic nature. Traditional data structures such as adjacency lists and matrices, while effective in certain scenarios, often suffer from performance trade-offs such as high memory consumption or slow update capabilities. To address these challenges, we introduce CoreGraph, an advanced graph framework designed to optimize both read and update performance. CoreGraph leverages a novel segmentation method and in-place update techniques, along with configurable memory allocators and synchronization mechanisms, to enhance parallel processing and reduce memory consumption. CoreGraph's update throughput (with up to 20x) and analytics performance exceed those of several state-of-the-art graph structures such as Teseo, GraphOne and LLAMA, while maintaining low memory consumption when the workload includes updates. This paper details the architecture and benefits of CoreGraph, highlighting its practical application in traffic data management where it seamlessly integrates with existing systems providing a scalable and efficient solution for real-world graph data management challenges.**

***Keywords*—Data structures; concurrency; graph processing; graph mutations; high-performance computing; traffic management**

I. Introduction

Research in graph processing technology continues to evolve and prosper due to the unique ability of graphs to represent complex relationships and inter-dependencies within data, making them suitable for many modern applications. Graph applications include Knowledge Graphs (KG) [1] used in search engines, personal assistants, and recommendation systems; Graph Neural Networks (GNNs) [2] for AI tasks like node classification, link prediction, and graph classification; and real-time graph analysis for streaming data from sources like Twitter and financial transactions.

As a result of the widespread use of graphs, there is a growing need to efficiently manage and analyze them. This has led to the development of graph processing systems and graph databases like Neo4j [3], which are well-suited for storing, analyzing, and streaming large graph data due to their scalability, real-time processing capabilities, and ability to handle complex relationships. However, these systems face multiple challenges inherent to graph processing and streaming that limit the effective use of graphs in the real world due to varying graph characteristics, memory-intensive algorithms, and access patterns that cause latency, as well as the continuous evolution of graph topology and properties [4].

To understand and address these challenges, it is important to recognize that at the heart of each graph system lies the graph data structure, which stores the vertices and their edges, and whose performance largely contributes to the overall performance of the system. However, classic graph data structures typically trade some characteristics for others [5]. For instance, Compressed Sparse Row (CSR) representations are memory efficient with good read-only performance but have poor update performance. The challenge, therefore, is to design graph data structures that ideally offer excellent read-only performance, fast mutations (i.e., vertex or edge insertions and deletions), and low memory consumption with or without mutations.

To address these problems with classical data structures, efforts have been made to improve the updated friendliness of CSR. These include in-place update techniques [6], [7], batching techniques [8], [9], changeset-based updates with delta maps [10], and multi-versioning [11]. However, systems still struggle to offer the best tradeoff between read and, update performance and memory consumption.

To this aim, we present in this paper CoreGraph, a highly configurable end-to-end graph framework designed to address these challenges. CoreGraph builds on our previous work CSR++ [12], to offer a high-performance concurrent data structure for graph topology and properties storage. The framework is designed for in-place graph mutations, achieving superior analytics and update performance with significantly lower memory requirements compared to state of art solutions like LLAMA [11].

The main contributions of our framework can be summarized as follows:

1) Efficient in memory architecture: CoreGraph implements an in-memory architecture that allows for efficient graph loading, mutation, and analysis, making it suitable for a wide range of applications.

2) Optimized storage and update protocol: CoreGraph offers a storage and update protocol based on a novel segmentation method that optimizes graph storage for both read and write operations while minimizing memory space.

3) High configurability: CoreGraph is highly configurable by integrating advanced synchronization mechanisms, including Hardware Transactional Memory (HTM) and adaptable locking strategies, to enhance parallel performance while minimizing contention. Furthermore, the framework supports configurable memory allocators to optimize resource utilization and performance based on specific application needs.

We also demonstrate the practical applicability and portability of CoreGraph as a lightweight graph storage and mutation framework through a case study on traffic data manage-

ment, a domain characterized by high-frequency updates and large data volumes. CoreGraph not only improved the performance of graph mutations and analytics but also seamlessly integrated with an existing framework for traffic data management. This integration highlights CoreGraph's potential as a vital component in real-world data discovery and exploitation chains.

The rest of this paper is organized as follows: Section II presents related works, Section III gives an overview of the proposed approach for graph storage and mutation, and Section V presents the use case related to the traffic data management domain, which aims to integrate CoreGraph with an existing framework for graph analysis and updates. The study concludes in Section VI.

II. Related Work

In this section, we present some notable state-of-the-art research work that proposes optimization to classic data structures for graph storage and mutation for graph processing and streaming. We give a brief overview of each work in terms of i) graph storage and ii) graph updates, and then draw their main limitations.

A. Graph Storage

First, their exists different techniques available for researchers to optimize data structures, among which we found: optimizing the memory layout of the data structures [27] [9] [25], using compression algorithms [26] [24] and using special memory allocator [19] [18].

SSTGraph [26] use a compression algorithm 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, by minimizing serialization overhead.

To improve the cache performance of dynamic graph data structures, many researchers [18] [19][20] 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. 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 [25].

