Enabling Efficient Direct Update on Rule-Based Compressed Graph

LIN FENG, Renmin University of China, China
FENG ZHANG, Renmin University of China, China
ZHENG CHEN, Tsinghua University, China
YUXIN TANG, Renmin University of China, China
JIAWEI GUAN, Renmin University of China, China
XIAOWEI ZHU, Tsinghua University, China
XIAOYONG DU, Renmin University of China, China

Dynamic graphs, pivotal in applications ranging from social networks to biological systems, pose significant challenges in storage and update efficiency due to their continuous evolution through insertions and deletions. Traditional graph compression methods, primarily optimized for static graphs, often struggle in dynamic environments, leading to costly decompression-recompression cycles during updates. To address this limitation, we propose a novel theoretical framework that enables efficient direct updates on rule-based compressed graphs. Additionally, we introduce a dynamic graph processing framework that balances space efficiency, update responsiveness, and query performance. Our framework treats updates as lightweight, localized modifications, sustains compression via background cleanup, and preserves query performance by ensuring structural integrity. Our method achieves significant memory savings, reducing usage by an average of 50.1% over state-of-the-art dynamic compressed graph systems, while demonstrating highly competitive update throughput and query performance. These advancements establish a scalable, high-performance solution for dynamic graph management, effectively bridging the gap between space efficiency, updates responsiveness and query performance.

CCS Concepts: • **Information systems** → **Graph-based database models**; • **Theory of computation** → **Data compression**.

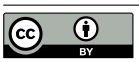
Additional Key Words and Phrases: graph compression; dynamic graph; direct update; rule-based compression

ACM Reference Format:
Lin Feng, Feng Zhang, Zheng Chen, Yuxin Tang, Jiawei Guan, Xiaowei Zhu, and Xiaoyong Du. 2026. Enabling Efficient Direct Update on Rule-Based Compressed Graph. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 32 (February 2026), 27 pages. https://doi.org/10.1145/3786646

1 Introduction

Graphs, fundamental to understanding complex structures in fields like social networks, biological systems, and transportation networks [3, 6, 15, 16, 28, 29], are growing ever larger and more intricate. This growth is especially pronounced in *dynamic graphs*, which evolve continuously through updates such as edge insertions and deletions. A prime example is the Facebook social

Authors' Contact Information: Lin Feng, fenglin0617@ruc.edu.cn, Renmin University of China, Beijing, China; Feng Zhang, fengzhang@ruc.edu.cn, Renmin University of China, Beijing, China; Zheng Chen, zheng-chen@mail.tsinghua.edu.cn, Tsinghua University, Beijing, China; Yuxin Tang, yuxintang@ruc.edu.cn, Renmin University of China, Beijing, China; Jiawei Guan, guanjw@ruc.edu.cn, Renmin University of China, Beijing, China; Xiaowei Zhu, coolerzxw@gmail.com, Tsinghua University, Beijing, China; Xiaoyong Du, duyong@ruc.edu.cn, Renmin University of China, Beijing, China.



ACM 2836-6573/2026/2-ART32
https://doi.org/10.1145/3786646

network. It expanded dramatically from 372 million users in 2011 to encompass over 2.1 billion daily active users and trillions of edges by the fourth quarter of 2023 [52]. Dynamic graphs are indispensable for real-world applications requiring constant updates [17, 18, 39–42, 57, 59, 64–66]. However, the ever-increasing size of graphs places significant strain on current graph management systems.

Managing ever-growing graphs necessitates techniques that are efficient in both storage and computation. Graph compression [1, 7–9, 11, 13, 19, 20, 26, 30, 32, 37, 54] addresses the storage challenge by exploiting structural redundancies. This same principle of redundancy exploitation also provides a significant computational advantage, as it allows graph algorithms to avoid redundant work during analysis. Consequently, performing queries and analytics directly on compressed graphs [10, 19, 32, 34, 38, 49, 69] has become a popular paradigm for static graphs, offering the compelling dual benefit of reduced memory footprint and faster processing times. However, the vast majority of these methods are designed for static graphs and lack support for dynamic updates. In real-world scenarios with frequently evolving graphs, this limitation forces costly cycles of full decompression and recompression for each change, negating the performance benefits. Therefore, there is an urgent need for technologies that support direct dynamic updates on compressed graphs.

Efficiently supporting dynamic updates directly on compressed graphs offers several advantages. First, it significantly enhances update responsiveness. By obviating the need for the prohibitively costly decompress-recompress cycle inherent in static methods, this approach enables high-throughput modifications directly on the compressed data structure. Second, it continuously leverages the intrinsic benefits of compression. This includes high space efficiency, which minimizes the storage footprint, and the potential for accelerated query and analysis performance. Third, it unlocks the scalability of large, dynamically evolving graphs. The efficiency gained from direct updates extends the applicability of compression frameworks to manage massive, continuously changing graphs, ensuring scalability and performance in real-time, resource-constrained scenarios where previous methods were untenable.

We choose rule-based compression as our foundational strategy, as it provides a unique combination of structural compatibility, dynamic adaptability, and performance benefits that are essential for our goals. First, it offers high structural compatibility with mainstream graph processing. Unlike compression methods such as k^2-tree [11, 21] that operate on the adjacency matrix, rule-based compression is naturally suited to adjacency lists and CSR, the standard formats for most large-scale dynamic graph systems [27]. Crucially, the resulting compressed structure remains conceptually equivalent to a standard graph, allowing for the direct application of conventional graph analysis algorithms. Second, it possesses inherent dynamic adaptability. Rule-based compression is amenable to localized updates. As we observe, changes to vertices and edges typically affect only their parent rule(s). This property enables efficient, in-place modifications, which is a distinct advantage over static methods like LZ-family or WebGraph [10]. Those methods lack mechanisms for granular updates and thus require prohibitively costly decompress-recompress cycles to handle any change. Third, it provides a strong performance foundation for balancing space and speed. By identifying and reducing recurring topological patterns, this strategy creates a significant computational advantage, as it allows analytical queries to avoid redundant work and process faster directly on the compressed data [19, 56]. Furthermore, the compression itself offers highly efficient overhead and achieves high compression ratios, competitive with mainstream static methods (as our evaluation demonstrates in Table 4 and Fig. 9), and scales effectively to large graphs [58].

Despite these benefits, enabling dynamic updates on rule-based compressed graphs remains fundamentally challenging. This difficulty primarily stems from the inherent contradiction between dynamic updates and graph compression: updates disrupt the stable, redundant structures essential for compression, while compressed formats lack the direct-access capabilities required for efficient

modifications. Furthermore, on rule-based compression, this core conflict raises several critical challenges. First, performing targeted updates requires efficient random access, but traditional rule-based compressed graphs lack native index support, making it non-trivial to map a specific edge to its physical location. Second, new insertions must be integrated without degrading overall compression efficiency, requiring an efficient mechanism to periodically compress and merge this new data. Third, deletions are particularly complex, as modifying a shared rule can inadvertently alter other parts of the graph, and the process inherently introduces data redundancy that undermines the graph's compression.

To address these challenges, our design philosophy is that an ideal approach must strike a delicate balance among the three competing factors: *space efficiency*, *update responsiveness*, and *query performance*. However, existing systems typically falter by making significant compromises. On one hand, traditional non-compressed dynamic graph systems like Teseo [22] offer excellent update speed but sacrifice space efficiency, making them untenable for increasingly large graphs. On the other hand, compressed dynamic graph systems attempt to solve the space issue but introduce other bottlenecks. For instance, ZipG [34] sacrifices query performance on certain complex types due to the structural fragmentation caused by its update mechanism. Conversely, Aspen [23] compromises update responsiveness, as its reliance on localized decompression-recompression for each modification introduces significant latency. Evidently, current solutions force a suboptimal trade-off, highlighting the critical need for a new approach that can simultaneously achieve all three objectives.

In this paper, we propose a rule-based framework designed to strike a delicate balance among the three competing factors: space efficiency, update responsiveness, and query performance. Our approach is grounded in a novel theoretical framework that establishes principles for effective direct update designs and a practical direct update framework that separates update and compression modules to preserve integrity and throughput. The core of our solution consists of several novel designs: (1) An efficient edge retrieval and indexing mechanism using a hierarchical path of pointers to uniquely locate any edge and a tailored index with tunable granularity. (2) A two-stage insertion method that buffers new insertions for high responsiveness and periodically compresses and merges them to maintain long-term efficiency. (3) A deletion method employing a recursive copy-on-write mechanism on local rules, which is paired with a two-phase maintenance strategy (rule-folding and secondary compression) to counteract compression degradation.

Experimental results validate that our framework delivers a superior performance profile compared to state-of-the-art dynamic graph systems. It reduces memory usage by an average of 50.1% compared to best dynamic compressed graph systems [23, 47], and 47.1% compared to best uncompressed dynamic graph management systems [27, 51], while demonstrating competitive update throughput. Crucially, these advantages are achieved while delivering query performance competitive with the leading solutions, thus breaking conventional performance trade-offs.

The main contributions of this paper are as follows:

- We propose a novel theoretical framework for efficient direct updates on rule-based compressed graphs, enabling localized updates that minimize overhead while maintaining the integrity of the compressed structure.
- We introduce the first dynamic compressed graph processing framework that integrates an indexing system with a tailored update mechanism, achieving a seamless balance among space efficiency, update responsiveness, and query performance.
- We evaluate our framework against the state-of-the-art, achieving high update throughput while preserving low memory usage and excellent query performance.

2 Background and Related Work

2.1 Graph Preliminaries and Compression

Graphs are fundamental data structures consisting of vertices and edges that connect these vertices. Formally, a graph G can be defined as a pair $G = (V, E)$, where V represents the set of vertices and E represents the set of edges, with each edge $(u, v) \in E$ connecting two vertices u and v. One common representation of graphs is the *adjacency list*, where each vertex stores a list of its neighbors. This representation is space-efficient for sparse graphs and enables efficient traversal, making it widely used in real-world applications [44, 68].

Graph Compression. Graph compression techniques aim to create a compact representation of a graph to reduce its storage footprint while preserving essential structural properties. Graph compression can be broadly categorized into two techniques: *encoding* and *reduction*. The primary distinction between the two lies in whether the graph's topology is modified. Encoding methods [1, 8, 9, 26, 54] preserve the original graph structure intact, while reduction methods [11, 13, 20, 30, 32, 37] transform the graph's topology. In practice, the choice of method depends on the specific application scenario. Encoding methods are suited for random access and simple query, as they compress the bit-level representation of vertices and edges while preserving the original graph structure intact. However, since these techniques achieve space savings through entropy-based encoding, such as Huffman or delta coding, their compression ratios are inherently limited by the entropy of the graph data [36]. Reduction methods [11, 13, 20, 30, 32, 37], in contrast, are well-suited for analyzing large-scale graphs with high structural redundancy. These methods identify recurring patterns or substructures and replace them with compact symbolic representations, thereby transforming the graph's topology and achieving significantly higher compression ratios. In our cases, we adopt rule-based compression—a type of reduction method (whose static compression performance has been recently optimized by works like [58])—due to its advantages in compression ratios and graph analysis.

Dynamic Graph. Extensive work on dynamic graphs can be broadly divided into two categories: optimizing the computational aspect of streaming graph processing, and designing efficient storage and management structures [12, 14, 21, 22, 27, 33, 51, 68]. The first category focuses on reducing redundant computations during analysis. For instance, ACGraph [31] accelerates monotonic graph algorithms by using a dependence hierarchy to normalize state propagation, which is synergistic with our work. Our research belongs to the second category, which focuses on the fundamental design of dynamic graph storage and management. This line of research primarily addresses the trade-offs among space usage, update latency, and query performance. Existing systems often compromise on one of these aspects. For instance, Teseo [22] employs sparse arrays and fat trees to provide high update throughput on uncompressed data, but sacrifices space efficiency due to these uncompressed structures and its index design. Spruce [51] also targets high update throughput, using a tree-like multilevel structure, but compromises query performance. In contrast, Sortledton [27] achieves excellent space and query efficiency by storing neighborhoods in a sorted set data structure, though its update throughput is lower than systems like Spruce. Recent benchmark work [53] has revisited the design of in-memory dynamic graph storage, identifying substantial space overhead in these systems, which calls for new space-saving solutions and highlights the importance of dynamic compressed graphs.

