University of Pisa
Department of Computer Science
Bachelor's degree in Computer Science

Final Dissertation

A Computation-Friendly Compressed Graph Database

Supervisor
Paolo Ferragina

Co-supervisor
Lorenzo Bellomo

Candidate
Michelangelo Pucci

Academic year 2024/2025

Abstract

Graphs have become increasingly influential in both research and industrial contexts, as they provide a flexible way of modeling interconnected, multimodal information through nodes and edges. Graph Databases (GDBs), in particular, have emerged as key tools for managing and analyzing graph-structured information. Although commercial GDB solutions are widely adopted, they frequently exhibit space inefficiencies **PF:** TIME???, especially when handling static datasets, resulting in suboptimal performance due to memory hierarchy constraints. To tackle these inefficiencies, we developed a computation-friendly compressed Graph Database optimized for static or slightly dynamic data scenarios

In this thesis, we first introduce a technique for compressing graph adjacency lists, which achieves best-in-class compression performance relative to existing approaches on most of our evaluated datasets. Expanding upon this foundation, we then extend our approach to efficiently support Labelled Property Graphs by enriching the compressed graph structure with variable-length node and edge properties. A wide set of experimental benchmarks on known labeled graphs show that our Compressed Graph DB not only substantially outperforms commercial state-of-the-art solutions, such as Neo4j and ArangoDB, in terms of space efficiency, but also offers competitive query performance and superior time-space tradeoffs, even if compared to other compressed graph database implementations.**PF:** CHECK ALLA FINE

Contents

Chapter 1

Introduction

Graph Databases (GDBs) are specialized data stores designed to represent information as a network of nodes (entities) and edges (relationships), closely mirroring the complex, interconnected structures found in domains such as social media, recommendation systems, and biological data [2].

While a graph refers generically to any structure comprising entities and their connecting labeled/typed relationships, a Knowledge Graph (KG) specifically integrates semantic context by modeling entities, their attributes, and their interrelations according to a clearly defined schema or ontology [17]. In this way, KGs represent not just the connectivity between data items, but structured factual knowledge that can be systematically queried, traversed, and reasoned over. GDBs naturally lend themselves as the ideal technological framework for implementing and managing knowledge graphs due to their inherent structural alignment.

This inherent structure enables efficient traversal through direct connections, possibly driven by their labels, in contrast to relational databases, which rely on computationally expensive join operations to establish relationships.

The longstanding interest in KGs, particularly in search engines [16], has grown significantly with their increasing role in Retrieval-Augmented Generation (RAG) systems [32]. In these applications, Knowledge Graphs enhance Large Language Models (LLMs) by supplying context-rich, structured information, ultimately improving the relevance and accuracy of generated responses.

1.1 Labeled Property Graphs

We start by presenting the core concept of Labeled Property Graphs. A Labeled Property Graph (LPG) is a data structure used to represent the knowledge about a certain domain. It can be seen as a directed graph where each node and edge can have properties and labels assigned. A property is a $(key, value)$ pair that provides more information about the node or edge that is assigned, while a label l is a marker that is typically used to indicate the type, role, or classification that a node or edge has in the graph. While vertices and edges can be associated with any number of properties, typically, they have exactly one label assigned.

Definition 1.1 *Formally, we define a Labeled Property Graph G as*

$$G = (V, E, L, \mathrm{l}_V, \mathrm{l}_E, K, W, \mathrm{p}_V, \mathrm{p}_E).$$

where

- *V and E contain respectively the n vertices and m edges of the graph. E is defined as a multi-set, as we want to model cases with multiple edges between two nodes (for example, with different labels or properties assigned).*
- *L is a finite set of labels, used to annotate both nodes and edges*
- $\mathrm{l}_V : V \to L \cup \{\varepsilon\}$ *and* $\mathrm{l}_E : E \to L \cup \{\varepsilon\}$ *are the functions that return the label assigned to a vertex or edge (if it has one, ε otherwise)*
- *K is a finite set of keys used in key-value property pairs.*
- *W is a (possibly infinite) set of literal values that can be assigned to those keys (e.g., strings, numbers, booleans).*
- $\mathrm{p}_V : V \times K \to W \cup \{\varepsilon\}$ *and* $\mathrm{p}_E : E \to \mathcal{P}(K \times W)$ *are the functions that retrieve the property value associated with a given key for respectively a vertex or an edge, returning ε if the key is absent.*

Example 1.2 (Academic Relationship Graph) *The example in Figure 1.1 illustrates a Labeled Property Graph G representing interactions within a university. It models students, professors, and the university, along with their roles and affiliations.*

Let

$$G = (V, E, L, \mathrm{l}_V, \mathrm{l}_E, K, W, \mathrm{p}_V, \mathrm{p}_E).$$

The set of nodes V includes:

- Alan Turing *($v_1 \in V$), with label* `"Student"` *and properties such that* $p_V(v_1,$ `"name"`$) =$ `"Alan"`, $p_V(v_1,$ `"surname"`$) =$ `"Turing"`, *and* $p_V(v_1,$ `"age"`$) = 25$.
- Alonzo Churchill *($v_2 \in V$), with label* `"Professor"` *and properties such that* $p_V(v_2,$ `"name"`$) =$ `"Alonzo"`, $p_V(v_2,$ `"surname"`$) =$ `"Churchill"`, *and* $p_V(v_2,$ `"age"`$) = 34$.
- Princeton University *($v_3 \in V$), with label* `"University"` *and property such that* $p_V(v_3,$ `"name"`$) =$ `"Princeton University"`.

The edges E describe the relationships:

- *An edge $e_1 = (v_1, v_3) \in E$ with label "STUDIES_AT" and $p_E(e_1) = \{(\texttt{"since"}, 1936)\}$.*
- *An edge $e_2 = (v_2, v_3) \in E$ with label "WORKS_AT" and $p_E(e_2) = \{(\texttt{"since"}, 1932)\}$.*
- *An edge $e_3 = (v_2, v_1) \in E$ with label "SUPERVISES" and $p_E(e_3) = \emptyset$.*

The definitions of L, l_V, l_E, K, W, p_V, and p_E are provided by context.