Another approach used by systems [19] [9] [18] in literature is using a special memory allocator, either internal to their system or external, to minimize the memory fragmentation caused by frequent reallocations on dynamic data structures. For instance, to efficiently perform memory reclamation and manage space, Hornet's [19] internal memory management uses a B+ tree for insertions and deletions to keep track of the available blocks of edges. However, when deletions are not frequent, which is the case in most real-world scenarios, the overhead of the memory reclamation makes the update performance slower.

B. Update Protocols

The ingestion and storage of the new updates play a crucial role in the overall performance of the systems. The methods used to implement them, that we refer to as update protocols, vary in the literature, therefore, we propose our classification for these protocols. First, update storage can be either in-place or using deltas, while update ingestion can be in bulk or concurrently with analytic workloads.

1) Update storage: Techniques that use in-place updates employ the static data structures in a way that allows for in-place digestion of sets with insertions and deletions of vertices and edges, without requiring the expensive rebuilding of the data structure. Systems [28] [20] [25], develop a variant of CSR based on Packed Memory Arrays (PMAs), that provides efficient in-place updates 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.

Dynamic Arrays are used by systems like NetworKit [6] to perform in-place 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. [9]. The author in [11] with the exception that the size of the new edge array is defined in terms of a customizable factor rather than a fixed factor of two. 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.

On the other hand, systems [10] [29] [23] [30] employ additional data structures to store the new updates referred to as *deltas*. GraphOne [23] implements a hybrid store for deltas 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 because of the indirection layer and the multiple levels of data in the adjacency list as [12] points out, making it not reliable for read-intensive workloads.

2) Update ingestion: The ingestion of updates can be performed in bulk, with alternating phases: pending updates wait for currently executing queries to finish before they start executing, and pending queries wait for currently executing updates to finish before they start executing. Update ingestion can also be concurrent with query execution, but in that case, the system needs to guarantee consistency on the data and at user level [11].

Moreover, the updates can be ingested as single operations or in batches. First, supporting single updates can be challenging. In fact, depending on the availability of memory, systems need to allocate new blocks to store the new edges [18]. 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 [25] [11] [23], opt for batch updates where the batch of edge updates is pre-processed to allow for parallel updates, and to reduce the system calls for frequent allocations.

C. Discussion

The related works provide a background for optimizing graph representations for graph storage and updates. Table I

TABLE I. SUMMARY OF REVIEWED SYSTEMS AND THEIR SUPPORTED CAPABILITIES. INFRA.: SINGLE MACHINE (SM) OR DISTRIBUTED (DIST); DS: DATA STRUCTURES; SU: SINGLE UPDATES; BATCH: BATCH UPDATES; MV: MULTIVERSIONING; COMPACT: COMPACTION; UP. STORE: IN-PLACE (IP) OR DELTA (D) STORAGE; 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[13]	SM	AL	+	-	-	+	IP	-	-	+
PGX.SM[14]	SM	CSR	-	+	+	-	D	+	+	-
Ligra[15]	SM	CSR	-	-	-	-	X	+	+	-
GraphLab[16]	Dist	CSR	-	-	-	-	X	-	+	-
PGX.D[17]	Dist	CSR	-	+	+	-	D	+	+	-
STINGER[18]	SM/Dist	AL	+	+	-	-	IP	-	-	+
Hornet[19]	GPU	AL	+	+	-	-	IP	-	+	+
Compact[9]	SM	AL	-	+	-	+	IP	-	-	+
PCSR[20]	SM	CSR	+	-	-	+	IP	+	-	+
LLAMA[21]	SM	CSR	-	-	+	+	D	+	-	-
Metall[22]	SM	AL	-	+	+	+	D	-	+	+
GraphOne[23]	SM	AL + EL	+	+	+	+	D	-	+	+
Aspen[24]	SM	Tree	-	+	+	+	IP	-	+	+
Teseo[25]	SM	Tree	+	+	+	+	IP	+	-	+
SSTGraph[26]	SM	AL	-	+	-	-	IP	-	-	+

presents a summary of our related work and their limitations, which we discuss below.

1) Achieving optimal performance trade-off: As discussed and highlighted in Table I, 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 struggle to support several versions of the graph when storing updates in deltas, because of high memory cost, and graph compaction [11] is necessary to reduce the amount of space needed for changes. However, the performing frequent compaction requires expensive computation and slows down the performance of the system.

2) Lack of configuration and hybrid representation: systems still exhibit limited configurability in their data structures. This limitation underscores the need for more adaptable data structures to accommodate diverse workload requirements, especially by considering different sizing of buckets within structures facilitating in-place updates. A noticeable absence of configurability in current literature highlights the need for refining existing designs to better suit diverse workloads tailored to specific application requirements.

3) Lack of comprehensive experimental study: 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, and a data storage-focused approach is vital for achieving high performance in real-world applications. We also highlight the lack of validation of these graph processing systems in real-world scenarios, as most are only evaluated against generic data sets or synthetic data data sets that are not diversified.