In parallel, several methods have been proposed to support updates on compressed graphs [2, 23, 47]. However, these methods often still require decompression and recompression steps—even if only localized—when handling updates. For instance, Aspen [23] must perform localized decompression and recompression within the chunks of its C-tree. Moreover, some compressed approaches introduce significant overheads that make them uncompetitive. For example, GraphZIP [47] employs

a clique mining algorithm for compression (with $O(3^{n/3})$ time complexity), making it intractable for large graphs. This complex structure, in turn, results in prohibitively low update throughput compared to uncompressed systems.

2.2 Rule-Based Reduction

Among reduction techniques, *rule-based reduction* has gained attention for its ability to systematically simplify graph structures while preserving essential properties [19, 20, 37, 43, 45, 46, 60–63]. Rule-based reduction identifies recurring patterns in a graph's adjacency lists and replaces them with compact symbolic representations, called *rules*. These rules are organized hierarchically, with higher-level rules encapsulating lower-level patterns. The top-level rule represents the entire graph. The time and space complexity of rule-based reduction are both $O(n)$, where n is the length of sequence to be reduced.

Workflow. Fig. 1 illustrates the general workflow of rule-based reduction. The compressor scans the input sequence repeated pairs of consecutive elements, known as *digrams*. These digrams are stored in a hash table for efficient tracking. When a digram appears multiple times, the compressor replaces it with a new rule. For example, in Fig. 1, the recurring digram $(3, 2)$ is replaced with rule r_1, and r_1 is added to the rule set. The sequence is updated with the new rule, and the process continues until no digram appears more than once.

The resulting rule set satisfies two key properties [45]:

- *Uniqueness*: No pair of adjacent symbols appears more than once in the rule set.
- *Reusability*: Each rule is used at least twice in the compressed representation.

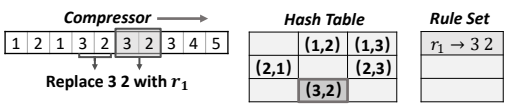


Fig. 1. The workflow of rule-based compression.

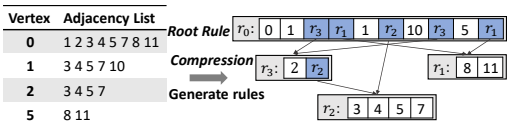


Fig. 2. An example of rule-based compressed graph.

Example. Fig. 2 demonstrates an example of applying rule-based reduction to compress a graph's adjacency list. The adjacency lists of all vertices are concatenated into a single integer sequence, with a special delimiter symbol separating the list of each vertex. Repeated patterns, such as $(8, 11)$ and $(3, 4, 5, 7)$, are identified and replaced by rules r_1 and r_2, respectively. Higher-level rules are recursively generated, such as r_3, which encapsulates $(2, r_2)$. Ultimately, the graph is represented by a root rule r_0, which encodes the entire structure in a hierarchical format. This process results in a Directed Acyclic Graph (DAG) of rules, where each rule represents a compressed substructure of the graph, significantly reducing redundancy while preserving the graph's essential properties.

3 Motivation

Rule-based compression methods achieve high compression ratios while preserving essential graph properties, enabling high-performance analysis with significant space savings on static graphs [19]. However, these techniques are inherently static and ill-suited for evolving graph datasets that require dynamic updates. A naive approach of decompressing the graph for batch updates and then recompressing it introduces prohibitive overhead. First, it introduces time overhead caused by decompression and recompression. As shown in Fig. 3 (a), in the *pkustk14* dataset of SuiteSparse Matrix Collection [55], the execution time for 100,000 batch edge insertions using this method is up to 10× slower than a native dynamic graph analysis framework such as Sortledton [27]. A time breakdown confirms this overhead is dominated by the decompression and recompression steps. Second, it imposes significant memory pressure. Fig. 3 (b) highlights a severe memory issue:

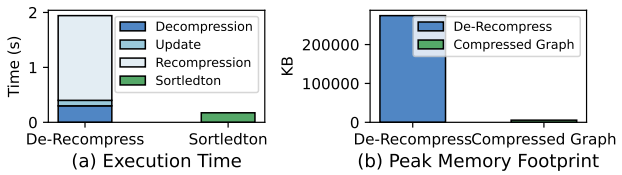


Fig. 3. Performance comparison of the decompress-recompress update strategy for 100,000 insertions on *pkustk14*.

the peak memory footprint is orders of magnitude larger than the final compressed graph's size, negating the core benefit of compression. Therefore, an approach that enables direct updates on the compressed graph is essential to avoid the costly cycle of full decompression and recompression.

Basic Idea. Our direct update method is designed to perform efficient updates on rule-based compressed graphs while preserving both the compression level and query performance. We propose a set of theoretical axioms—essential guidelines that all direct updates must obey—to ensure effectiveness and consistency. To achieve high update throughput, we utilize a design that separates the update and compression modules, allowing them to operate independently while preserving the global integrity of the compressed graph.

Novelties. To address the challenges mentioned in Section 1, we propose the following novel designs:

- **Framework for direct update on rule-based compressed graph (Section 4 and Section 5.1).** We propose the first theoretical framework for performing direct updates on compressed graphs, establishing the principles for effective direct update designs. Guided by this theory, we introduce a practical direct update framework that simultaneously achieves three objectives: it preserves space efficiency by periodically compressing modified parts, enables fast updates by minimizing structural modifications, and guarantees query performance by maintaining the integrity of the compressed structure.
- **Efficient approach for edge retrieval and indexing (Section 5.2).** We design a location description mechanism that uses a hierarchical path of pointers from the root to the target rule, which can uniquely locate any edge. We also develop a tailored index for rule-based compressed graphs, featuring tunable granularity to allow users to balance space overhead and lookup performance according to their specific requirements.
- **Insertion method balancing responsiveness and compression (Section 5.3).** We propose a direct insertion method that operates in two stages. Initially, to ensure high responsiveness, new insertions are buffered in an uncompressed, incremental subgraph. Subsequently, these updates are periodically compressed and merged to maintain long-term compression efficiency.
- **Deletion method with compression efficiency guaranteed (Section 5.4).** We propose a direct deletion method that employs a recursive copy-on-write mechanism on local rules. To counteract the impact on compression, it then applies a two-phase maintenance strategy: rule-folding to eliminate inefficient rules, followed by secondary compression to identify new common patterns.

4 Theoretical Framework

A rule-based compressed graph G' can be denoted as a pair $G' = (\Sigma, R)$, where $\Sigma = \{\sigma_1, \sigma_2, ...\sigma_n\}$ is a set of symbols, and each symbol is either a vertex or a rule. $R = \{r_1, r_2, ...r_n\}$ refers to a rule set, where each rule $r = (\sigma_1, \sigma_2, ...\sigma_n)$, $\sigma_k \in \Sigma, k \in \{1, 2, ..., n\}$ is an ordered sequence of symbols. Direct update on rule-based compressed graph aims to achieve equivalent operation outputs on

both compressed and uncompressed graphs, encompassing direct insertion, deletion and in-situ update operations. Graph update can be either edge update or vertex update, where vertex update can be reduced to edge update. For example, deleting a vertex can be accomplished by deleting all its associated edges. [1] Consequently, this paper focuses on the direct update of edges. For simplicity, we use the term *direct update* to refer to direct update of edges.

Definition 1 (**Direct Update**). *Given a compression mapping $\mathcal{C} : G(V, E) \to G'(\Sigma, R)$ and decompression mapping $\mathcal{D} : G'(\Sigma, R) \to G(V, E)$, where $G(V, E)$ represents an uncompressed graph and $G' = (\Sigma, R)$ denotes the corresponding compressed graph. We define the operations $\oplus'_G : G' \times E' \to G'$ and $\ominus'_G : G' \times E' \to G'$ as the operation of inserting and deleting the edge set S on the compressed graph G', respectively. Here, $E' \subset \mathbb{Z}^+ \times \mathbb{Z}^+$ ($\mathbb{Z}^+$ represents the set of positive integers) is a set of edges, each represented by pairs of positive integer vertex IDs. If the following conditions hold:*

$$\mathcal{D}(G' \oplus'_G S) = G(V, E \cup S), \quad \mathcal{D}(G' \ominus'_G S) = G(V, E \setminus S)$$

then the operations $\oplus'_G$ and $\ominus'_G$ are referred to as direct insertion and direct deletion on compressed graph G', respectively.

With the definition of direct insertion and deletion, a more complex direct modification operation, which replaces edge set E_{old} with new edge set E_{new}, can be modeled as a composition of these two fundamental operations: $G' \ominus'_G E_{old} \oplus'_G E_{new}$. Direct update enable operations to be performed directly on the compressed graph, avoiding the computational overhead of global decompression and recompression processes.

The practical viability of this direct update paradigm hinges on a set of foundational principles. To formalize these principles, we establish the following three axioms. First, for direct updates to be computationally meaningful, their performance must be asymptotically superior to the naive decompress-recompress cycle. This is only achievable if the modifications to the compressed structure are contained and proportional to the scale of the update, not the entire graph. Axiom 1 establishes this fundamental prerequisite for efficiency.

Axiom 1 (**Locality**). *Given a rule-based compressed graph $G' = (\Sigma, R)$ with direct insertion $\oplus'_G$ and direct deletion $\ominus'_G$, $\oplus'_G$ and $\ominus'_G$ should modify only a subset of Σ and R.*

Second, to maintain compression efficiency, the size of the compressed graph after a direct update must be smaller than that of its uncompressed counterpart after the same update. This principle is formalized in Axiom 2.

Axiom 2 (**Compression Efficiency**). *Let $G' = (\Sigma, R)$ be a rule-based compressed graph and let $G = (V, E)$ be its corresponding uncompressed graph. For any given edge set $E' \subset \mathbb{Z}^+ \times \mathbb{Z}^+$, the direct insertion $\oplus'_G$ and direct deletion $\ominus'_G$ operations must satisfy:*

$$|G' \oplus'_G E'| < |G(V, E \cup E')|, \quad |G' \ominus'_G E'| < |G(V, E \setminus E')|$$

Beyond efficiency, a direct update must also preserve the semantic and operational integrity of the compressed representation. An update should not render the resulting graph incompatible with existing query mechanisms, as this would defeat the purpose of a unified data structure. Axiom 3 provides this crucial guarantee, ensuring consistent interoperability between query and update operations.

Axiom 3 (**Closure under Updates**). *Let $\mathbb{Q}$ be a class of query algorithms. Let $\mathcal{G}'_{\mathbb{Q}}$ be the set of all compressed graphs G' for which every algorithm $q \in \mathbb{Q}$ is well-defined. Then, the set $\mathcal{G}'_{\mathbb{Q}}$ is closed under the direct update operations $\oplus'_G$ and $\ominus'_G$.*

[1] The case of isolated vertices is not considered in this paper. Vertices without edges are directly removed from the graph.

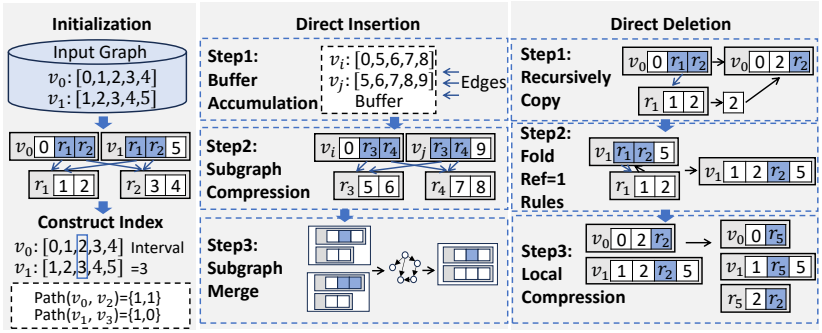


Fig. 4. An overview of the rule-based compressed graph framework.