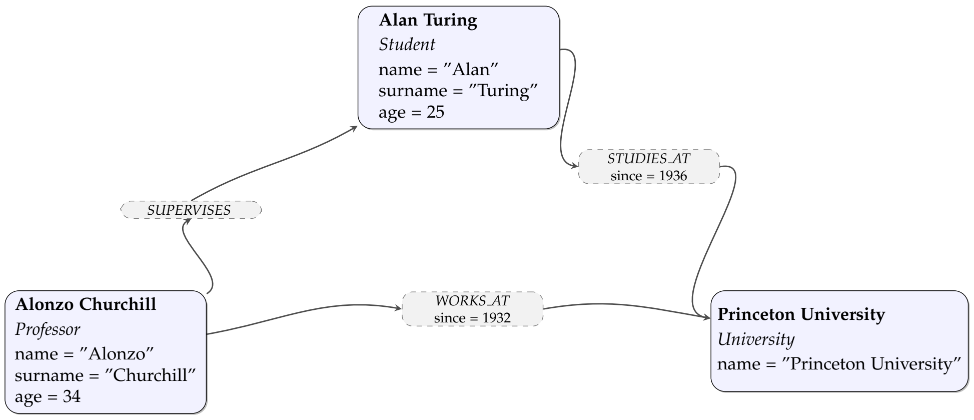

Figure 1.1: A simple example of an academic LPG. Node Properties and edges are displayed under the node/edge label.

1.2 Graph Databases

Graph databases (GDBs) have emerged as essential tools for managing large, dynamic, and richly attributed datasets. Although they are all designed to operate with graph-like data, GDBs differ in the features they offer and implementation details.

Besta et al. [3] presented a comprehensive taxonomy that differentiates state-of-the-art systems based on several interrelated dimensions. The first distinction lies in the underlying data models. GDBs are first divided into *Native Graph Databases* and *Key-value/Document stores*. Native Graph Databases use a graph data model as their underlying schema, representing the relationships between the data explicitly, while the latter, such as ArangoDB [1], are more general NoSQL data stores where data is stored as a collection of key–value pairs. Most of the Native Graph Databases, such as Neo4j, use some variants of the LPG model. On the other hand, other Native Graph Databases such as Amazon Neptune [2] rely on the Resource Description

[1] https://arangodb.com/
[2] https://aws.amazon.com/it/neptune/

Framework (RDF) model [25], storing the data in terms of *subject-predicate-object* triples. Subjects typically refer to resource identifiers, while an object can be a simple value or a resource identifier.

The design of a Graph Database architecture inevitably influences the performance when serving different types of requests. We distinguish between Online Transactional Processing (OLTP) and Online Analytical Processing (OLAP) requests. The first involves the processing of small and close portions of the graph, for example, through property lookups or neighbor queries. The latter are more complex queries that span large regions of the graphs and may involve computing the PageRank [22] of the nodes, computing shortest paths between a given pair of nodes, or visiting the graph through a Breadth-First search (BFS).

1.2.1 Neo4j

Neo4j is a prominent native graph database management system, purpose-built for efficiently storing, querying, and processing graph data.[3] It adopts the Labeled Property Graph (LPG) model, as previously defined (Definition 1.1), where both nodes and edges (relationships) can carry multiple labels and a set of key-value properties.

As one of the most widely adopted and mature graph databases[4], Neo4j benefits from a large community, industry support, and enterprise-level deployment. It employs Cypher[5] as its declarative query language, designed to facilitate expressive graph pattern matching and traversal.

Under the hood, Neo4j uses a native graph storage engine, leveraging index-free adjacency to optimize relationship traversal independent of the overall graph size.[6] Additionally, Neo4j ensures full ACID compliance for transactional consistency and reliability.[7]

1.2.2 ArangoDB

ArangoDB[8] represents a native multi-model database that integrates graph, document, and key/value data models within a single engine and query language. It stores graph data by representing vertices and edges as JSON documents within distinct collections. Edges make use of special `_from`

[3]Neo4j documentation: https://neo4j.com/docs/getting-started/graph-database/
[4]GDB popularity ranking: https://db-engines.com/en/ranking/graph+dbms
[5]Cypher language overview: https://neo4j.com/docs/getting-started/cypher/
[6]See Neo4j's native graph architecture: https://neo4j.com/docs/getting-started/cypher/
[7]Transaction handling in Neo4j: https://neo4j.com/docs/query-api/current/transactions/
[8]ArangoDB's site: https://arangodb.com/

and `_to` attributes to reference vertex documents, and efficient traversals are enabled through persistent indexing mechanisms.

The database features AQL (ArangoDB Query Language), which is particularly notable for allowing complex operations (including joins, filters, and traversals) across all supported data models in a single query.[9] This flexibility is central to ArangoDB's value proposition, as it simplifies application development and data management for systems requiring diverse data representations, while also supporting horizontal scaling and distributed deployment.[10]

1.3 Interacting with Graph Databases

Although different graph databases notoriously come with different query languages, since 2019, there has been an effort to standardize a query language specifically designed for Knowledge Graphs. This effort led to the publication of ISO/IEC 39075:2024 in April 2024 [1], which defines the Graph Query Language (GQL) standard. As a result, most database manufacturers are now committed to making their proprietary languages GQL-compliant.

GQL inherited most of its features and syntax from Cypher [13], the query language used by Neo4j, and it is defined in terms of five primitive statements:

- `MATCH`, which is the core operator of GQL, as it allows one to look for specific patterns [see Section 1.3.1] in the graph.
- `FILTER`, which has to be used after a `MATCH` clause and plays the same role as `WHERE` in SQL. In fact, the results are filtered according to some specified criteria before being included in the output table. These criteria consist of conditions that use the alias defined in the `MATCH` clause and might leverage built-in functions (e.g., `count`, `max`)
- `LET`, which allows us to bind the result of subqueries to variables.
- `FOR`, which iterates over elements, such as nodes or edges, within a graph. It typically specifies a traversal or a looping construct, allowing operations to be performed on each element that matches the query criteria.
- `ORDER BY` and `LIMIT`.