Moreover, The performance of graph systems is significantly influenced by the dynamic memory allocator employed. To our knowledge, there are only few studies [31] that provide experimental analysis on how memory allocation impacts high-performance query engines, but not for graph storage and mutation systems.

Therefore, in this work, we aim to address these challenges by proposing a generic and domain-independent system for graph storage and mutation.

III. PROPOSED FRAMEWORK

In this section, we present our highly configurable end-to-end graph framework CoreGraph that allows the loading, storage and mutation of graphs, as well as their analysis, while providing high performance, low memory consumption and high update throughput. The architecture of CoreGraph is shown in Fig. 1.

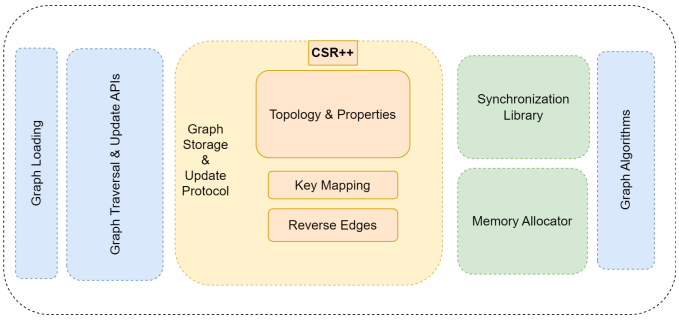

Fig. 1. CoreGraph framework.

CoreGraph enables fast concurrent accesses to the main graph data (vertex and edge tables) and stores additional graph data, such as reverse edges, user-defined keys, and vertex and edge properties. Additionally, it supports graph analytics kernels based on Green-Marl implementation [32], which are optimized to run in parallel (Section III-A).

Moreover, CoreGraph offers the main capabilities to load graphs in-memory using smart allocation and mutate graphs by exposing storage and update APIs, as well as a high performance and low memory footprint storage (Section III-B),

Finally, CoreGraph offers high configurability based on specific requirements and workloads by providing i) a configurable reallocation size for edge arrays to reduce the memory footprint when there are less frequent edge insertions (Section III-B), ii) configurable synchronization mechanism to enhance parallelism and reduce the overhead of contention (Section III-C), and iii) configurable memory allocators to optimize performance and resource utilization (Section III-D).

The main elements of our framework are represented in Fig. 1, and a detailed description will be presented below.

A. Graph Storage

The storage component of CoreGraph is a concurrent multimap that maintains the graph's topology and properties, based on the segmentation method where a fixed number of keys are grouped in an array referred to as segment. The segmentation method enables a less bulky, cache-friendly layout. Additionally, CoreGraph maintains additional graph data in addition to the basic graph data (vertex and edge tables), allowing for rapid algorithms. By storing extra graph data like reverse edges and user-defined keys, it can handle directed graphs and external IDs.

1) Topology: We store vertices in *segments* and the number of vertices stored in each segment of a graph is determined by the global configuration NUM_V_SEG. When there are N vertices in a graph, the number of segments is: num_segment = N / NUM_V_SEG + 1, as shown in Fig. 2(A). As for the structure of vertices, CoreGraph stores i) the degree ii) a pointer to their neighbor list if the degree is more than 1 and iii) optionally a pointer to the edge properties as shown in Fig. 2(B). This structure is similar to a combination of CSR and adjacency lists; However, CoreGraph's storage performance is superior to other adjacency-inspired approaches for skewed graphs due to the *inlining* of the single edges in the vertex structure, meaning if a vertex has only one neighbor, then the edge corresponding to said neighbor is stored within the structure of the vertex instead of a pointer, which minimizes the memory consumption and the cache misses when accessing the edge. Finally, if a vertex has more than one neighbor, then it's adjacency is stored as an array of edges; each edge contains the vertex ID corresponding to the index of the neighbour in the segment, and segment ID of its corresponding neighbor.

A complete sensitivity analysis [12] of the segment size shows that setting the value depends on the workload, since 1) for read intensive workloads, cache performance decreases if the size of segments is small, and the frequently allocated segments are not necessarily be allocated contiguously in memory. However, for update intensive workloads, contention between threads updating vertices in the same segment may increase, if the NUM_V_SEG is set to a big value, causing a decrease in the update throughput. Therefore we expose this option in the configuration of CoreGraph for users to set the The NUM_V_SEG so that they get the best performance depending on their workload. when configuring the segment layout. CoreGraph sets the default segment size to a value that is a multiple of 4 to enable cache alignment [4].