Axioms 1, 2, and 3 collectively establish the viability of our direct update paradigm by defining three fundamental requirements. Specifically, Axiom 1 ensures the computational efficiency of updates; Axiom 2 maintains space efficiency by keeping the graph compact; and Axiom 3 guarantees that the graph remains structurally sound and amenable to subsequent queries.

5 Direct Update on Rule-Based Compression

5.1 Overview

Guided by the theoretical framework established, we propose a direct update framework that consists of three integral modules: **Initialization**, **Direct Insertion**, and **Direct Deletion**, depicted in Fig. 4. These modules construct a rule-based compressed graph with its index, and dynamically maintain it by *direct update* (Definition 1) operations. A core design principle of our framework is the strict separation of the live update from background cleanup optimization. This separation is what enables us to support high-throughput, live updates (e.g., buffered insertions and copy-on-write deletions) without requiring any decompression-recompression on the critical path. The decompress-recompress strategy is employed only as a non-blocking, asynchronous background process (Section 5.3) to merge subgraphs. As justified in Theorem 3, this is an explicit trade-off to prevent long-term compression degradation while prioritizing foreground responsiveness.

Our framework is designed to satisfy three key axioms. First, it satisfies Locality (Axiom 1). It is crucial to note that Axiom 1 is not a *precondition* that must be checked for an update; rather, it is an *inherent property* that our update mechanisms are designed to satisfy. Our append-only insertion strategy (Section 5.3) is local by design, and our copy-on-write deletion mechanism (Section 5.4) confines modifications to a specific rule path. In the theoretical edge case where a deletion targets the final instance of a rule chain (i.e., a part of the graph that is already uncompressed), the overhead gracefully degrades to be equivalent to modifying an uncompressed structure, as no shared rules are affected. Second, it upholds Compression (Axiom 2) through a continuous background cleanup process. Finally, it ensures Closure (Axiom 3), guaranteeing the graph remains consistently queryable after any operation.

Workflow. The workflow is initiated in the **Initialization** module, which takes an input graph, compresses it via a rule-based technique, and then establishes *index list* for efficient navigation and retrieval. When insertions come, **Direct Insertion** is performed in three stages. First, insertions are accumulated in a buffer as an incremental graph. Once the buffer reaches a specified threshold, the collected edges are compressed into a subgraph. Finally, once a sufficient number of these subgraphs have accumulated, a background process merges them into the main graph structure. **Direct Deletion** is also a three-step process. To remove an edge, it first recursively copies the rules along the path from the root to the target rule and performs edge deletion on the copies.

Subsequently, a background cleanup process folds and removes any rules with a single reference into their parent. The final step involves applying local secondary compression to these rules modified by the folding process, ensuring they are optimally compressed.

Next, we will introduce the detailed implementation of our framework in the order of *Initialization*, *Direct Insertion*, and *Direct Deletion*.

5.2 Initialization

Design. To apply rule-based compression to the input graph, we first concatenate all adjacency lists into a single sequence, using separators to distinguish between them. After compression, Theorem 1 guarantees that the resulting compressed adjacency lists maintain their independence. Therefore, we can safely split them apart, which not only prevents the root rule from becoming excessively long, but also facilitates rapid access to each individual adjacency list.

Theorem 1 (**Adjacency List Independence Principle**). *The starting position of each adjacency list in the rule-based compressed graph is directly traceable from the root rule.*

Proof. In the original uncompressed graph, each adjacency list ends with a special delimiter symbol, denoted as *sep*, followed by the next source vertex v, forming the pair (sep, v). Since this pair appears only once in the graph and is structurally unique, it cannot be reduced into a non-terminal during rule-based compression. As a result, the pair (sep, v) must remain intact in the root rule after compression. Consequently, the adjacency list of v either (1) starts with v as a terminal symbol in the root rule, or (2) appears as the first symbol within a non-terminal in the root rule, where the non-terminal represents a larger compressed substructure. In both cases, the starting point of the adjacency list is directly traceable from the root rule. □

In order to enable accelerated edge lookup within each adjacency list, we construct indexes by recording (*neighbor vertex*, *access path*) pairs at regular intervals, named as *index list*. A *path* is defined as a sequence of offsets that navigates from the root rule to the rule containing the target edge. Each offset in this sequence acts as a coordinate, specifying both the position within a rule's body and the pointer to sub-rule referenced at that position. The *path* to an edge can uniquely locate it within the hierarchical structure of rules, as defined in Definition 2.

Definition 2 (**Path**). *Given a rule-based compressed graph $G'(\Sigma, R)$, a path to a given edge $e \in E(\mathcal{D}(G'))$ in G' is an ordered sequence $p(e) = \langle \delta_1, \delta_2, \ldots, \delta_n \mid \delta_i \in \{0, 1, \ldots, \ell(r_i) - 1\} \rangle$. Each element δ_i denotes the offset within rule r_i, which is the corresponding rule at the i-th layer, and $\ell(r_i)$ denotes the length of r_i.*

The *index list* is a nested dictionary that maps each vertex to its sampled indexes; each index entry then maps a neighbor to an access path, as demonstrated in Definition 3.

Definition 3 (**Index List**). *Given a vertex $v_i \in V$, the index list of v_i is a mapping $N(v_i) \rightarrow P$, where $N(v_i)$ is the set of neighbor vertices of v_i, and P denotes a set of paths. It maps the adjacent vertices within each entry at specific intervals to their corresponding paths from the root rule.*

When retrieving an edge in rule-based compressed graph, *index list* constrains the search space by leveraging the known intervals between indexed vertices, and avoids a linear scan of the entire list.

To optimize the trade-off between retrieval performance and maintenance overhead, the index structure is highly configurable: users can set a maximum path length to curb space from infrequent deep rules and adjust the sampling interval to balance retrieval speed against maintenance costs.

Example. An example of a *path* and an *index list* for a rule-based compressed graph is illustrated in Fig. 5. The *path* to edge (v_0, v_3) is $(2, 0)$, which indicates that the edge is located by first navigating

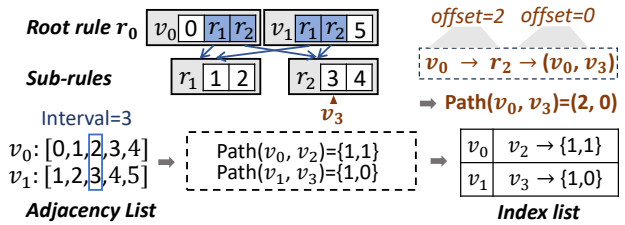


Fig. 5. An example of path and index list.

to *offset* = 2 in the root rule r_0 (which leads to sub-rule r_2), and then navigating to *offset* = 0 within r_2. The *index list* is constructed by sampling neighbors from the original uncompressed graph at an *interval* = 3 and recording the *path* to each sampled vertex.

Algorithm. Algorithm 1 details the process of index construction and path retrieval through index. Following initialization (Lines 1-2), the algorithm builds the *index list* (Lines 3-7): an index is created for a neighbor if its access path length is within the specified maximum depth and the sampling interval from the last indexed neighbor has been met. Finally, Lines 8-11 describe the edge retrieval procedure. This process first uses a binary search on the *index list* to find the nearest preceding indexed entry, and then performs a linear scan—the "last mile search"—from that entry to locate the target edge.

Algorithm 1 Index Construction and Path Retrieval

Require: Compressed graph G', interval between indexes I, max depth of index path D
Ensure: Build index for G' and retrieve path through index

```
1:  function BuildIndex(G', I, D)
2:      index_list ← ∅
3:      for each adjacency list A(v) in G' do
4:          for each symbol s ∈ G' do
5:              if PrevIndexInterval(s) ≥ I and Depth(s) ≤ D then
6:                  index_list[v].insert(s, Path(s))
7:      return index_list
8:  function RetrievePath(G', e)                    ▷ Find path to edge e
9:      I_lower ← BinarySearch(index_list(e.first), e.second)   ▷ Locate lower index for e
10:     (v_upper, Path) ← LastMileSearch(I_lower, e.second)
11:     return (v_upper, Path)
```

Complexity. The upper bound of the last-mile-search cost within such an interval is formally established in Theorem 2.

Theorem 2 (**Search Space Bound between Indexed Vertices**). *Consider a rule-based compressed graph $G'(\Sigma, R)$ and a vertex $v_i \in V$. Let v_{n1} and v_{n2} be two vertices in the index list of v_i. If the interval between v_{n1} and v_{n2} in the original, uncompressed adjacency list $A(v_i)$ is k, then for any vertex v located between them in $A(v_i)$, The number of elements traversed to access v in G' is bounded by $\leq \lceil \frac{k}{2^{d-1}} \rceil \cdot (2^d - 1)$. Here, d represents the maximum depth of rules between v_{n1} and v_{n2}.*

Proof. The process of locating v in G' is equivalent to a pre-order traversal of a traversal tree, which is formed by unrolling the DAG of rules. The traversal cost—the total number of visited nodes—is maximized when the ratio of internal nodes (rule pointers) to leaf nodes (stored vertices) is at its highest. A key property of our rule set is that the length of every rule is at least 2, implying each internal node in the traversal tree must have at least two children. Based on this, we model

the structure yielding the highest cost as a full binary tree of the maximum allowed depth d. This structure represents a maximal branching configuration, thereby maximizing the node visits required to access its leaves. A full binary tree of depth d has 2^{d-1} leaves and a total of $2^d - 1$ nodes. To bound the search cost, we partition the k elements into two parts: (1) a portion that forms $\lfloor \frac{k}{2^{d-1}} \rfloor$ complete full binary trees of depth d, and (2) the remaining elements. The total number of elements accessed is therefore bounded by $\lfloor \frac{k}{2^{d-1}} \rfloor \cdot (2^d - 1) + (2^d - 1) = \lceil \frac{k}{2^{d-1}} \rceil \cdot (2^d - 1)$. □

According to Theorem 2, the time complexity of edge retrieval by *index list* is $O(\max\{\lceil \frac{k}{2^{d-1}} \rceil \cdot (2^d - 1), \log(n)\})$, where n is the number of indexes in source vertex's *index list*. The space complexity of *index list* is $O(\frac{|E|d}{k})$, where $|E|$ is the edge count in the original uncompressed graph. Its tangible performance improvements are empirically validated in our experiments (Section 6).

5.3 Direct Insertion

Design. Based on our observation in Section 3, appending new edges does not alter the final results of graph analysis. To preserve the immutability of the original compressed graph, we manage new insertions in a separate incremental graph stored in buffer. During analysis, this incremental graph is logically combined with the original one. To accelerate edge lookups within this structure, we maintain its adjacency lists in a sorted order, which enables efficient retrieval via binary search.

To maintain overall system efficiency, the incremental graph is compressed once its size reaches a predefined threshold. Since our compression is most effective on dense graphs, we recommend performing this operation when the number of edges is substantially greater than the number of vertices—for instance, by two orders of magnitude.

As compressed incremental graphs accumulate over time, edge retrieval performance degrades because a single lookup may need to traverse multiple graph fragments. To mitigate this issue, these graphs must be periodically merged. An intuitive strategy for this consolidation would be to directly unify the two rule sets. This approach seems to align with our goal of avoiding full decompression. However, such direct rule-set merge can be very costly, as introduced in Theorem 3.