GQL also defines a standard for create, update, and remove operations and transactions, but we will not provide details about them as they are not relevant to the scope of this thesis.

[9] AQL reference for graph traversals: https://docs.arangodb.com/3.12/aql/graphs/traversals/

[10] See ArangoDB's model flexibility: https://arangodb.com/model-flexibility/

1.3.1 Graph patterns

Node and edge patterns

Both node and edge patterns look for nodes and edges that have a specific label assigned or given specific $(key, value)$ property pairs. We use round and square brackets to respectively indicate a vertex or an edge, semicolons to indicate the name of the labels, while filters on properties are within curly brackets. We can also assign an alias to refer to the matched node or edge elsewhere in the query.

Finally, edge patterns have various syntactic elements to indicate the desired direction. For instance, we use:

- `-[edge_alias :EdgeLabel {property=value}]->` for right-pointing directed edges
- `~[edge_alias :EdgeLabel {property=value}]~` for undirected edges
- `~[edge_alias :EdgeLabel {property=value}]~>` for undirected edges or directed edges pointing right
- `<-[edge_alias :EdgeLabel {property=value}]->` for any directed edge
- `-[edge_alias :EdgeLabel {property=value}]-` for any edge.

Path patterns

Path patterns are more complicated and are defined recursively in terms of node and edge patterns, using the following operations:

- *Concatenation*. Node and edge patterns are combined to form path pattern expressions through concatenation (without any operation sign). These expressions are constructed by alternating node patterns with edge patterns. GQL also addresses the case of consecutive patterns of the same type: consecutive node patterns refer to the same node, whereas consecutive edge patterns imply an implicit node pattern between them.

  For example, the following pattern looks for the customer who purchased a product from a specific store

  ```
  (c:Customer)
  -[:PURCHASED] ->
  -[:SOLD_BY]->
  (:Store {name="TechShop"})
  ```

- *Grouping* allows you to define a pattern using other subpath patterns, putting such subpatterns within curly brackets. It also gives the option to assign aliases or constraints to these subpatterns.
- *Alternation* allows you to define patterns that can match multiple alternative paths, providing flexibility in pattern matching. This is accomplished by specifying different subpaths that the pattern can follow. Alternation is achieved using the operators `|` and `|+|`, which differ in their semantics: `|` employs set semantics, treating matching paths as unique and ignoring duplicates, while `|+|` uses multiset semantics, where duplicates are considered significant.

```
(u1:User)
{ -[r:LIKES]-> | -[r:FOLLOWS]-> }
(u2:User)
```

- *Quantification*. More complex path patterns can be expressed using quantification, which allows us to express variable-length patterns by specifying the minimum and maximum times that an edge or subpath pattern must be matched.

```
(origin :Airport)
-[:FLIGHT]{1,3} ->
(destination :Airport)
```

- *Optionality*. The `?` operator placed after an edge or subpath pattern marks such subpattern as optional within the parent pattern.

When dealing with variable-length paths and graphs that have loops, quantification might lead to an infinite number of paths that all satisfy the given pattern. For this reason, GQL uses selectors and restrictors to specify which paths have to be returned.

Selectors are optional and specify which of the selected paths should be returned. Examples of selectors are `ALL`, `ANY SHORTEST`, and `ALL SHORTEST`. Selectors whose names begin with "ANY" are to be considered non-deterministic.

On the other hand, restrictors are the tools used to avoid selecting an infinite number of paths. The default restrictor is `WALK` that does not put any constraint on the paths that can be selected. We then have `TRAIL` to select paths without repeated edges, `ACYCLIC` to only select paths without repeated nodes, and `SIMPLE`, which returns all the paths without repeated nodes except for the first and last node that can be the same.

1.4 Modelling static graphs

To the extent of this thesis, we are interested in modelling static graphs, namely graphs whose structure changes very rarely and hence can be considered read-only. Although this assumption might sound limiting, these graphs are very common in Business Intelligence applications where we are mainly interested in running OLAP queries. Given the computational intensity of these queries, which often involve navigating sparse regions of the graph, we can leverage the static nature of the data to construct specialized indexing structures. These indexes, while highly effective for read-heavy workloads, would be impractical in dynamic environments where frequent updates are expected. Furthermore, in these static scenarios, the underlying data's infrequency of change allows for the complete reconstruction of graph data structures from scratch. In the following, we will aim for data structures whose reconstruction scales well in time and space, thus making this one-time cost negligible and, in turn, enabling their optimization for the specific query patterns anticipated.

Another instance where the represented data can be assumed to be static is the comprehensive collection of laws within a legislative system, which remains clearly unchanged until new laws are enacted. In this context, Colombo et al. [7] recently demonstrated that its representation through Graph databases effectively captures the hierarchical structure and inter-dependencies of laws, thereby uncovering significant patterns and insights within the legislative corpus.

Furthermore, leveraging graph structures is not limited exclusively to data that is natively stored or primarily modeled as a graph. Specialized graph representations can be constructed on-the-fly through occasional preprocessing steps, even when data resides within relational or columnar databases. For example, expensive join-based queries can be effectively transformed into graph traversals, often leading to substantial performance improvements. Such cases strongly benefit from our assumption of a static dataset.

1.5 Our contribution

In this thesis, we propose and evaluate *computational-friendly* compression techniques designed for static and slightly dynamic LPGs. Our methods achieve significant reductions in storage requirements without substantially penalizing query performance, enabling the efficient implementation of expressive, standards-compliant graph query engines. Specifically, our core contributions include:

1. We introduce a novel approach for efficiently compressing graph connectivity information. This approach supports fast encoding, decoding,

and traversal operations while effectively compressing both sparse and dense graphs. Experimental evaluation demonstrates that our method often achieves better compression ratios and competitive lookup performance compared to other compressed graph representations [**?**].