2) Properties: CoreGraph allows the user to configure the storage to store extra pointers for the vertex and edge properties. If the to-be-loaded graph data set does not include

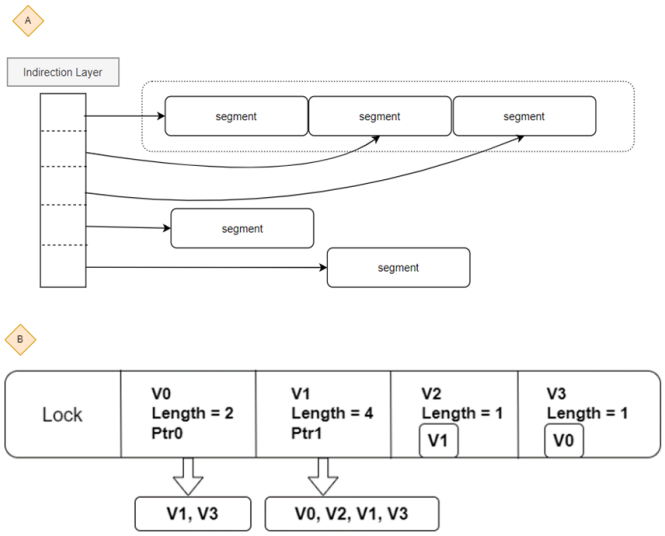

Fig. 2. Representation of Topology in CoreGraph. A) An indirection layer stores pointers to the segments to allow adding new ones and the segments are initially stored contiguously upon the loading of the graph to enhance the read performance. B) The segment structure stores a fixed number of vertices which in turn store information about their adjacencies.

vertex/edge properties, then property support can be disabled to save memory. If activated, CoreGraph keeps a vector of pointers to vertex-property arrays which are parallel to the vertex array. Similarly, if edge properties are activated, CoreGraph stores a pointer to an array of edge-property values within each vertex structure and we use the same segmentation approach as for edges. In case of multiple properties, we allocate an array that stores the values for different edge properties in a cache-aligned manner. The edge properties are stored separately from the edge arrays to enable copy-on-write functionality for the edge-property arrays of only the updated vertices, unlike CSR which requires rebuilding the edge properties for the entire graph. This approach also makes it easier to maintain the property values in the same order as the edges if sorting is needed after an update. While this design incurs a moderate memory overhead, it offers significant benefits for performance, as the performance of the insertion of new edges with an edge properties of CoreGraph is an order of magnitude faster than other state-of-the-art in-place update systems, namely, Teseo [25] and STINGER [18].

B. Update Protocols

For fast, low-memory-usage graph mutation, CoreGraph offers update APIs as abstractions for an update protocol. The update protocol includes creating new vertices and edges, removing existing ones, and modifying the vertex and edge properties. When compared to CSR, which must reallocate and copy edge and vertex arrays whenever the graph's topology is changed, our segmented approach offers significantly higher update throughput due to the fact that each vertex and edge can be updated independently during graph mutation. Vertices in CoreGraph are kept in indexed arrays (see Fig. 3). If there is free space in a segment, adding a new vertex costs O(1), otherwise CoreGraph creates a new segment if necessary. However, in CoreGraph, adding a new vertex is faster reaching an update throughput of 9M vertices/s due to the segmented

nature of the vertex array; that is, we do not need to replicate the entire vertex array when adding or removing entries.

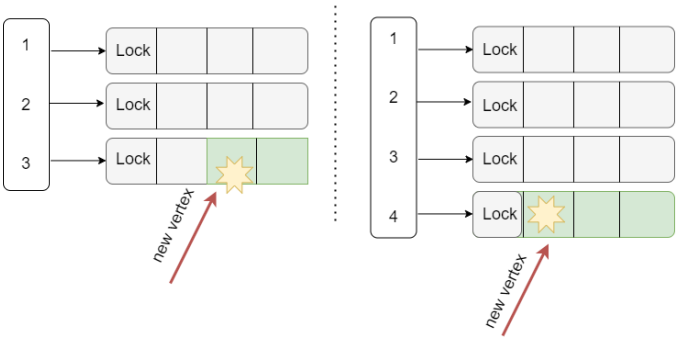

Fig. 3. Update Protocol for vertex insertions. If there is space in the segment, CoreGraph sets the length value of the first non-valid vertex from -1 to the degree of the new vertex (left), otherwise, it allocates a new segment and sets the new vertex (right).

Moreover, adding a new edge only necessitates reallocating the edge array of the source vertex because edges are stored in an array per vertex as shown in Fig. 4. 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. CoreGraph addresses this problem in two ways.

First, we use a smart memory manager 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.

Second, CoreGraph exposes the reallocation factor as a configuration option for users to set it at compile time. The reason for this is that most system only test on specific use cases such as the SNAP [9] data sets and assuming they're analyzing power-law graphs, thus they set the reallocation factor to double the size by default. However, today users can take advantage of the power of machine learning [33] to estimate the factor by which we can grow our dynamic arrays, using statistical variables in a stream for specific types of graphs. For instance, we can learn the patterns of the creation of new relationships in social networks, e.g., a post that goes viral on social media might bring many followers in a short amount of time 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. Therefore, by allowing the configuration of the reallocation factor of edge arrays, CoreGraph adapts to different types of graphs and workloads, and minimizes the overall memory consumption of its dynamic graph data structure.

With respect to deletions, CoreGraph includes a mechanism for removing entities (both physically and virtually). The system reallocates the edge array by shifting the other edges to the left to physically delete an edge. In case of virtual deletion, where an edge or vertex is logically deleted from the CoreGraph, the deleted flag is set to indicate this.