Theorem 3 (**Complexity of Rule Set Unification**). *Given two compressed graphs, $G'_1(\Sigma_1, R_1)$ and $G'_2(\Sigma_2, R_2)$, the worst-case time and space complexity for unifying the rule sets R_1 and R_2 by identifying and merging equivalent rules is $O(\sum_{r \in R_1, R_2} |\mathcal{D}(r)|)$. Here, $|\mathcal{D}(r)|$ is the length of the terminal string derived from a rule r.*

Proof. The worst-case complexity arises when determining the equivalence of two rules, $r_a \in R_1$ and $r_b \in R_2$, requires their full expansion. This typically occurs when their internal derivation trees are structurally different and share no common non-terminal symbols that could be used for an optimized comparison. Consider two equivalent rules, r_1 and r'_1, whose productions result in different derivation trees. For instance, let $r_1 \to ar_2r_3$ and $r'_1 \to r'_2r'_3e$, with further expansions $r_2 \to bc$, $r_3 \to de$, $r'_2 \to ab$, and $r'_3 \to cd$. Although both rules ultimately derive the identical terminal string $\mathcal{D}(r_1) = \mathcal{D}(r'_1)$ = "abcde", their component non-terminals (r_2, r_3, r'_2, r'_3) are all distinct. To prove that r_1 and r'_1 are equivalent in such a scenario, an algorithm has no choice but to fully expand them to their terminal representations. Extending this principle to the entire unification process, the overall complexity is dictated by the cumulative cost of all such necessary expansions. In the worst case, this may involve expanding every rule in both R_1 and R_2, with complexity bounded by $O(\sum_{r \in R_1, R_2} |\mathcal{D}(r)|)$.

Matching the resulting strings can be implemented efficiently by building a hash table over the expanded strings of all rules in R_1, and then probing it with the expanded strings of rules from R_2. The time for performing the hash-based matching is $O(\sum_{r \in R_1} |\mathcal{D}(r)|$ for building hash table, and

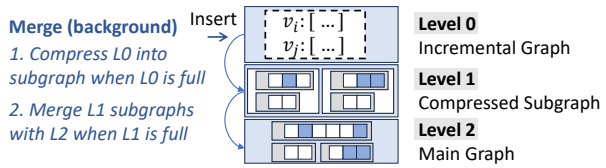


Fig. 6. The process of direct insertion.

$\sum_{r\in R_2} |\mathcal{D}(r)|$ for matching. The space for hash table is $O(\sum_{r\in R_1} |\mathcal{D}(r)|)$. Therefore, the total time and space complexity is $O(\sum_{r\in R_1,R_2} |\mathcal{D}(r)|)$. □

Theorem 3 indicates that a direct unification of rule sets can be exceptionally costly. Since the time and space complexity to decompress and recompress the two graphs are only $O(|\mathcal{D}(G_1')| + |\mathcal{D}(G_2')|) \ll O(\sum_{r\in R_1,R_2} |\mathcal{D}(r)|)$, any direct-merge algorithm is paradoxically outperformed by this simpler, brute-force method. Therefore, we choose to merge incremental graphs by first decompressing and then recompressing them.

To orchestrate this periodic merge and minimize the scope of consolidation, we employ a tiered architecture inspired by the design of Log-Structured Merge-Trees (LSM-Trees), as shown in Fig. 6. New edge insertions are buffered in a dynamic, in-memory graph (Level 0). When this graph reaches a size threshold, it is compressed into an immutable fragment at Level 1. Finally, when the number of Level 1 fragments becomes excessive, they are merged with the main graph in Level 2 using the aforementioned decompress-recompress strategy. This structured, multi-level approach systematically manages the lifecycle of graph fragments, ensuring that both high write throughput and efficient long-term read performance are maintained.

Algorithm. The direct insertion algorithm is shown in Algorithm 2. Upon receiving a set of new edges $E_{\text{ins}}(u)$, the algorithm first checks the existence of these edges in the original compressed graph by path retrieval (Line 2), and then commits them to the dynamic, in-memory buffer G_{buf} (Line 3), which represents Level 0. This operation is highly efficient, leveraging the sorted adjacency list structure discussed previously.

Following the insertion, the algorithm evaluates two cascading trigger conditions for compaction. First, if the size of G_{buf} exceeds its configured threshold, a minor compaction is initiated: the buffer is compressed into a new, immutable fragment and promoted to Level 1 (Lines 4–5). Subsequently, if this promotion causes the number of fragments in Level 1 to surpass its own limit, a major compaction is triggered. All Level 1 fragments and the original Level 2 graph are consolidated into a single, larger Level 2 graph using decompress-recompress strategy (Lines 6–7). These compaction routines are designed to run as background tasks to ensure that the primary insertion path remains non-blocking and consistently fast.

Algorithm 2 Direct Insertion Algorithm

Require: Compressed graph G', edges of vertex u to insert $E_{\text{ins}}(u) = \{(u, v_1), (u, v_2), \ldots, (u, v_n)\}$
Ensure: Insert new edges into graph G'

1: **function** DirectInsert(G', $E_{\text{ins}}(u)$)
2: Check the existence of $E_{\text{ins}}(u)$ in G'
3: Insert $E_{\text{ins}}(u)$ in the incremental graph G_{buf} in buffer
4: **if** Incremental graph size reaches threshold **then**
5: Compress G_{buf} and place in Level 1
6: **if** Level 1 size reaches threshold **then**
7: Compact all Level 1 fragments with Level 2 graph

Complexity. For the Level 0 uncompressed graph in buffer, we implement each adjacency list as a balanced binary search tree (e.g., a red-black tree). Consequently, the time complexity for inserting a new edge is $O(\log k)$, where k is the number of edges for the source vertex in the incremental graph. The space complexity for the new entry is $O(1)$.

The compaction process for Level 1, which consolidates multiple fragments, is linear in the total size of the input data. Its time and space complexity is $O(\sum_{i=1}^{n} |G_i|)$, where n is the number of graphs to be merged and $|G_i|$ is the size of the i-th graph after decompression.

5.4 Direct Deletion

Design. Unlike insertion, direct deletion necessitates modifications to the compressed graph structure. Guided by Axiom 1, these modifications affect only a part of the compressed graph. As established in Theorem 4, such modifications are systematically managed by recursively copying rules along the path from the root to the target edge.

Theorem 4 (**Locality of Rule-Based Update**). *Let $G'(\Sigma, R)$ be a rule-based compressed graph, and let $e \in E(\mathcal{D}(G'))$ be an edge uniquely identified by its path $p(e)$. The insertion or deletion of e can be performed by creating new versions of only those rules that lie on the path $p(e)$. All rules in R that are not part of this path are unaffected and remain unchanged.*

Proof. Let the rules corresponding to $p(e)$ be $r(e) = \langle r_0, \ldots, r_k \rangle$, where r_0 is the root rule and r_k is the rule directly containing e. The update is performed via a bottom-up, copy-on-write strategy. First, we create a new version of the leaf rule, r'_k, by adding or removing the edge e from a copy of r_k. Then, we iteratively ascend the path. For each rule r_i on the path (from r_{k-1} up to r_0), we create a new version, r'_i, which is identical to r_i except that its reference to the child rule r_{i+1} is replaced with a reference to the newly created r'_{i+1}. Crucially, no rule is ever modified in-place. Any rule not on the path remains unchanged. Furthermore, any part of the graph that previously shared a rule $r_i \in r(e)$ continues to point to the original, unaffected version of r_i. The modification is thus confined to the newly created chain of rules $\langle r'_0, \ldots, r'_k \rangle$. □

However, a challenge arises from the update mechanism in Theorem 4: the poor compression efficiency of newly generated rules. The copy-on-write process creates a chain of new rules, but each is referenced only once. This low utility means they not only fail to improve compression but actively harm it. To analyze and ultimately fix this issue, we first need to formally quantify a rule's efficiency.

Theorem 5 (**Rule Efficiency**). *Given a rule $r \in R$ that is referenced f times in rule-based compressed graph G', the total number of elements saved by this rule r is given by $S(r) = f(|r|-1)-|r|$, where $|r|$ is the length of the rule.*

Proof. The net saving from a rule r is the difference between the gross savings from its applications and the cost of its storage. Each of the f references to rule r replaces a sequence of $|r|$ elements with a single pointer to r, yielding a gross saving of $f(|r|-1)$ elements. However, the rule r itself, being a sequence of length $|r|$, must be explicitly stored, which incurs a cost of $|r|$ elements. Therefore, the total number of elements saved, $S(r)$, is the gross savings minus the storage cost, which can be expressed as $S(r) = f(|r|-1)-|r|$. □

Therefore, according to Theorem 5, a rule yields a net compression saving if and only if it has a length of at least two and is referenced two or more times. However, a naive application of the update strategy in Theorem 4 creates a chain of new rules, each referenced only once. This leads to a negative compression saving, as each new rule yields a saving of $S(r) = -1$. Moreover, replacing the original sub-rule in the root rule reduces its reference count, further decreasing its saving by

$|r| - 1$. This naive strategy, therefore, results in a total redundancy of (number of copied rules + $|r| - 1$), negating any potential compression benefits.

To maintain compression benefits, we flatten all copied elements into a single sequence during recursive copy operations, which prevent the generation of rules with non-positive efficiency (i.e., $S(r) \leq 0$), and use it to substitute the original element at the root of the deletion path. In addition, a *two-phase cleanup strategy* is applied in background: *rule-folding* and *secondary compression*. The first phase, *rule-folding*, folds and removes sub-rules with $S(r) \leq 0$ into their parents by replacing the pointer with the sub-rule's full content, thereby preserving local efficiency. However, rule-folding is effectively a form of local decompression that can degrade the global compression ratio. To counteract this, the second phase consists of a periodic *secondary compression* running in background. Rules affected by rule-folding are marked for this procedure, which re-applies the compression algorithm under the strict condition that it only introduces new rules with a net positive saving ($S(r) > 0$), thus restoring compression efficiency.

Index Maintenance. Following direct deletion, index maintenance becomes essential and consists of two steps: (1) removing indexes built on copied elements, and (2) shifting indexes built on vertices to the right of the fold point in the root rule. We accomplish this by scanning the index list of the source vertex of the edge to be deleted. If an index item corresponds to an element contained in the obsolete element, we remove it. If the index item is positioned after the folded point, we adjust its path offset within the rule by adding the length of the folded sequence to the first item of its path. To ensure the correctness of this process, we formally define the required transformations in Theorem 6.

Theorem 6 (Index Maintenance Correctness). *Let $G'(\Sigma, R)$ be a compressed graph equipped with index list I, where $I(v)$ denotes the set of index entries for a vertex v. Consider the deletion of an edge (v_{src}, v_{des}) with path $\Pi_p = \langle \delta_0, \delta_1, \ldots, \delta_k \rangle$, which triggers a "rule-folding" event. This event proceeds as follows:*

- *Rules along the edge's path Π_p are copied and flattened into a temporary sequence.*
- *The deletion is performed on this temporary sequence, resulting in a new final sequence, s.*
- *The single symbol at offset δ_0 in the root rule is replaced by this new sequence s.*

The index I remains correct after this operation if and only if the following two conditions are met:

(1) ***Removal****: Stale index entries for the source vertex v_{src} starting at path offset δ_0 are removed:*

$$\forall i \in I(v_{src}),\ \text{if } i.path[0] = \delta_0,\ \text{then } i \text{ is removed.}$$

(2) ***Shifting:*** *All index entry paths starting after δ_0 are shifted by the net change in the root rule's length, which is $\ell(s) - 1$:*

$$\forall i \in I,\ \text{if } i.path[0] > \delta_0,\ \text{then } i.path[0] \leftarrow i.path[0] + \ell(s) - 1.$$

For detailed proof, please refer to the Appendix.

Index maintenance is also essential after *secondary compression phase. Index list* is rebuilt over these recompressed adjacency lists to ensure correctness for future operations.

Example. Fig. 7 provides an example of direct deletion. To delete edge (v_0, v_5) from the rule-based compressed graph, we first recursively copy the rules along its path to form a flattened sequence and then remove v_5. Next, we replace the original pointer to r_2 in the root rule with this modified sequence. Lastly, we perform index maintenance by deleting the entry $v_5 \rightarrow \{2, 1, 1\}$ from v_0's *index list*. Periodically, a background two-phase cleanup is triggered, which folds the singly-referenced rule r_2 into its parent, r_1, and subsequently recompresses the affected rules.