2. We propose encoding schemes optimized for efficient storage of fixed- or variable-length properties of nodes and edges. By leveraging type-aware compression methods, we significantly reduce the storage overhead of commercial GDBs with minimal computational cost, thus maintaining high property retrieval performance.

3. We provide an extensive experimental comparison against widely used commercial GDBs, as well as (unlabeled) compressed graphs. Our results showcase substantial space occupancy reductions across a diverse selection of real-world heterogeneous datasets while maintaining competitive query times. **PF:** Qualche numero? For example, we achieve up to 000% improvement in space occupancy wrt GDBs, still guaranteeing similar query performance; and similar space occupancy than SOTAs approaches to (unlabeled) graph compressors ... time traversal....

This computation-friendly, lightweight compression design ensures full GQL compliance, allowing ”core” query operations to efficiently execute directly on the compressed representation, thus setting the stage for future development of expressive, compliant, and high-performance GQL-enabled query engines.

These contributions were developed within the scope of a collaborative project involving Sadas[11] and its Graph DBs, in a partnership that transitioned from the University of Pisa to the Sant'Anna School of Advanced Studies, showing additionally its industrial relevance.

1.6 Thesis outline

This thesis is organized as follows. In Chapter 2, we review the fundamental concepts and data structures that underpin our approach, including graph relabelling, compact representations, and succinct index structures. We also survey related work on graph compression techniques, highlighting both classical and recent methods relevant to static graph workloads.

Chapter 3 introduces the datasets used to evaluate our system. We describe six diverse graph collections—ranging from biological and patent citation networks to academic and e-commerce graphs—detailing their size, schema, and structural properties. This heterogeneous corpus enables us to test scalability, compression ratio, and query performance across multiple domains.

[11] https://www.sadas.com/

In Chapter 4, we present our method for representing the graph topology. We formalize the problem of mapping arbitrary user node identifiers to a compact range, and propose two space-efficient mappings based on permutation encodings and FM-Index structures. We then propose a novel approach to compress arbitrary integer sequences and describe how to apply this technique to compress edge adjacency lists after appropriately relabelling graph nodes to take full advantage of this compression scheme.

Chapter 5 covers the representation of node and edge properties. We classify properties into fixed-size types (integers, dates, doubles, enums) and variable-size strings, introduce a node type system to avoid storing sparse nulls, and detail the layout and compression of each property class. Our design balances rapid lookups with overall space reduction.

Chapter 6 reports on benchmarks and experimental evaluation. We measure compression ratios, build times, and query latencies (e.g., neighbors and neighbors of neighbors) on each dataset, comparing against state-of-the-art systems. Results demonstrate that our techniques achieve competitive or superior space-time trade-offs for large static graphs.

Finally, Chapter 7 summarizes our contributions and highlights future research directions.

Chapter 2

Background

In this chapter, we present the tools, concepts, and foundational knowledge necessary to understand the context and motivations for the work described in this thesis.

2.1 Graph relabelling

Graph relabelling is a fundamental problem in Graph Theory. It consists of finding a bijection $\pi : V \to V$ that "relabels" the nodes and edges of a given graph $G = (V, E)$ (either directed or undirected).

2.1.1 Graph bandwidth

The *bandwidth* of G according to π is defined as the maximum difference between the labels of the endpoints of any edge. In other words, it is given by

$$\text{Bandwidth}(G, \pi) = \max_{(u,v)\in E} |\pi(u) - \pi(v)|.$$

Furthermore, the bandwidth of the graph G itself is defined as the minimum bandwidth achievable among all possible labellings π:

$$\text{Bandwidth}(G) = \min_{\pi} \max_{(u,v)\in E} |\pi(u) - \pi(v)|.$$

This problem of finding a relabelling that minimizes the maximum label difference over all edges is known as the graph bandwidth problem, and it is a well-known NP-hard problem [23] in combinatorial optimization and graph theory. Moreover, even finding a relabelling that approximates the minimum bandwidth to within any guaranteed constant factor is NP-hard.

2.1.2 Cuthill-McKee reordering

A commonly used heuristic to tackle the bandwidth minimization problem in practice is the *Cuthill-McKee algorithm* [8]. This algorithm is designed to reduce the bandwidth of the matrix associated with a sparse graph. Although this method does not guarantee that the bandwidth is minimized to the optimal value, it often produces a considerably reduced bandwidth in many practical applications, especially where the graph represents a sparse matrix in numerical simulations or finite element methods.

While Appendix A.1 provides the formal pseudocode, we briefly summarize its key ideas. The algorithm is designed for undirected graphs [1], and begins by constructing a symmetric version of the graph and calculating the degree of each vertex. It then selects a suitable starting vertex based on a global priority determined by vertex degrees.

A breadth-first search (BFS) is then initiated from the selected vertex. During the traversal, each vertex is assigned a new label sequentially. Before enqueueing, the unvisited neighbors of the current vertex are sorted in ascending order of degree, ensuring that vertices with lower connectivity are processed first.

If the graph is disconnected, the algorithm iteratively selects an unvisited vertex using the global degree-based priority and performs BFS until all vertices have been labelled.

2.2 Compact data structures

2.2.1 Bit-packing

Let $\mathcal{S} = s_1, \ldots, s_n$ be a sequence of non-negative integers. A usual representation of $\mathcal{S}$ (e.g., C++ `std::vector`) would allocate either 32 or 64 bits for each s_i. This is often inefficient when the actual values that the integers can take are much smaller, as many of the allocated bits remain unused.

Bit-packing addresses this inefficiency by storing the integers using exactly the number of bits required to represent their maximum value. To achieve this, we explicitly store the number of bits required to represent the largest integer in the sequence. This bit width is then used to compactly encode all the integers we intend to store, eliminating redundant leading zeros.

In this thesis, we make use of the `sdsl::int_vector` data structure from the Succinct Data Structure Library (SDSL) [14] to enable bit-packed storage of integer sequences. This data structure allows specifying the bit-width b used to store each element, and supports efficient access, update, and serialization

[1]It has also been adapted for directed graphs; see [26].

of the packed values. Clearly, such b should be such that each number s_i can be represented using b bits (i.e., $b \geq \lceil \log_2 \max_i s_i + 1 \rceil$).