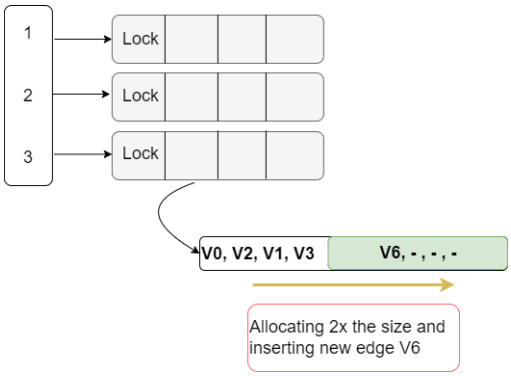

Fig. 4. Update protocol for edge insertion.

Algorithm 1 Batch Update Protocol in CoreGraph for Vertex and Edge Insertions

Input: Set of new edges E_{new}
Output: Updated CoreGraph with new edges
//Step 1: Grouping and Conversion
for all edge $e \in E_{new}$ **do**
  Group e by its source vertex $src(e)$
  Convert source and destination user keys to internal keys
end for
Insert new vertices in CoreGraph and assign new internal IDs (Sequential)
//Step 2: Sorting and Insertion
for all source vertex v in parallel **do**
  Sort new edges originating from v
  Insert sorted edges into direct and reverse maps
end for
//Step 3: Final Sorting and Reallocation
for all modified segment s in parallel **do**
  Merge old edges and new edges for each source vertex in s
  Sort the final edge arrays
  Reallocate edge properties according to the new order of edges
end for

Last but not least, CoreGraph uses a batch insertion protocol to insert vertices and edges in parallel, greatly enhancing the update's performance over a singular update. CoreGraph uses a bulk update mode, wherein updates and analytical procedures are executed sequentially rather than simultaneously. Using a fast in-place update protocol, CoreGraph is resilient enough to handle frequent small updates. Algorithm 1 shows the steps to achieve the batch insertions of vertices and edges in CoreGraph.

C. Synchronization Library

CoreGraph as a customizable graph storage and mutation framework, offers different configurations for synchronization to ensure consistency when running parallel graph analytics or parallel graph mutations. CoreGraph allows the user to configure the synchronization mechanism by offering different synchronization primitives to cater for different types of workloads and graph characteristics. CoreGraph allows locking at the segment level, however this may lead to high contention

when updates are targeting the same segment can cause a slowdown , such as multiple threads inserting edges to the same vertex in high skew graphs.

To remediate to this problem, CoreGraph allows two main configurations for low and high contention workloads. This is done by 1) using adaptable locking mechanism to switch to different lock algorithm depending on the contention level and 2) using HTM for high contention workloads to speed up the parallel execution without incurring extra memory overhead.

First, CoreGraph integrates the library GLS [34] for two reasons. It offers adaptiveness, which is designed to switch the synchronization depending on the workload, allowing it to adapt to different types of workloads and therefore give the best performance in most cases. Moreover, in debug mode it gives us information about the level of contention thanks to its built-in profiler lock (as shown in Fig. 5) which allows us to switch to different synchronization mechanism.

```
Prof Lock + RTM
Threads | Stats
1   |   [lock stats] avg spin: 0          | avg queue: 0.00  | avg lat: 24          | n_acq: 5649 @ (0x2aaaacc81298)m
2   |   [lock stats] avg spin: 16         | avg queue: 0.04  | avg lat: 72          | n_acq: 137324 @ (0x2aaaacc81298)m
4   |   [lock stats] avg spin: 47         | avg queue: 0.22  | avg lat: 560         | n_acq: 677173 @ (0x2aaaacc84298)m
6   |   [lock stats] avg spin: 94         | avg queue: 0.92  | avg lat: 1650        | n_acq: 1483192 @ (0x2aaaacc83298)m
8   |   [lock stats] avg spin: 227        | avg queue: 2.85  | avg lat: 4887        | n_acq: 1935047 @ (0x2aaaacc83298)m
12  |    [lock stats] avg spin: 707       | avg queue: 7.86  | avg lat: 15899       | n_acq: 2002555 @ (0x2aaaacc84298)m
24  |    [lock stats] avg spin: 2022      | avg queue: 20.12 | avg lat: 48325       | n_acq: 2003008 @ (0x2aaaacc8f298)m
```

Fig. 5. Real-time monitoring of contention using profiler lock in GLS and HTM. The average queue allows to quantify the contention level.

CoreGraph also allows swapping synchronization primitives automatically depending on the level of conflict: it prioritizes queue-based locks for high contention workloads such as edge insertions on vertices in the same segment and spinlocks like ticket locks for low contention workloads.

Furthermore, CoreGraph offers high parallel performance through the use of Hardware Transactional Memory (HTM) in Intel Restricted Transactional Memory (RTM) as a synchronization mechanism to simulate fine-grain locking at the vertex level without incurring extra memory cost. Each segment uses a single lock as a backup locking mechanism in case of aborts, i.e., lock elision.