Algorithm. The direct deletion algorithm is shown in Algorithm 3. Upon receiving an edge e for deletion, the algorithm first retrieves the derivation path to e in G' (Line 2). It then recursively creates

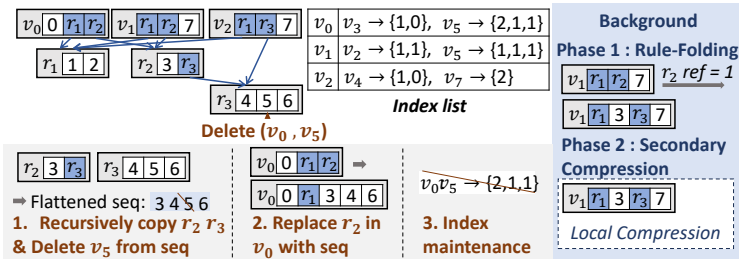


Fig. 7. An example of direct deletion.

copy of the rules along this path, from which e is removed. This new sequence subsequently replaces the original sub-rule pointer to complete the deletion (Lines 3, 7-8). Following this modification, the algorithm updates the reference count of the original, now-replaced sub-rule (Line 9) and the corresponding vertex index to ensure data integrity (Line 4). Finally, a folding and secondary compression step is triggered (Line 5), where rules with low reference counts are folded into their parents to maintain high compression efficiency throughout the graph's lifecycle (Lines 10-14).

Algorithm 3 Direct Deletion Algorithm

Require: Compressed graph $G' = (\Sigma, R)$ with root rule r_0 and *index list* I, edges to delete $e = (v_{src}, v_{des})$
Ensure: e is deleted from G' and I is correct

1: **function** DirectDelete(G', e)
2: $\Pi_p = \langle \delta_0, \delta_1, \ldots, \delta_k \rangle \leftarrow$ RetrievePath(G', e)
3: RecursivelyCopyAndRemoveEdge(Π_p, e)
4: Update $I(v_{src})$
5: FoldAndSecondaryCompress(G')
6: **function** RecursivelyCopyAndRemoveEdge(Π_p, e)
7: $s \leftarrow$ Copy rules along Π_p and store them as a flattened sequence
8: Delete e from s and replace $r_0[\delta_0]$ with s
9: Decrement ref_count of the rule at $r_0[\delta_0]$
10: **function** FoldAndSecondaryCompress(G')
11: **for** rule r with $S(r) \leq 0$ **do**
12: Fold r into its parent rule
13: $R_{comp} \leftarrow$ Rules with folded sequences
14: Compress(R_{comp})

Complexity. The time complexity of the recursive copy-and-delete operation is $O(\Sigma_{r \in Copied\ Rules}|r| + |I(v_{src})|)$. This comprises the first item for the deletion process itself and the second item for maintaining the index of the source vertex, where $|I(v_{src})|$ is the size of its index list, and the space complexity is $O(\Sigma_{r \in Copied\ Rules}|r|)$.

6 Evaluation

In this section, we measure the performance of direct insertion and deletion in three aspects: query performance, execution time, and space efficiency.

6.1 Experimental Setup

Methodology. We evaluate our framework (denoted as *Direct*) against a diverse set of baselines representing three distinct approaches:

- **Dynamic Compressed Graph Systems.** Frameworks that, like our work, support updates directly on compressed or specialized data structures. Baselines include Aspen [23], GraphZIP [47], and a dynamic k^2-tree implementation [21].
- **Dynamic Graph Management Systems.** High-performance, in-memory systems operating on uncompressed data, which serve as a benchmark for raw update and query speed. Baselines include Teseo [22], Sortledton [27], LiveGraph [68], and Spruce [51].
- **Static Compression (Decompress-Recompress).** Methods lacking native update support. We evaluate them using a full decompress-recompress workflow to quantify the high cost of applying updates to static formats. Baselines include Gzip, WebGraph [10], and our rule-based compression (denoted as *Offline*).

System Setup. The experiments are conducted on a machine running Ubuntu 20.04.4, equipped with a 12th Gen Intel(R) Core(TM) i7-12700K processor (3.61 GHz) and 128 GB of RAM. The implementation of our direct update framework was built in C++ and compiled by GCC 11.4.0.
Argument Setup. The parameters for our experiments were configured as follows. For our *index list* structure, we set the *interval* and *depth* parameters to 10. In our insertion strategy, we employed a buffer with a threshold of 100,000. Consequently, an L1 subgraph generation is triggered upon the accumulation of 100,000 edge insertions. An L1-L2 merge is triggered after 10 L1 subgraphs generation. To manage deletions, we initiated a background two-phase background cleanup for every 100,000 edge deletions.
Datasets. We evaluate using diverse real-world and synthetic graphs. From the SuiteSparse Matrix Collection [55], we use six standard benchmark graphs and the billion-scale *sk-2005* graph (detailed in Table 1). These datasets are not only common benchmarks in related works [5, 19, 48, 58, 67], but also exhibit significant structural regularities ($degree \geq 100$)—the primary target for rule-based compression. They span a wide spectrum of scales (from 14.8M to 1.95B edges) and topological complexities (e.g., rule depth from 31 to 3467), providing diversity crucial for testing our approach. Table 1 details properties of these datasets, including file compression ratio (FCR), rule count (R.C.), max rule depth (M.D.), and avg. path length (Avg.P.L.). The file compression ratio is defined as:

$$FCR = \frac{\text{size of uncompressed CSR format graph}}{\text{size of compressed graph}}.$$

Table 1. Real-world datasets with properties.

Dataset	Vertices	Edges	FCR	R.C.	M.D.	Avg.P.L.
pkustk14	151k	14.8M	10.53	372k	31	9.62
coPapersDBLP	540k	30.5M	4.41	1.35M	204	7.24
coPapersCiteseer	434k	32.1M	6.14	1.05M	115	8.65
dielFilterV3clx	420k	32.9M	5.45	1.76M	93	11.14
RM07R	381k	37.5M	11.40	840k	42	11.92
hollywood-2009	1.14M	113M	2.30	5.75M	1075	54.09
sk-2005	50.6M	1.95B	5.67	43.9M	3467	16.41

We also use the LDBC [25] *SF 10* graph to test dynamic performance, and generate synthetic graphs using RMAT [35] to test scalability, as shown in Table 2. The parameters a, b, and c are arguments for RMAT generator.

6.2 Query and Update Performance

Mixed Workload Performance. We evaluate a mixed workload of 200,000 streaming updates (90% insertions, 10% deletions, following Aspen [20]) and concurrent BFS queries. As shown in Table 3, our

Table 2. RMAT datasets.

Datasets Info	Abbr.	Vertex	Edge	a	b	c
small dense	sd	10k	500k	0.45	0.22	0.22
near uniform	nu	1M	10M	0.25	0.25	0.25
medium standard	ms	1M	10M	0.45	0.22	0.22
high skew	hs	1M	10M	0.57	0.19	0.19
large sparse	ls	10M	50M	0.45	0.22	0.22
xlarge	xl	50M	500M	0.45	0.22	0.22

Table 3. Update throughput and query latency comparison across four methods.

Dataset	Teseo		Sortledton		Aspen		Direct	
	Upd.	Qry.(s)	Upd.	Qry.(s)	Upd.	Qry.(s)	Upd.	Qry.(s)
pkustk14	5.15×10^5	1.77	6.35×10^5	2.81×10^{-2}	3.37×10^5	8.68×10^{-3}	$7.75 \times 10^{5*}$	$6.19 \times 10^{-3*}$
coPapersCiteseer	3.61×10^5	3.45	4.57×10^5	4.37×10^{-2}	2.60×10^5	$1.06 \times 10^{-2*}$	$5.18 \times 10^{5*}$	1.34×10^{-2}
coPapersDBLP	3.80×10^5	3.72	5.09×10^5	4.70×10^{-2}	2.62×10^5	$8.71 \times 10^{-3*}$	$5.33 \times 10^{5*}$	1.18×10^{-2}
RM07R	3.01×10^5	5.64	6.06×10^5	3.00×10^{-2}	2.63×10^5	1.21×10^{-2}	$6.67 \times 10^{5*}$	$4.75 \times 10^{-3*}$
dielFilterV3clx	4.05×10^5	3.57	$5.93 \times 10^{5*}$	3.36×10^{-2}	2.59×10^5	1.44×10^{-2}	5.36×10^5	$1.34 \times 10^{-2*}$
hollywood-2009	3.93×10^5	14.71	5.33×10^5	1.29×10^{-1}	2.83×10^5	$1.67 \times 10^{-2*}$	$6.01 \times 10^{5*}$	2.41×10^{-2}

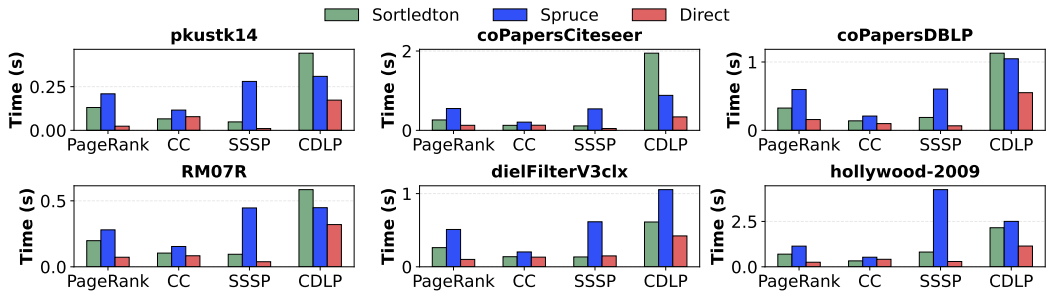


Fig. 8. Query times of LDBC Graphalytics benchmark.

Direct method achieves superior update throughput, outperforming Teseo, Sortledton, and Aspen by 56.9%, 8.9%, and 117.6% on average, respectively. For query latency, *Direct* is highly competitive, performing on par with Aspen and orders of magnitude faster than Teseo and Sortledton. This result validates our system's ability to balance high-speed updates with fast queries.

Analytical Query Performance. We further evaluate performance on key LDBC Graphalytics benchmarks, comparing against Sortledton and Spruce in Fig. 8. *Direct* demonstrates strong performance for complex analytics. For PageRank, *Direct* is 3.2× faster than Sortledton and 6.2× faster than Spruce on average. For SSSP, *Direct* is 2.7× faster than Sortledton and 13.0× faster than Spruce on average. For CDLP, *Direct* achieves average speedups of 2.5× against Sortledton and 1.8× against Spruce, respectively. *Direct*'s performance for CC is also competitive with the baselines. These results show that *Direct*'s benefits extend beyond simple queries, efficiently supporting complex graph algorithms directly on compressed data. This speedup comes from our compression that reduces redundant topological patterns, which directly reduces redundant computations in analysis tasks.

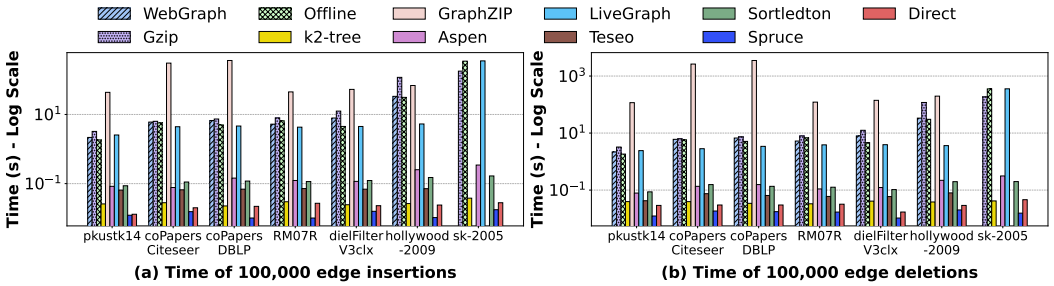


Fig. 9. Time of 100,000 edges batch insertions and deletions between different methods in log scale.