2.2.2 Rank-Select Bit Vectors

A *rank-select bit vector* [9, Section 15.1] is a data structure used to index static binary sequences while supporting some efficient queries, specified below. Given a 0-indexed binary sequence $\mathcal{B}$ of length n, it supports the following two fundamental operations:

- $\text{rank}_{\mathcal{B}}(i)$, defined as the number of ones in the first i elements of $\mathcal{B}$ (i.e., $\text{rank}_{\mathcal{B}}(i) = \sum_{j=0}^{i-1} \mathcal{B}[j]$).
- $\text{select}^1_{\mathcal{B}}(i)$ returns the position of the i-th occurrence of 1 in $\mathcal{B}$.
- $\text{select}^0_{\mathcal{B}}(i)$ returns the position of the i-th occurrence of 0 in $\mathcal{B}$.

Using auxiliary data structures, both operations can be supported in constant time using $o(n)$ additional bits of space on top of the n bits required to store the original bit vector.

2.2.3 Elias-Fano compressed indexing

Elias-Fano (EF) compressed indexing [29] is a compressed data structure that provides an efficient and compact representation of monotonically non-decreasing integer sequences while allowing constant-time operations.

Given a non-decreasing sequence of non-negative integers $\mathcal{S} = s_0 \leq \cdots \leq s_{n-1}$, we define $u = s_{n-1} + 1$ and initially encode S using $\lceil \log_2 u \rceil$ bits. The $\lceil \log_2(u/n) \rceil$ least significant bits of each integer are stored consecutively in a (bit-packed) vector $\mathcal{L}$. The remaining higher-order bits are represented by encoding the counts of occurrences in unary form within another bit-vector $\mathcal{H}$. Constant time retrieval of the i-th element is then achieved by equipping $\mathcal{H}$ with select and rank support.

The total storage space required by this compressed structure can be estimated as $n(2 + \lceil \log_2(u/n) \rceil) + o(n)$ bits. It can be shown that this space bound is almost optimal if $\mathcal{S}$ is drawn uniformly from [0, u). In such a case, indeed, $\log_2(u/n)$ represents the logarithm of the average gap between two elements s_i and s_{i+1}.

One may think that, given that every sequence $\mathcal{S}$ can be converted into a non-decreasing one through its prefix sum, EF encoding can be applied to every sequence, regardless of its monotonicity. This is obviously true, but encoding non-monotonic sequences with EF might be inconvenient space-wise. As a rule of thumb, EF encoding of a non-monotonic sequence $\mathcal{S}$ is space-convenient compared to its bit-packed representation whenever $\max S \geq 4 \text{avg } S$. We provide the argument of this statement in Section A.2.

2.2.4 Representing (inverse) permutations

A permutation π of $[0, n-1]$ is a sequence where each value $0 \leq v \leq n-1$ appears exactly once.

Given the bijective nature of π, we are interested in supporting both $\pi(i)$ and $\pi^{-1}(i)$ primitives efficiently without storing $\pi^{-}1$ explicitly.

A first observation is that π can be represented using $n\lceil \log_2 n \rceil$ bits thanks to bit-packing. Implementing $\pi(i)$ is straightforward, while $\pi^{-1}(i)$ can be computed by calling $i_0 = i$ and computing $i_k = \pi(i_{k-1}) = \pi^k(i)$ until $i_k = i$.

This procedure has worst-case $O(n)$ time complexity, but it can be generalized to guarantee $O(t)$ time complexity for any $t \geq 1$ with the draw-back of using $n(1 + \frac{2}{t+1}\lceil \log_2 n \rceil) + o(n)$ additional bits [20, Section 5.1].

Specifically, every t steps along permutation cycles of length $l > t$ we store the shortcut $i \to \pi^{-t}(i)$. Thus, computing $\pi^{-1}(i)$ under this generalized scheme now requires taking the first shortcut we encounter while traversing π with the usual scheme.

2.2.5 FM-Index

The Suffix Array (SA) of a given text $\mathcal{T}$ of length n is an array of integers specifying the starting positions of all suffixes of $\mathcal{T}$ sorted in lexicographical order. This structure enables efficient pattern matching. For instance, finding all occurrences of a pattern P of length m can be achieved by binary searching the SA, typically requiring $O(m \log_2 n)$ time to identify the range of suffixes prefixed by P, followed by constant time retrieval for each occurrence's starting position [9, Section 10.2].

The SDSL library [14] offers a compressed implementation of SA (CSA) based on Huffman-shaped Wavelet-Trees [10] constructed over the Burrows-Wheeler Transform [19] of the input text, whose design is remarkably inspired by the FM-Index [11].

This specific implementation uses explicit sampling of the Suffix Array and Inverse Suffix Array (respectively, t_{dens} and t_{inv_dens}) to balance space and query time. The primitives offered are the same offered by FM-Indexes, namely:

- $\text{count}_{\mathcal{T}}(P)$, which returns the number of occurrences of a pattern P within the text $\mathcal{T}$. It has $O(m \log_2 \sigma)$ time complexity where σ is the alphabet size.
- $\text{locate}_{\mathcal{T}}(P)$, which returns the positions of all the occurrences of the pattern P within $\mathcal{T}$. In addition to the work of $\text{count}(P)$, each reported occurrence requires "walking" from the wavelet structure back to a sampled SA entry; this part is governed by the sampling interval t_{sample}.

- extract$_{\mathcal{T}}(P, start, end)$, which retrieves the text substring $T[start, end]$. The time complexity grows with the substring length and again depends on how frequently we store samples of the inverse SA (parameter t_{inv_sample}).

At a high level, the total space is close to the zero-order entropy of $\mathcal{T}$ (i.e $nH_0(L)$ bits, where L is the last column of the BWT of $\mathcal{T}$), plus a modest overhead for the wavelet tree, plus the bits needed for the SA and inverse-SA samples.