We run an experiment to compare the performance of CoreGraph with HTM compared to fine-grained synchronization. We implemented of variant of CoreGraph with fine-grained synchronization where instead of storing one lock for individual segments, we store one lock per individual vertex. Update times for several segments and multithreaded scalability are shown in Fig. 6. The NUM_V_SEGMENT is modified to store the graph in different number of segments, and we insert 100K edges using a random uniform distributions on the final segments.

Fig. 6 demonstrates that even with 24 threads, the quickest solution is fine-grain locking, made possible by OpenMP scheduling. When utilizing several threads, CoreGraph with coarse locks stops scaling, and performance degrades as segment sizes increase. This is due to the significant conflict among threads updating adjacent segment vertices. On the other hand, HTM performs much better when there are many conflicts created as we use only two segments to store a whole graph.

Finally, the memory requirements of several locking mechanisms are shown in Table II. The cost of implementing HTM

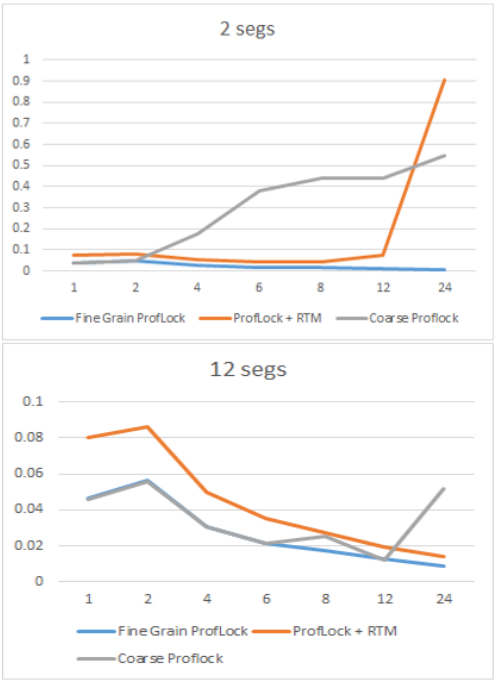

Fig. 6. Update times for several segments and multithreaded scalability. X-axis: number of threads, Y-axis: time (ms).

is comparable to that of a coarse lock since it does not need additional memory. However, fine-grained locking requires much more storage space since it adds S = sizeof(lock) * num_of_vertices, to the overall size of the structure.

Users may opt for HTM for suitable cases such to take most advantage of its performance, namely in cases where there are many conflicts. Moreover, it does not cause an increase in memory consumption.

D. Memory Allocation

Rather than using standard memory allocation functions for storing and updating large skewed graphs in-memory, which would be highly inefficient, CoreGraph optimizes this process with two main components : 1) an internal memory pool that pre-allocates the memory for loading the adjacency lists in a contiguous space and 2) a configurable allocation strategy using three state-of-the-art dynamic memory allocators, namely Glibc, Jemalloc [35] and TCMalloc [36]

First, we implement a smart loading procedure to optimize the physical memory layout of CoreGraph and therefore CoreGraph's analytic performance. Here, we describe the loading protocol using the memory manager in CoreGraph.

At the start of a graph loading process, memory of size S = sizeof(edge) * number_of_edges, is allocated. Then, we utilize that area to store the entire graph's edges in a continuous space within the same primary block. While we still keep track of pointers to edge lists for each vertex, the smart loader ensures that the edges are first stored in contiguous space. The memory compression made possible by this optimization brings us very near within 10% to the CSR in terms of read performance.

TABLE II. CONFIGURATIONS PROVIDED IN COREGRAPH

Configuration	Implementation	Impact
Reallocation Factor	Resizing the edge array when inserting new edges based on factor in config.	Low memory footprint, less memory allocations
Enabling Vertex/Edge Properties	Enabling storage for extra pointers for vertex/edge properties	Lower memory footprint, high cache performance
Synchronization Mechanism	Spinlock, Ticket/Profiler Lock, Intel HTM	Less contentions, high parallel performance
Memory Allocation	Glibc, Jemalloc, TCMalloc	Less fragmentation, high parallel performance

Secondly, CoreGraph employs efficient dynamic memory allocators, which allow to reduce memory fragmentation that occurs when memory is constantly being reallocated and so has chunks of unused space, or fragments, left over. In fact, in addition to the standard Glibc memory allocator, users can configure CoreGraph to use two state-of-the-art libraries for dynamic memory allocation: Jemalloc and TcMalloc. To demonstrate the performance gain from using these libraries, we performed a sensitivity analysis on CoreGraph with the three libraries for memory allocation [12] by running the same workload on three different allocator configurations of CoreGraph. As expected, Jemalloc and TCMalloc allow for a better performance than Glibc's performance for edge insertions. This is due to the parallel and thread-safe malloc implementation in both Jemalloc and TCMalloc, which gives better scalability when using multiple threads as shown in the plots.