Batch Update Performance. We evaluate the batch execution time (100,000 edge insertions/deletions, 20 threads) in Fig. 9 (log scale) across three baseline categories. First, *Direct* is orders of magnitude faster than static methods (*Offline*, Gzip, WebGraph), confirming the prohibitive cost of the decompress-recompress workflow. Second, among dynamic *compressed* systems, *Direct* outperforms Aspen and GraphZIP, with latency comparable to k^2 tree, as its matrix-based structure allows for fast random-access bit flips. However, k^2-tree's speed comes at the cost of a much larger memory footprint (Section 6.3) and limited query support, highlighting our system's superior overall balance. Finally, compared to high-throughput dynamic systems, Spruce is the fastest. This is expected, as Spruce is purpose-built with a tree-like multilevel structure specifically to maximize update throughput. *Direct*, in contrast, is designed to efficiently add dynamic capabilities to a rule-based compressed structure that is fundamentally optimized for space efficiency and query performance. Even so, *Direct* outperforms other established dynamic systems, including Teseo (2.86× faster) and Sortledton (5.24× faster). This performance difference largely reflects different design priorities: *Direct* employs a lightweight lock-based design that decouples online updates from background maintenance, whereas systems like Teseo and Sortledton provide robust features such as full ACID transactional guarantees, which inherently incur additional overhead.

Vertex Update. We also conduct experiments on batch vertex updates (1,000 insertions/deletions). As mentioned in Section 4, we perform vertex deletions by deleting all associated in-edges and out-edges. We compare against Teseo, as it is the only baseline that natively supports true vertex deletions. Our vertex insertion is two orders of magnitude faster, as it only involves creating a new adjacency list. Our vertex deletion is one order of magnitude faster.

6.3 Space Efficiency

Peak Memory Usage. Table 4 compares the peak memory consumption of 100,000 insertions and deletions, measured as Resident Set Size (RSS), for all methods. As demonstrated, our proposed method, *Direct*, achieves a remarkable reduction in memory usage compared to all baseline methods. Specifically, *Direct* reduces memory usage by an average of 15.0% compared to the best static compression method, 50.1% compared to the best dynamic compressed graph system, and 47.1% compared to the best uncompressed dynamic graph management system. Note that Teseo fails on *sk-2005* due to out of memory.

Compression Ratio. Table 5 compares the in-memory compression ratio, defined as the total size of vertex and edge data divided by their combined compressed size, after 100,000 updates between the *Offline* (decompress-recompress) baseline, our *Direct* (naive), and *Direct+GC* (with background cleanup) methods. The results show our background cleanup (*Direct+GC*) is highly effective. For deletions, *Direct+GC* is crucial, recovering an average of 2.82% CPR over the naive *Direct* method, which otherwise performs 8.75% worse than *Offline*. More notably, for insertions, our

Table 4. Memory usage comparison across datasets (in MB).

Method	pkustk	Citeseer	DBLP	RM07R	diel	holly	sk-05
WebGraph	245	534	520	612	545	1851	34020
Gzip	384	794	950	919	982	3525	35223
Offline	268	616	620	644	596	2633	35840
k^2-tree	569	1299	1376	1893	1301	2663	68858
GraphZIP	497	1170	1112	1010	1136	3490	63969
Aspen	1173	1253	1243	1248	1273	2218	45568
LiveGraph	1235	2843	2752	3474	2596	5280	64143
Teseo	2804	3886	4113	3614	3908	8137	-
Sortledton	576	801	794	820	775	3596	65810
Spruce	417	1007	896	852	935	7398	68914
Direct	**197**	**448**	**504**	**370**	**472**	**1800**	**30404**

method consistently surpasses *Offline*. This is because *Offline*'s global recompression is degraded by fragmented updates, whereas our approach preserves existing rule structures, proving more robust.

Table 5. Comparison of in-memory compression ratios under 100,000 dynamic operations.

Dataset	Insertion			Deletion		
	Offline	Direct	Direct+GC	Offline	Direct	Direct+GC
pkustk14	9.850	11.863	11.868	7.949	6.243	6.770
coPapersCiteseer	6.010	7.215	7.216	5.872	5.632	5.758
coPapersDBLP	4.342	5.179	5.179	4.284	4.177	4.206
RM07R	11.039	13.976	13.981	10.405	9.045	9.297
dielFilterV3clx	5.317	7.222	7.224	5.103	4.597	4.700
hollywood-2009	2.291	2.555	2.555	2.286	2.252	2.264

Space Breakdown. First, we analyze the primary components of the system's memory usage. Table 6 shows the memory footprint of *index list* in MB. Compared with Table 4, *index list* constitutes the vast majority of the memory footprint, ranging from 81.7% on the *hollywood-2009* dataset to 88.4% on *pkustk14*. Notably, this provides a clear mechanism for optimization: in scenarios where minimizing memory is the top priority, a larger *index list interval* can be configured to achieve an even smaller memory footprint. Based on our parameter analysis in Section 6.5, we recommend an *index list interval* of 10, as it provides an excellent balance between performance and space savings.

Second, we break down the compression ratio recovery to quantify the impact of our background cleanup. For two-phase deletion cleanup, the results show that the rule-folding phase by itself improves the compression ratio by less than 0.01 in all cases. This minor improvement is expected, as each folded rule only saves a single pointer field; its primary purpose is to prepare the structure for the subsequent secondary compression. Consequently, the vast majority of the compression recovery comes from this secondary compression phase. This confirms that rule-folding functions primarily as a preparatory step that simplifies the rule structure, enabling the secondary compression to effectively identify new patterns and restore high space efficiency. For L1-L2 merge in insertion, the compression ratio is the same as Insertion *Offline* shown in Table 5.

Table 6. Index Size Comparison (in MB)

Dataset	pkustk	Citeseer	DBLP	RM07R	diel	holly
Index Size (MB)	175	390	436	326	412	1471

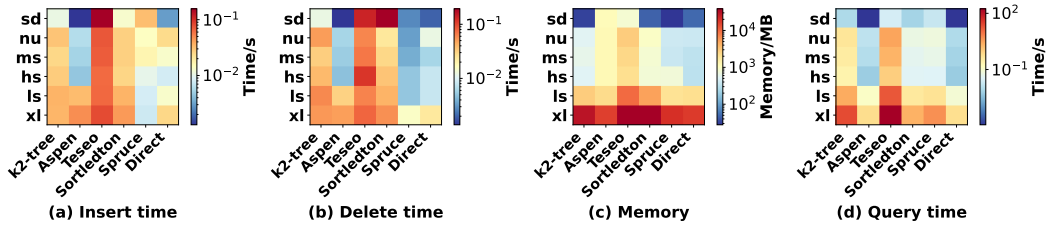


Fig. 10. Performance comparison on RMAT datasets. Time (s) is shown in log scale; memory in MB.

Cleanup Memory Overhead. We also evaluate the additional memory overhead required by the background cleanup process of direct insertion and deletion, which is primarily attributed to graph compression tasks. For cleanup operations triggered by insertions, the L1 subgraph generation consumes an average of 7.62 MB. The subsequent L1-L2 merge process is more demanding, requiring an average of 598.66 MB for the first five datasets and 2976.42 MB for the larger *hollywood-2009* dataset. In addition, cleanup following deletions consuming 38.87 MB on average. Crucially, this localized cleanup approach yields substantial memory savings compared with recompressing the entire graph.

6.4 Scalability and Stress Tests

Data Scalability. We measure data scalability on two fronts. First, on the billion-scale sk-2005 graph, we perform 100,000 batch insertions and deletions. As shown previously in Fig. 9 and Table 4, *Direct* maintains both stable update throughput and a clear space advantage at this scale. This result directly addresses the concern about the overhead of our recursive copy-on-write (COW) mechanism, which is employed during batch deletions. Our experiments confirm that the overhead of COW is not proportional to the total graph scale (i.e., the 1.95B edges). Instead, as our complexity analysis shows, the cost is determined by the number of elements copied, which depends on the average rule depth (i.e., the average path length). This average path length is primarily a function of the graph's topological redundancy, not its absolute size.

Furthermore, on RMAT-generated graphs of various sizes and densities, we evaluate update and BFS query scalability. Results in Fig. 10 confirm our significant advantages in space and query speed, while our update performance remains competitive, trailing only the specialized Spruce framework. As synthetic graphs, RMAT datasets are sparser and less regular than our real-world benchmarks, resulting in lower compression ratios ($< 2\times$). Consequently, our space-saving advantage is less pronounced, but *Direct* still achieves the lowest memory footprint on all RMAT sets except the small-dense (sd) graph. This lower compression ratio also implies shallower rule depths, which leads to higher update throughput than on the real-world datasets. We observe that Aspen's high update throughput on the four smaller graphs is due to its pre-allocated memory pool—a design choice that also makes it the most memory-intensive on those inputs. This advantage disappears on the larger graphs (*ls*, *xl*) once the pool is exhausted. Our query advantage remains significant, as even modest compression effectively reduces redundant computations during traversal.

Parallel Scalability. Fig. 11 analyzes our parallel scalability by batch size (up) and thread count (down). The batch size analysis shows that parallel processing (20 threads) is inefficient for small batches ($\leq$100), underperforming the serial stream (batch=1) approach due to management overhead.

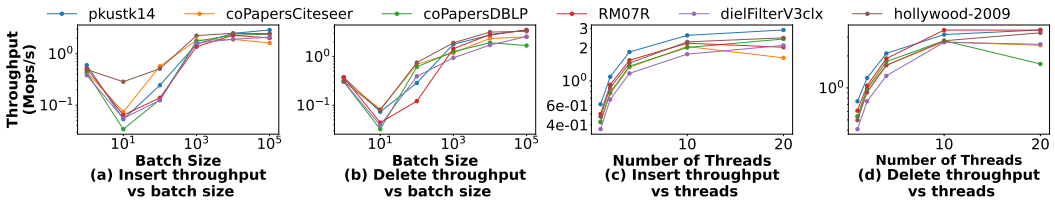


Fig. 11. Throughput of batch insertions (a,c) and deletions (b,d) under varying batch sizes (a,b) and thread counts (c,d).

Throughput for both insertions and deletions stabilizes at a batch size of 10,000. The thread count analysis demonstrates strong parallel scalability, with near-linear speedup up to 4 threads. This efficiency stems from the high independence of adjacency lists in our compressed structure, which minimizes execution conflicts. Performance growth continues but slows between 10 and 20 threads, indicating an approaching scalability bottleneck.

Workload Stress Tests. First, we benchmark 6 representative LDBC SNB update workloads on *SF 10* dataset against Sortledton in Fig. 12. We select these six workloads to cover a representative range of operations: simple edge updates (Insert 8, Delete 8), node insertions with local edges (Insert 1, Insert 6), and complex cascading deletes (Delete 1, Delete 6). Our approach achieves 35.4× higher throughput on average, while consuming only 62.7% of the memory. This significant speedup over Sortledton is due to two factors: (1) the *SF 10* graph generates shallower compression rules, leading to faster updates; (2) Sortledton's performance degrades significantly on complex cascading deletes (Delete 1, Delete 6) compared to the simple batch deletions tested earlier. Second, in an 80% deletion stress test on *coPapersDBLP* in Fig. 13, throughput remains stable, and query performance exhibits similar stability. This test cumulatively deletes 80% of the edges over 10 rounds, measuring throughput and graph size with and without background cleanup (GC) in each round. Enabling GC maintains 88-93% of the update throughput while providing significant space savings. The space efficiency gains from GC are most significant in the initial rounds and diminish over time. This is expected, as the continuous random deletions progressively destroy the original common patterns identified by compression, thus reducing the graph's overall structural regularity. Both methods maintain the final in-memory compression ratio above 1.40. Third, our hotspot contention test performs 100,000 insertions and deletions concentrated on 1,000 high-degree vertices. Fig. 14 shows a moderate throughput decrease compared to random updates due to data races, except for *hollywood-2009* deletion where its large rule set dilutes contention. All hotspot tests maintain similar memory usage to random updates.

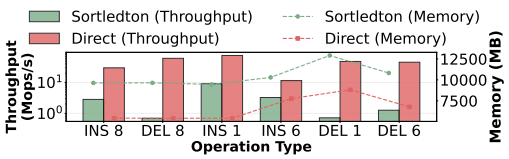


Fig. 12. Throughput and memory footprint when performing LDBC SNB update workloads.