2.3 On graph compression

In this section, we review techniques used to compactly represent either general graphs or LPGs. Given the scope of this thesis, we narrow our focus to representations that offer fast decompression speed.

2.3.1 Compressed Sparse Rows representation

A common approach when representing sparse graphs (either directed or undirected) is to use the *Compressed Sparse Row* (CSR) format. Given a graph $G = (V, E)$ and calling $n = |V|$ and $m = |E|$, its CSR representation is given by $(\mathcal{A}, \mathcal{O})$ where:

- $\mathcal{A}$ is an array of size m containing the concatenated neighbor lists of each vertex
- $\mathcal{O}$ is an array of $n + 1$ offsets, where $\mathcal{O}[i]$ is the sum of the out-degree of the first $i - 1$ nodes for any $1 \leq i \leq m$ and 0 for $i = 0$.

Hence, the neighbors of the i-th vertex can be found in the range $[\mathcal{O}[i] : \mathcal{O}[i + 1]]$ within $\mathcal{A}$.

This representation is clearly equivalent to the corresponding adjacency lists, but it offers the advantage of storing pointers to lists only when a node has outgoing edges. More importantly, it eliminates the overhead of maintaining a list (and therefore its header) for each neighborhood.

Usually, the CSR framework is implemented by representing $\mathcal{A}$ and $\mathcal{O}$ as standard arrays (e.g., using C++ `std::vector`). However, we can easily observe that $\mathcal{A}$ and $\mathcal{O}$ can be easily compressed. Indeed, $\mathcal{A}$ can be represented with a bit-packed array since it is made up of vertices' ids, which are within the range $[0, n - 1]$. It is also easy to observe that $\mathcal{O}$ is a non-decreasing sequence, hence can be coded using the EF encoding. We thus conclude the following theorem.

Theorem 2.1 *The CSR representation of a graph $G = (V, E)$ with n nodes and m edges, takes*

$$m\lceil\log_2 n\rceil + n\left(2 + \left\lceil\log_2 \frac{m+1}{n}\right\rceil\right) + o(n) \tag{2.1}$$

bits.

For the rest of this thesis, with CSR we will refer to this implementation that employs bit-packing and EF.

2.3.2 LogGraph

Besta et al. [4] recently proposed a compressed representation of graph adjacency lists with low-overhead decompression.

The core idea behind *LogGraph* is to sort the adjacency list of the nodes before concatenating them when forming $\mathcal{A}$ and exploiting this ordering to store the gaps. Such gaps are stored used Varint encoding [2], a variable-length encoding that uses fewer bits for smaller numerical values. In this variant, however, $\mathcal{O}[i]$ is not the sum of the out-degree of the $i-1$ nodes anymore but the bit position where the i-th adjacency list begins in $\mathcal{A}$.

This definition of $\mathcal{O}$, which comes with a bigger max-value, obviously has an intrinsic overhead. Moreover, gap encoding alone does not guarantee significant compression improvements without further strategies.

To boost compression, LogGraph extensively uses various vertex relabelling techniques. Among the approaches proposed, their Degree-Minimizing (DM) schemes relabel nodes according to their degrees, such that vertices with higher degrees receive a lower ID value. The authors strongly position the DMd variant (Degree-Minimizing with differences encoded, often using a Varint-like approach) as offering the best space/performance tradeoff among the techniques they developed for logarithmizing the adjacency structure $\mathcal{A}$. They also claim that DMd achieves compression ratios comparable to WebGraph [5].

2.3.3 CGraphIndex

Huo et al. [18] proposed a self-index for knowledge graphs, supporting efficient queries.

Similarly to LogGraph, *CGraphIndex* represents the topology of G via sorted adjacency lists. The core topological structure, `gStruct`, consists of:

- $\mathcal{S}$ is a hybrid-encoded sequence obtained by concatenating, for each vertex u in increasing order, either its Elias-Fano (EF) encoding or a

[2]See https://protobuf.dev/programming-guides/encoding/

fixed-length (F) code of its sorted adjacency list (or its gap sequence for F code), whichever is shorter. When using Elias-Fano encoding, the cost is approximately $\lfloor \log_2(n/\text{outdegree}(u)) \rfloor + 3$ bits per element on average.

- $\mathcal{M}$ is a bit-array of length $|V|$ marking, for each u, which encoding method (EF or F) was used for its adjacency list Adj_u ; this dictates how to decode Adj_u from $\mathcal{S}$ on the fly.
- $\mathcal{B}_1$ is a bit-array of length $|\mathcal{S}|$ with a 1 at each position where a new vertex's encoded adjacency list begins in $\mathcal{S}$ (and 0 elsewhere). Combined with $\mathcal{B}_3$, $\text{select}_1(\mathcal{B}_1, \text{rank}_1(\mathcal{B}_3, u))$ helps locate the start of vertex u's block in $\mathcal{S}$.
- $\mathcal{B}_2$ is a bit-array of length $|E|$ with a 1 at each index corresponding to the start of a vertex's adjacency list in the conceptual global edge sequence Adj . This helps determine the number of neighbours (degree) for a vertex u using ranks on $\mathcal{B}_3$.
- $\mathcal{B}_3$ is a bit-array of length $|V|$ where $\mathcal{B}_3[u] = 1$ if Adj_u is non-empty, and 0 otherwise . It is used to map a vertex u to its ordinal among vertices with non-empty lists for indexing into $\mathcal{B}_1$ and $\mathcal{B}_2$.

An entirely analogous set of arrays $(\mathcal{S}', \mathcal{M}', \mathcal{B}'_1, \mathcal{B}'_2, \mathcal{B}'_3)$ (termed `gInStruct`) is built over the incoming adjacency lists to support in-neighbor queries in the same way .

The bit-arrays $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$ are stored in compressed form using RRR-style [24] indexable dictionaries, which for a bit-vector of length l with k set bits occupy $\log_2 \binom{l}{k} + o(k)$ bits . The method array $\mathcal{M}$ is stored as a plain bit-array using $|V|$ bits .