IV. EVALUATION

We evaluated our framework on popular reference benchmarks [12] using readily available real-world and synthetic data sets that are popular in the graph research community, using two internal and external benchmark suites, Green-Marl [32] and GFE [37]. We evaluate the performance using real-world graph data sets [38], namely Twitter (1.4 billion edges), LiveJournal (68 million edges), and synthetic dataset [39] such as Uniform-24 (260 million edges), and Graph500-22 (69 million edges). We ran the experiments on a two-socket, 36-core machine with 384 GB of RAM. Its two 2.30 Ghz Intel Xeon E5-2699 v3 CPUs have 18 cores (36 hardware threads) and 32 KB, 256 KB and 46 MB L1, L2, and LLC caches, respectively.

CoreGraph outperforms the state of the art systems, reaching an update throughput of up to 24 Meps (million edges per second) compared to STINGER [18] (10 Meps), Teseo [25] (2.5 Meps) and GraphOne [23] (2 Meps). It also performs within 10% of CSR in graph analytics, while having a moderate memory overhead of 33% compared with CSR.

We highlighted the performance gains from using our framework in terms of high read performance, high update throughput while still maintaining a low memory footprint.

We demonstrated that CoreGraph offers high degree of configurability and we summarize these findings in Table II.

V. COREGRAPH FOR TRAFFIC DATA MANAGEMENT

In this section, we present our case study. We deemed important to test our solution on a more domain-specific dataset that represents real-world use case and can be exploited to extract insights for researchers and industry practitioners. The use case focuses on the traffic data management, which is critical for improving urban and highway traffic conditions to optimize traffic flow, detect congestion, and enhance overall transportation efficiency.

In this context, we extended the framework DIKCC, developed by our research team [40] [41] for traffic data management. The framework extracts knowledge from heterogeneous data sources and construct a knowledge graph characterized by high volume (300M edges and 4M edges) and high frequency of updates (10M updates/s).

Building on this foundational work, we integrate our framework CoreGraph into DIKCC framework, to provide high performance storage necessary in a context of high volume data and constantly evolving graphs. Our framework ensures efficient updates which are crucial in environements with real-time data stream. It also enables advanced analytic capabilities to address complex issues in traffic management. Moreover, CoreGraph outperforms graph databases or other state-of-the-art systems, that offer resource-consuming features that are not relevant to the traffic data discovery use case studied by our research team.

We utilized the same dataset as in the previous study [40], which consists of continuous views of three tables: Event, Route, and Intersection. The result of the knowledge construction from this data through is a property graph where the nodes are the intersections and the edges are the roads that link these intersections Fig. 7 shows the resulting graph.

We use the created mapping from relational-to-graph model, and we pre-process the datasets by converting them from CSV format as generated by the SQL queries, into file format that is supported by our framework, i.e, adjacency files, which does not require any additional schema definition and is consumed out of the box. Using this graph, our framework allows to recommend the optimal route to users to reach their destination by executing path traversal algorithms, which provide information about the state of the roads.

The updates to the graph are triggered from changes in the relational data by the creation of a new congestion event, that we process by updating the property value of edge of the corresponding road. Our framework supports topological mutation in addition to the updates to property values. Therefore, we extend the update workloads to the ones triggered by the creation of new roads between intersections, which we process as the insertion of new nodes and new edges. As for the new incoming updates, CoreGraph consumes continuous views to get the updates from the database as logs of edge lists, and performs the updates on the congestion status of roads (i.e. the edge property values) using our parallel batch updates API, and using our single update API for the insertions of new intersections and new roads (i.e. topological updates) since they are infrequent.

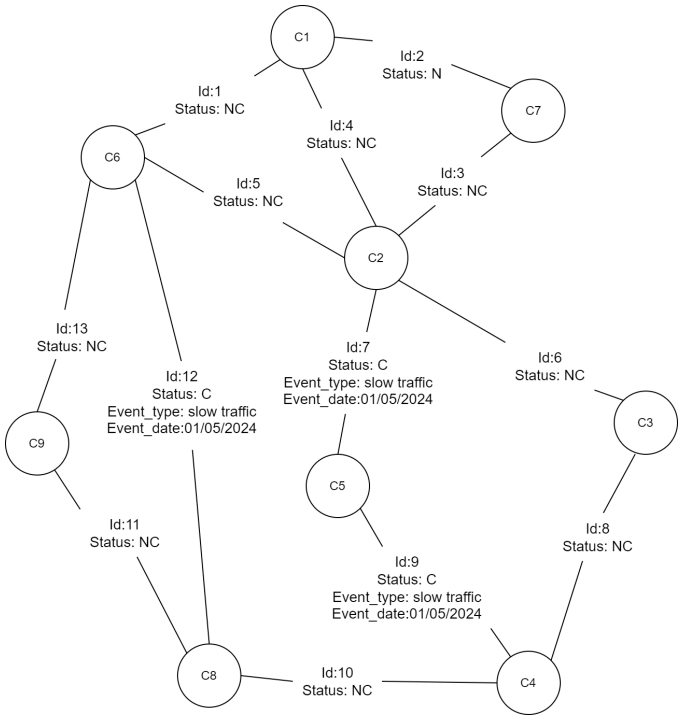

Fig. 7. Knowledge Graph for Traffic [40]. CoreGraph finds the shorted not congested routes by running BFS and setting the property values of edges accordingly.