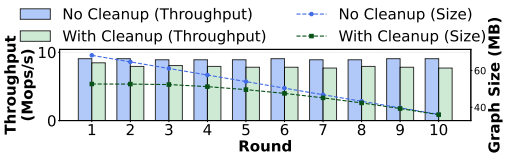


Fig. 13. Throughput and graph size when performing 80% deletion on *coPapersDBLP* with/without background cleanup.

6.5 System Internals and Ablation Study

Component Time Breakdown. A time breakdown for single update operations in Fig. 16 reveals that **Path Retrieval**—traversing the compressed rule structure to locate the target—is the primary

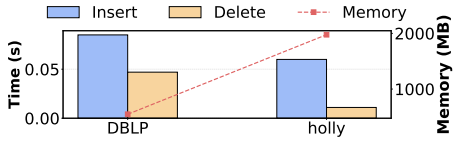


Fig. 14. 100,000 batch updates on 1,000 hotspots.

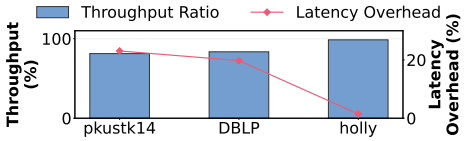


Fig. 15. Performance with live background cleanup.

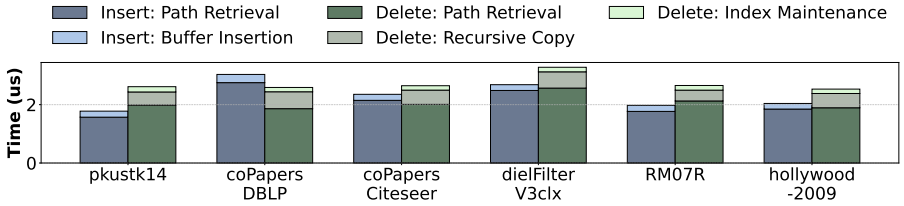


Fig. 16. Execution time breakdown of insert and delete.

bottleneck for both operations. This phase accounts for 90.5% of insertion time (vs. 9.5% for the lightweight *Buffer Insertion*) and 76.1% of deletion time (vs. 23.9% for structural reorganization via *Recursive Copy* and *Index Maintenance*). This cost distribution is remarkably consistent across all datasets.

Ablation Study. We conduct a component-level analysis to isolate the contribution of our key design components. **(1) Indexing:** Enabling the index boosts insertion and deletion throughput by an average of 3.07× and 2.31×, respectively. **(2) Buffered Insertion and COW Deletion:** The benefits of these two strategies are demonstrated in Fig. 9. By decoupling the update from the recompression, it achieves throughput orders of magnitude higher than the *Offline* decompress-recompress workflow. **(3) Background Merge:** L1-L2 background merge is essential for maintaining long-term performance and space efficiency. For throughput, background merge is critical for preventing performance collapse due to fragmentation. On dataset *coPapersDBLP*, background merge catapults throughput, delivering 3.01× faster edge queries and 2.77× faster insertions compared to its fragmented state (10 unmerged L1 subgraphs). For space efficiency, the merge achieves a compression ratio comparable to a full *Offline* recompression. Its memory overhead is also comparable to that of a full *Offline* recompression. **(4) Two-Phase Cleanup:** The primary role of two-phase cleanup is to maintain high compression ratio after deletion, while imposing minimal overhead on foreground updates. We validate this by comparing the naive *Direct* (COW without GC) method against *Direct+GC*. For space efficiency, the GC is crucial for maintaining compression ratio. As shown in Table 5, *Direct+GC* successfully recovers 2.82% compression ratio compared to naive *Direct*, resulting in a final compression ratio that is only 6.35% lower than a full *Offline* recompression on average. For throughput, the background process imposes minimal overhead. Our 80% deletion stress test in Fig. 13 confirms this, showing that *Direct+GC* retains 88-93% of the update throughput of naive *Direct*.

Background Cleanup Overhead. Our background cleanup tasks are highly efficient and minimally interfere with online performance. The absolute execution times are low: the frequent L1 insertion generation averages just 127 ms, the two-phase deletion cleanup averages 261 ms, and the less frequent L1-L2 merge averages 5.65s (scaling with graph size, e.g., 35.24s for *hollywood-2009*). More importantly, when running a mixed workload (80% insert, 10% delete, 10% edge query) concurrently with background cleanup, the impact on update throughput is moderate, ranging from a 1.43% overhead on *hollywood-2009* to 18.75% on *pkustk14*, as shown in Fig. 15. This concurrent cleanup process incurs a query latency overhead of 1.45% to 23.07%, an effect most pronounced on smaller graphs like *pkustk14* due to higher lock contention. This is because these workloads impact

a significantly larger portion of the graph's data—nearly 30% of the edges on *pkustk14*, versus less than 5% on *hollywood-2009*—leading to more frequent resource conflicts.

Parameter Sensitivity. We evaluate the impact of the *index list interval* on the average edge retrieval throughput, with the results presented in Table 7. The data shows a clear trend: throughput consistently decreases as the index interval grows, demonstrating the effectiveness of a denser index. While performance is similar for intervals of 5 and 10, the throughput at an interval of 10 is 3.75× faster than at an interval of 1000, confirming the significant benefit of our indexing scheme. In contrast to the interval size, we find that varying the *depth* parameter of the *index list* had a negligible impact on throughput.

Table 7. Average edge retrieval throughput with varying index list intervals.

Index List Interval	**5**	**10**	**20**	**50**	**100**	**1000**
Throughput (ops/s)	679k	648k	537k	405k	272k	173k

Read Performance. To assess read performance stability, we measure edge retrieval throughput before and after a workload of 100,000 updates. We observe no significant degradation in performance. This stability holds even with the presence of temporary L1 subgraphs created by insertions, as their small size relative to the main graph adds negligible overhead to the path retrieval process. This demonstrates that our direct update mechanism effectively preserves query performance in a dynamic environment.

6.6 Discussion

Generalizability. The generalizability of our direct update method hinges on local decodability, which provides the random edge access that updates require. This makes our approach readily applicable to mainstream locally decodable frameworks, like virtual node [13], k^2-tree [21] and clique-based compression [47]. While our method is optimized for frameworks that support local decodability, it can also be applied to a broader range of graph compression. For sequential methods like the LZ-family, which do not inherently provide local decodability, our approach can be effectively complemented by chunk-based compression techniques [23]. These methods introduce block-level localization of updates, thereby facilitating compatibility with a wider variety of compression strategies.

Practical Implications and Scope. Our primary contribution is the first framework for direct updates on rule-based compressed graphs, demonstrating the feasibility and efficiency of dynamic operations on this highly compressed structure. The practical benefits are naturally most pronounced on graphs with the structural regularity that grammar-based compression thrives on. Our current prototype is an in-memory system, allowing us to focus on these novel algorithmic challenges. This provides a solid foundation for future work.

Supported Queries. Graph queries typically fall into three main categories: neighbor access, adjacency traversal, and pattern matching. Our update-aware representation is designed to natively support the first two—neighbor access and adjacency traversal—as fundamental primitives. Consequently, complex queries composed of these primitives are also directly supported. Pattern matching queries, the third category, can then be supported by decomposing them into a sequence of these fundamental operations.

6.7 Future Work

ACID Transactions. Our framework is designed to be extensible to support ACID transactions. Atomicity, Consistency, and Durability can be achieved using a Write-Ahead Log (WAL). Both online updates and background cleanups would apply copy-on-write logic, creating new versions of rules rather than modifying them in-place. A transaction is committed and durable once its log record is flushed, at which point pointers are atomically swapped to make the changes live. Isolation can be implemented via Multi-Version Concurrency Control (MVCC) to provide Snapshot Isolation. Each transaction would receive a unique ID (TxID) and operate on a consistent, versioned snapshot of the graph, thereby preventing readers and writers from blocking each other.
Potential Extensions. Our framework can be naturally extended to support weighted and attributed graphs by decoupling topology from data. Edge weights can be stored in a parallel structure, remaining retrievable due to our compression's preservation of relative neighbor order, with aggregates (e.g., min, max) stored for rules. Vertex attributes can be held in an auxiliary structure accessible by ID, as our method only targets the graph structure. We also believe approximate aggregation queries (e.g., top-k) are supportable by decomposing them into the fundamental neighbor access and traversal operations our framework provides.
Synergy with Recent Advancements. Our work is also synergistic with recent advancements. Our in-memory framework could leverage LSMGraph [59] as an ideal persistent storage system. It can also serve as a dynamic backend for summarization frameworks like GraphSUM [50] and approximate querying [24], enabling their analytics on evolving graphs. Furthermore, HAL's [4] strategies for out-of-order updates are complementary, potentially helping to better balance the space and performance trade-offs of our order-sensitive compression.

7 Conclusion

In this paper, we introduced a novel framework that enables direct updates on rule-based compressed graphs, successfully resolving the trade-offs among space efficiency, update latency, and query performance. We achieve this through several key innovations: a specialized *index list* for random-access queries; localized update operations for high responsiveness; and tailored background cleanup strategies—periodic merging for insertions and two-phase cleanup for deletions—to maintain high compression ratios without degrading query performance. Experimental results validate the superiority of our approach across all three metrics. This work confirms the feasibility of efficient dynamic compressed graph management and provides a foundation for scalable processing of evolving graph.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Nos. U25B2018, 62322213, and 62461146205), the Beijing Science and Technology Project (No. Z251100008125032), the China Postdoctoral Science Foundation (Grant Nos. GZC20251044 and 2025M781495), and the Shuimu Scholar Fellowship of Tsinghua University (Grant No. 2025SM061). Lin Feng, Feng Zhang, Zheng Chen, Yuxin Tang, Jiawei Guan, and Xiaoyong Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and the School of Information, Renmin University of China. Feng Zhang is the corresponding author of this paper.

References

[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4, Article 20 (Oct. 2017), 44 pages.
[2] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. 2015. Succinct: Enabling queries on compressed data. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USA, 337–350.

[3] Ahmed Al-Baghdadi and Xiang Lian. 2020. Topic-based Community Search over Spatial-Social Networks. *Proceedings of the VLDB Endowment* 13, 11 (2020), 2104–2117.

[4] Angelos Christos Anadiotis, Muhammad Ghufran Khan, and Ioana Manolescu. 2024. Dynamic Graph Databases with Out-of-Order Updates. *Proc. VLDB Endow.* 17, 13 (Sept. 2024), 4799–4812.

[5] Noushin Azami and Martin Burtscher. 2022. Compressed In-memory Graphs for Accelerating GPU-based Analytics. In *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. 32–40.

[6] Prithu Banerjee, Wei Chen, and Laks VS Lakshmanan. 2019. Maximizing Welfare in Social Networks under a Utility Driven Influence Diffusion Model. In *Proceedings of the 2019 International Conference on Management of Data*. 1078–1095.

[7] Maciej Besta and Torsten Hoefler. 2018. Survey and Taxonomy of Lossless Graph Compression and Space-Efficient Graph Representations. *arXiv preprint arXiv:1806.01799* (2018).

[8] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian. 1998. The Connectivity Server: Fast Access to Linkage Information on the Web. *Computer Networks and ISDN Systems* 30, 1-7 (1998), 469–477.

[9] P. Boldi, M. Santini, and S. Vigna. 2009. Permuting Web and Social Graphs. *Internet Mathematics* 6, 3 (2009), 257–283.

[10] P. Boldi and S. Vigna. 2004. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web*. ACM, New York, NY, USA, 595–602.

[11] N. R. Brisaboa, S. Ladra, and G. Navarro. 2009. k2-Trees for Compact Web Graph Representation. In *International Symposium on String Processing and Information Retrieval*. Springer, 18–30.

[12] G. Brodal and R. Fagerberg. 1999. Dynamic Representations of Sparse Graphs. In *Algorithms and Data Structures*. 773–773.

[13] G. Buehrer and K. Chellapilla. 2008. A Scalable Pattern Mining Approach to Web Graph Compression with Communities. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*. ACM, 95–106.

[14] D. Caro, M. A. Rodríguez, and N. R. Brisaboa. 2015. Data Structures for Temporal Graphs Based on Compact Sequence Representations. *Information Systems* 51 (2015), 1–26.