Contrarily to LogGraph, *CGraphIndex* assigns a numeric vertex identifier (*Vid*) based on the vertex label and its original ID within that label's class: first, a minimal perfect hash $\text{MPhash}(x, \text{ID})$ ranks nodes within each label class x; then, a class offset $H[h(x)]$ is added, so the final $Vid = \text{MPhash}(x, \text{ID}) + H[h(x)]$. This gives a dense, label-stratified numbering in $[1 \ldots n]$, computable in $O(1)$ time using $O(n)$ bits. The paper uses MPhash on the vertex ID within its class.

Finally, vertex and edge properties are then handled as follows:

- *Vertex properties:* All vertex property strings are concatenated into a single text $\mathcal{T}_V$, indexed with a self-index $\mathcal{V}Index$ (e.g., FM-index style) and a bit-vector $\mathcal{V}B$. Locating and retrieving the property of a vertex *Vid* reduces to one $select_1$ on $\mathcal{V}B$ followed by a substring extract from $\mathcal{V}Index$.

- *Edge residual properties:* All residual edge property strings are concatenated into $\mathcal{T}_R$, indexed with $\mathcal{R}Index$ and a bit-vector $\mathcal{R}B$, supporting retrieval analogously via operations like *getPosR* and *ExtractR* .
- *Edge relation types:* The (constant-sized) set of distinct edge relation labels is encoded in a compact array $\mathcal{E}_r$, where each edge's relation type maps to a fixed-length codeword based on its *Eid* . Thus, retrieving the relation property, $eRelation(Eid)$, is an $O(1)$ lookup .

Chapter 3

Datasets

In this chapter, we present the research datasets used for testing the GraphDB. For each dataset, we describe the node and edge attributes, report the number of nodes and edges, and provide some basic information and statistics.

Table 3.1: Dataset Statistics. Node and edge counts have been rounded for readability. Property counts indicate the number of distinct property keys across all nodes or edges.

Dataset	# Nodes	# Edges	Type	# Node Props	# Edge Props
Stark-Prime	130k	4M	Undirected	5	4
DBLP	15M	35M	Directed	9	3
Patents	3M	26M	Directed	23	3
Stark-Amazon	1M	4M	Directed	10	3
Stark-Mag	2M	20M	Directed	15	3

3.1 Stark-Prime: graph of biological entities

Stark-Prime is a biological dataset that integrates various types of biological entities, such as genes, proteins, diseases, and medications. The dataset compiles information extracted from scientific publications, non-standard repositories, and established biological ontologies [6]. Although the number of nodes is relatively low, the graph exhibits high density, as indicated by an elevated average node degree. This dataset is part of the Stanford Stark SKB project[1] [31].

The processed dataset is available as two files, `nodes.tsv` and `edges.tsv`, which contain all the nodes and edges, respectively. [2]

[1] https://stark.stanford.edu/dataset_prime.html

[2] The processed dataset can be downloaded from https://drive.google.com/file/d/1nw4e5ZWXNXx7NKBbCjrsH-SdvsuGpztW/view?usp=sharing.

The graph comprises approximately 130,000 nodes and 4 million *undirected* edges. When compressed in the ZIP file, the total size is 28 MB. In uncompressed form, the `nodes.tsv` file requires around 7.6 MB, while the `edges.tsv` file requires approximately 140 MB.

Node Properties

The nodes in the graph include the following attributes:

- *ID:* A unique integer identifier.
- *Type:* The classification of the biological entity (e.g., gene/protein, disease).
- *Name:* The designated name of the entity, as determined by the corresponding reference ontology.
- *Source:* The reference ontology (e.g., `NCBI` stands for National Center for Biotechnology Information).
- *OntoID:* Identifier of this biological entity in the reference ontology, specified in the `Source` column.

Edge Properties

Each edge in the graph is characterized by the following properties:

- *Source ID:* The identifier of the source node, corresponding to the `ID` column in the nodes file.
- *Target ID:* The identifier of the target node, also referring to the `ID` column in the nodes file.
- *Type:* A short abbreviation denoting the type of interaction between the two entities (e.g., `PPI` for protein-protein interaction).
- *Relation Type:* A more descriptive version of the string provided in the `Type` field.

3.2 DBLP: academic citation graph of scientific articles

The dataset is extracted from DBLP—a comprehensive bibliography of scientific articles in computer science—as well as other sources, including ACM (Association for Computing Machinery) and MAG (Microsoft Academic Graph), among others [28]. From the various provided versions, we employed the most recent one (dump v14) dated January 31, 2023. This dataset contains nodes representing articles, authors, affiliations, and venues, establishing relationships among them.

The processed dataset is available in two files (nodes.tsv and edges.tsv), which include all the nodes and edges.[3] Our dataset was derived from dump v14, available at https://www.aminer.cn/citation.

The graph counts about 15.6M nodes and 35M edges. The nodes.tsv and edges.tsv files, if compressed into a ZIP file, occupy 903 MB. Uncompressed, the nodes and arcs files take 1.5 GB and 1.7 GB, respectively.

Nodes

Each node in the graph is characterized by the following attributes:

- *id*: A unique string identifier for the node.
- *type*: The type of the node (i.e., article, author, venue, affiliation).
- *label*: Depending on the node type, it specifies:
  - For an article: the article title.
  - For an author: the author's full name.
  - For an affiliation: the organization's name.
  - For a venue: the name of the conference or journal.
- *issn*: Applicable only to articles; indicates the ISSN corresponding to the serial publication title.
- *isbn*: Applicable only to articles; provides the 13-digit ISBN for identifying the "book" of the publication.
- *n_citation*: Applicable only to articles; denotes the number of citations.
- *year*: Applicable to both articles and venues; for articles, it represents the year of publication, while for venues, it indicates the year in which the conference took place.
- *doi*: Applicable only to articles; it offers the URL that uniquely identifies the article.
- *doc_type*: Applicable only to articles; specifies the type of article (e.g., journal, conference paper).