We write a script to load the graph and measure the memory consumption. We find that the in-memory size is about 12 GB, which in principal can be supported on the same machine where the relational database resides.

After loading the graph, we run the path traversal algorithm namely BFS to find the shortest non-congested route from source to destination as depicted in Fig. 7. We then compare the output results to a validation data set for the top 10 nodes. After validating the correctness of our results, we measure the performance of the analytic workload which takes 2s to traverse the graph 300M of edges, which introduces minimal overhead to the end-to-end value chain process.

As for updates, since we store the adjacency of vertices in sorted order by id of intersections, we are able to locate the edge defined for the road by doing a binary search over the neighbour list using the identifier of the destination node which is the intersection. We use the index in that list to update the property value and set the congested state of the road. Moreover, for our extension to support high throughput topological updates, we use the prediction ML model from DIKCC to set the reallocation factor of the edge arrays, by predicting how much space we need to allocate for newly added intersections giving us an average throughput of 24M updates/sec.

We showed that our framework can be used as a vital component of the data discovery chain by demonstrating how our lightweight framework is easily integrated into a pre-existing framework with low overhead for traffic data management. We concluded that the performance gains from our integration are significantly higher than integration with a graph database system especially, and our framework can be easily compiled as a C++ library and shipped with the end-to-end framework without requiring extra configuration.

VI. Conclusion

In conclusion, this paper introduced CoreGraph, a highly configurable graph framework designed to address the challenges of efficient graph storage, mutation, and analysis. CoreGraph's storage and update protocol designs offer superior performance in terms of memory efficiency and update throughput compared to traditional graph data structures while maintaining read performance within 10% of CSR. The framework's integration of advanced synchronization mechanisms and configurable memory allocators enhances its adaptability to diverse workloads, making it suitable for real-time applications such as traffic data management. Our case study demonstrated that CoreGraph not only improves update throughput and analytic performance but also seamlessly integrates with existing frameworks, highlighting its potential as a critical component in modern data discovery and management systems. This study lays the groundwork for future research and development in optimizing graph processing systems, emphasizing the importance of balancing performance, memory consumption, and update efficiency in real-world scenarios.

References

[1] C. Peng, F. Xia, M. Naseriparsa, and F. Osborne, "Knowledge graphs: Opportunities and challenges," *Artificial Intelligence Review*, vol. 56, no. 11, pp. 13 071–13 102, 2023.

[2] C. Gao, Y. Zheng, N. Li, Y. Li, Y. Qin, J. Piao, Y. Quan, J. Chang, D. Jin, X. He *et al.*, "A survey of graph neural networks for recommender systems: Challenges, methods, and directions," *ACM Transactions on Recommender Systems*, vol. 1, no. 1, pp. 1–51, 2023.

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

[4] 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.

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

[6] 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.

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

[8] 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.

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

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

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

[12] S. Firmli and D. Chiadmi, "A scalable data structure for efficient graph analytics and in-place mutations," *Data*, vol. 8, no. 11, p. 166, 2023.

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

[14] 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.

[15] 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.

[16] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[17] 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.

[18] 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.

[19] 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.

[20] 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

[21] 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.

[22] 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.

[23] 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

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

[25] 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

[26] 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.

[27] 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.

[28] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious B-trees," *SIAM Journal on Computing*, vol. 35, no. 2, pp. 341–358, 2005. [Online]. Available: https://doi.org/10.1137/S0097539701389956

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

[30] M. Paradies, W. Lehner, and C. Bornhövd, "GRAPHITE: an extensible graph traversal framework for relational database management systems," in *SSDBM*, 2015.

[31] D. Durner, V. Leis, and T. Neumann, "On the impact of memory allocation on high-performance query processing," in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, ser. DaMoN'19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3329785.3329918

[32] "Green-Marl code," https://github.com/stanford-ppl/Green-Marl.

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

[34] J. Antić, G. Chatzopoulos, R. Guerraoui, and V. Trigonakis, "Locking made easy," in *Proceedings of the 17th International Middleware Conference*, 2016, pp. 1–14.

[35] J. Evans, "A scalable concurrent malloc(3) implementation for FreeBSD," in *BSDCan*, 2006.

[36] 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.

[37] "GFE driver code," https://github.com/cwida/gfe_driver.

[38] "SNAP (2014). Stanford Network Analysis Platform," http://snap.stanford.edu/snap.

[39] M. Capotă, T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, and P. Boncz, "Graphalytics: A big data benchmark for graph-processing platforms," in *Proceedings of the GRADES'15*, 2015, pp. 1–6.

[40] S. Yousfi, D. Chiadmi, and M. Rhanoui, "Smart big data framework for insight discovery," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 10, pp. 9777–9792, 2022.

[41] S. Yousfi, M. Rhanoui, and D. Chiadmi, "Towards a generic multimodal architecture for batch and streaming big data integration," *arXiv preprint arXiv:2108.04343*, 2021.