[15] Chengliang Chai, Guoliang Li, Jian Li, Dong Deng, and Jianhua Feng. 2016. Cost-Effective Crowdsourced Entity Resolution: A Partial-Order Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). 969–984.

[16] Chengliang Chai, Guoliang Li, Jian Li, Dong Deng, and Jianhua Feng. 2018. A Partial-Order-Based Framework for Cost-Effective Crowdsourced Entity Resolution. *The VLDB Journal* 27, 6 (2018), 745–770.

[17] Di Chen, Ye Yuan, Wenjin Du, Yurong Cheng, and Guoren Wang. 2021. Online Route Planning over Time-Dependent Road Networks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 325–335.

[18] Qing Chen, Michael H. Böhlen, and Sven Helmer. 2025. An Experimental Comparison of Tree-data Structures for Connectivity Queries on Fully-dynamic Undirected Graphs. *Proc. ACM Manag. Data* 3, 1, Article 10 (Feb. 2025), 26 pages.

[19] Zheng Chen, Feng Zhang, JiaWei Guan, Jidong Zhai, Xipeng Shen, Huanchen Zhang, Wentong Shu, and Xiaoyong Du. 2023. Compressgraph: Efficient parallel graph analytics with rule-based compression. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–31.

[20] F. Claude and G. Navarro. 2007. A Fast and Compact Web Graph Representation. In *International Symposium on String Processing and Information Retrieval*. Springer, 118–129.

[21] Miguel E. Coimbra, Alexandre P. Francisco, Luís M. S. Russo, Guillermo De Bernardo, Susana Ladra, and Gonzalo Navarro. 2020. On Dynamic Succinct Graph Representations. In *2020 Data Compression Conference (DCC)*. 213–222.

[22] Dean De Leo and Peter Boncz. 2021. Teseo and the analysis of structural dynamic graphs. *Proc. VLDB Endow.* 14, 6 (Feb. 2021), 1053–1066.

[23] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-Latency Graph Streaming Using Compressed Purely-Functional Trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 918–934.

[24] Stefania Dumbrava, Angela Bonifati, Amaia Nazabal Ruiz Diaz, and Romain Vuillemot. 2019. Approximate Querying on Property Graphs. In *Scalable Uncertainty Management*. Springer International Publishing, 250–265.

[25] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, 619–630.

[26] A. Farzan and J. I. Munro. 2008. Succinct Representations of Arbitrary Graphs. In *European Symposium on Algorithms*. Springer, 393–404.

[27] Per Fuchs, Domagoj Margan, and Jana Giceva. 2022. Sortledton: a universal, transactional graph data structure. *Proc. VLDB Endow.* 15, 6 (Feb. 2022), 1173–1186.

[28] Jun Gao, Jiazun Chen, Zhao Li, and Ji Zhang. 2021. ICS-GNN: Lightweight Interactive Community Search via Graph Neural Network. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1006–1018.
[29] Advitya Gemawat. 2021. GraphGem: Optimized Scalable System for Graph Convolutional Networks. In *SIGMOD '21: International Conference on Management of Data* (Virtual Event, China), Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). 2920–2922.
[30] S. L. González. 2011. *Algorithms and Compressed Data Structures for Information Retrieval.* Ph. D. Dissertation. Universidade da Coruña.
[31] Zihan Jiang, Fubing Mao, Yapu Guo, Xu Liu, Haikun Liu, Xiaofei Liao, Hai Jin, and Wei Zhang. 2023. ACGraph: Accelerating Streaming Graph Processing via Dependence Hierarchy. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6.
[32] C. Karande, K. Chellapilla, and R. Andersen. 2009. Speeding up algorithms on compressed web graphs. *Internet Mathematics* (2009).
[33] A. Khan, S. S. Bhowmick, and F. Bonchi. 2017. Summarizing Static and Dynamic Big Graphs. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1981–1984.
[34] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. 2017. ZipG: A Memory-Efficient Graph Store for Interactive Queries. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*. ACM, 1149–1164.
[35] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Scalable SIMD-Efficient Graph Processing on GPUs. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT '15)*. 39–50.
[36] J. Körner. 1973. Coding of an Information Source Having Ambiguous Alphabet and the Entropy of Graphs. In *Transactions of the Sixth Prague Conference on Information Theory*. 411–425.
[37] N. J. Larsson and A. Moffat. 2000. Off-line Dictionary-based Compression. *Proc. IEEE* 88, 11 (2000), 1722–1732.
[38] N. Lehmann and J. Pérez. 2015. Implementing graph query languages over compressed data structures: A progress report. In *In Alberto Mendelzon International Workshop on Foundations of Data Management*. 96.
[39] Jianxin Li, Xinjue Wang, Ke Deng, Xiaochun Yang, Timos Sellis, and Jeffrey Xu Yu. 2017. Most Influential Community Search over Large Social Networks. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 871–882.
[40] Tong Li, Kai Zheng, Ke Xu, Rahul Arvind Jadhav, Tao Xiong, Keith Winstein, and Kun Tan. 2020. TACK: Improving Wireless Transport Performance by Taming Acknowledgments. In *ACM SIGCOMM*. 15–30.
[41] Zijian Li, Lei Chen, and Yue Wang. 2019. G*-Tree: An Efficient Spatial Index on Road Networks. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 268–279.
[42] Meihao Liao, Cheng Li, Rong-Hua Li, and Guoren Wang. 2025. Efficient Index Maintenance for Effective Resistance Computation on Evolving Graphs. *Proc. ACM Manag. Data* 3, 1, Article 36 (Feb. 2025), 27 pages.
[43] Yuyu Luo, Xuedi Qin, Nan Tang, and Guoliang Li. 2018. Deepeye: Towards automatic data visualization. In *2018 IEEE 34th international conference on data engineering (ICDE)*. IEEE, 101–112.
[44] Amine Mhedhbi, Pranjal Gupta, Shahid Khaliq, and Semih Salihoglu. 2021. A+ Indexes: Tunable and Space-Efficient Adjacency Lists in Graph Database Management Systems. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1464–1475.
[45] Craig G. Nevill-Manning and Ian H. Witten. 1997. Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm. *Journal of Artificial Intelligence Research* 7, 1 (1997), 67–82.
[46] Craig G. Nevill-Manning and Ian H. Witten. 1997. Linear-Time, Incremental Hierarchy Inference for Compression. In *Proceedings of the Data Compression Conference (DCC'97)*. IEEE, 3–11.
[47] Ryan A. Rossi and Rong Zhou. 2018. GraphZIP: a clique-based sparse graph compression method. *Journal of Big Data* 5, 1 (2018), 10.
[48] Subhajit Sahu and Kishore Kothapalli. 2025. GVEL: Fast Graph Loading in Edgelist and Compressed Sparse Row (CSR) Formats. In *Euro-Par 2024: Parallel Processing Workshops*. Springer Nature Switzerland, Cham, 268–280.
[49] Mo Sha, Yuchen Li, and Kian-Lee Tan. 2019. GPU-Based Graph Traversal on Compressed Graphs. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. ACM, 775–792.
[50] Nasrin Shabani, Amin Beheshti, Jia Wu, Maryam Khanian Najafabadi, Jin Foo, and Alireza Jolfaei. 2024. GraphSUM: Scalable Graph Summarization for Efficient Question Answering. In *EDBT*. 794–797.
[51] Jifan Shi, Biao Wang, and Yun Xu. 2024. Spruce: a Fast yet Space-saving Structure for Dynamic Graph Storage. *Proc. ACM Manag. Data* 2, 1, Article 27 (March 2024), 26 pages.
[52] Statista. n.d.. Number of daily active Facebook users worldwide 2023. https://www.statista.com/statistics/346167/facebook-global-dau/ Accessed: 2024-11-25.
[53] Jixian Su, Chiyu Hao, Shixuan Sun, Hao Zhang, Sen Gao, Jiaxin Jiang, Yao Chen, Chenyi Zhang, Bingsheng He, and Minyi Guo. 2025. Revisiting the Design of In-Memory Dynamic Graph Storage. *Proc. ACM Manag. Data* 3, 1, Article 70 (Feb. 2025), 27 pages.

[54] T. Suel and J. Yuan. 2001. Compressing the Graph Structure of the Web. In *Data Compression Conference, 2001. Proceedings. DCC 2001.* IEEE, 213–222.
[55] Texas A&M University. n.d.. Sparse Matrix Research Group. https://sparse.tamu.edu Accessed: 2024-11-25.
[56] Brian Wheatman, Xiaojun Dong, Zheqi Shen, Laxman Dhulipala, Jakub Łącki, Prashant Pandey, and Helen Xu. 2024. BYO: A Unified Framework for Benchmarking Large-Scale Graph Containers. *Proceedings of the VLDB Endowment* 17, 9 (May 2024), 2307–2320.
[57] Derong Xu, Jingbo Zhou, Tong Xu, Yuan Xia, Ji Liu, Enhong Chen, and Dejing Dou. 2023. Multimodal Biological Knowledge Graph Completion via Triple Co-Attention Mechanism. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 3928–3941.
[58] Qian Xu, Juan Yang, Feng Zhang, Zheng Chen, Jiawei Guan, Kang Chen, Ju Fan, Youren Shen, Ke Yang, Yu Zhang, and Xiaoyong Du. 2024. Improving Graph Compression for Efficient Resource-Constrained Graph Analytics. *Proc. VLDB Endow.* 17, 9 (May 2024), 2212–2226.
[59] Song Yu, Shufeng Gong, Qian Tao, Sijie Shen, Yanfeng Zhang, Wenyuan Yu, Pengxi Liu, Zhixin Zhang, Hongfu Li, Xiaojian Luo, Ge Yu, and Jingren Zhou. 2024. LSMGraph: A High-Performance Dynamic Graph Storage System with Multi-Level CSR. *Proc. ACM Manag. Data* 2, 6, Article 243 (Dec. 2024), 28 pages.
[60] Feng Zhang, Weitao Wan, Chenyang Zhang, Jidong Zhai, Yunpeng Chai, Haixiang Li, and Xiaoyong Du. 2022. CompressDB: Enabling efficient compressed data direct processing for various databases. In *Proceedings of the 2022 International Conference on Management of Data*. 1655–1669.
[61] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. 2018. Efficient document analytics on compressed data: Method, challenges, algorithms, insights. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1522–1535.
[62] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. 2018. Zwift: A programming framework for high performance text analytics on compressed data. In *Proceedings of the 2018 International Conference on Supercomputing*. 195–206.
[63] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. 2021. TADOC: Text analytics directly on compression. *The VLDB Journal* 30, 2 (2021), 163–188.
[64] Nan Zhang, Yutong Ye, Xiang Lian, and Mingsong Chen. 2024. Top-L Most Influential Community Detection Over Social Networks. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 5767–5779.
[65] Qianru Zhang, Zheng Wang, Cheng Long, Chao Huang, Siu-Ming Yiu, Yiding Liu, Gao Cong, and Jieming Shi. 2023. Online Anomalous Subtrajectory Detection on Road Networks with Deep Reinforcement Learning. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 246–258.
[66] Kangfei Zhao, Zhiwei Zhang, Yu Rong, Jeffrey Xu Yu, and Junzhou Huang. 2022. Finding Critical Users in Social Communities via Graph Convolutions (Extended Abstract). In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 1539–1540.
[67] Zhiqiang Zhao, Ying Zhang, and Zhuo Feng. 2021. Towards Scalable Spectral Embedding and Data Visualization via Spectral Coarsening. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining* (Virtual Event, Israel) *(WSDM '21)*. Association for Computing Machinery, 869–877.
[68] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: a transactional graph storage system with purely sequential adjacency list scans. *Proc. VLDB Endow.* 13, 7 (March 2020), 1020–1034.
[69] Sandra Álvarez García, Borja Freire Castro, Susana Ladra, and Oscar Pedreira. 2019. Compact and Efficient Representation of General Graph Databases. *Knowledge and Information Systems* 60 (09 2019).

Received July 2025; revised October 2025; accepted November 2025