Edges

Edges within the graph are described by the following attributes:

- *source*: The ID of the source node.

[3]The processed dataset can be downloaded from https://drive.google.com/file/d/1ACqpewlYwdvCpxvWxcBhD7eyCa5WonGu/view?usp=sharing.

- *type*: The nature of the relationship, which depends on the involved node types (e.g., an author *WROTE* an article).
- *target*: The ID of the target node.

3.3 Patents: american patent citation graph

This section describes part of a project conducted by the National Bureau of Economic Research (NBER) [15] on American patent data from 1976 to 2006. The project encompasses details on over 3 million patents, including more than 16 million citations among them[4].

In this graph, the nodes represent either patents or assignees. Patent nodes are interconnected by edges of type `CITES`. Additionally, corresponding relationships link patents to their assignees—entities, companies, or individuals.[5]

The entire dataset occupies 233 MB of storage when compressed in a ZIP archive, containing approximately 3.5 million nodes and 26.4 million edges. In its uncompressed form, the nodes file and the edges file require approximately 338 MB and 905 MB, respectively.

All information about categories and subcategories is based on the classification provided by the Commerce Control List (CCL).

Node Properties

(Almost all properties are set only for nodes of type `patent` unless explicitly stated otherwise.)

- *id*: Unique identifier for each node.
- *type*: Indicates whether the node is a `Patent` or an `Assignee`.
- *name*: (For nodes of type `Assignee`) The name of the assignee.
- *Cat*: Technological category according to the CCL, ranging from 1 to 6.
- *Cclass*: United States Patent Classification (USPC) code that categorizes the patent.
- *Status*: Authorization status of the patent (where `m` indicates missing and `w` indicates withdrawn).
- *Subcat*: CCL-related subcategory (values between 11 and 69).
- *Subclass*: Subclass for the given USPC class (as defined by the CCL).

[4]The raw data is available at https://sites.google.com/site/patentdataproject/Home/downloads/patn-data-description?authuser=0

[5]The processed dataset can be downloaded from https://drive.google.com/file/d/1mCov-nqQCOBE-gGBf7NXq4YRRG9PEnDh/view?usp=sharing.

- *subclass1*: Similar to `Subclass`, but referring to the alpha-subclasses.
- *icl*: International classification.
- *icl_class*: A 4-character code specifying the class according to the International Patent Classification (IPC).
- *icl_maingroup*: The main group containing the IPC class.
- *Gday*: Day on which the patent was granted.
- *Gmonth*: Month in which the patent was granted.
- *Gyear*: Year in which the patent was granted.
- *Appyear*: Year in which the patent was submitted.
- *Nclaims*: Number of claims in the patent.
- *Iclnum*: Sequence number associated with the IPC class.
- *Nclass*: A three-digit number specifying the patent class according to the CCL.
- *term_extension*: Number of days the patent term was extended.
- *allcites*: Total number of citations received by the patent up to 2006. Note that this number is imprecise for patents with submission years between 2006 and 2009; a secondary field, `hjtwt`, is provided to correct this inaccuracy.
- *hjtwt*: Correction value for `allcites`.
- *uspto_assignee*: (For nodes of type `Assignee`) The original identification number of the assignee as provided by the United States Patent and Trademark Office (USPTO).

Edge Properties

- *source*: The source node.
- *target*: The destination node.
- *type*: Indicates the type of relationship, e.g., `ASSIGNED_TO` or `CITES`.

3.4 Stark-Amazon

The dataset consists of a product recommendation graph for an online catalog (Amazon) (3). It comprises a list of products along with associated metadata, review details, and relationships such as *also purchased* and *also viewed*. This dataset [6]was extracted from the SKB (Semi-structured Knowledge Base) of the

[6]The processed dataset can be downloaded from https://drive.google.com/file/d/13A9DCImmOZPFI6TyHZRUpaZcMQb3RqOV/view?usp=sharing.

Stark project [31], which provides three SKBs, one of which is the Amazon catalogue available at https://stark.stanford.edu/dataset_amazon.html.

The graph is archived in a ZIP file with a total size of 208 MB and contains slightly fewer than 1 million nodes and 4 million edges. In its uncompressed state, the nodes and edges files require 663 MB and 94.9 MB, respectively.

Nodes represent either products or categories. The edges connect products to their respective categories and further link products together based on the following user interactions:

- Users who have viewed both product pages.
- Users who have purchased both products.

Node Properties (all defined for products, with only a subset applicable to categories):

- *ID*: Unique identifier.
- *Type*: Specifies whether the node represents a product or a category.
- *Asin*: An alternative identifier, valid only for products.
- *Title*: The name of the product in the Amazon catalogue, or the name of the category if the node is of type category.
- *Review_score*: The average review rating available on the website (a float ranging from 1 to 5).
- *Price*: The product price in dollars, or 0 if unspecified.
- *Brand*: The product's brand.
- *Feature*: A list of tags associated with the product on the Amazon catalogue.
- *Rank*: A string that specifies the ranking of the product within its corresponding category.
- *Description*: A text description providing additional details about the product.

Edge Properties:

- *Source*: The identifier of the source node.
- *Type*: The type of relationship (can be ALSO_VIEWED, ALSO_BOUGHT, HAS_CATEGORY).
- *Target*: The identifier of the target node.

3.5 Stark-MAG: academic citation graph

The following graph is another academic graph, very similar to the previously described "DBLP" academic graph. We decided to include this dataset to enable a comparison between two datasets of different scales while maintaining a similar network skeleton. This dataset was entirely constructed from a snapshot of the Microsoft Academic Graph (MAG) project (now discontinued) [30, 27]. It is a graph of scientific articles that are connected with their authors and the venues (either journals or conferences) where they were presented. The processed graph was built, analogously to the Prime graph (the biological graph) and the Amazon dataset, using data provided by Stark [31] by extracting the files related to the Semi-structured Knowledge Base (SKB)[7].

The compressed graph, contained in a ZIP archive, occupies 389 MB and consists of approximately 1.9 million nodes and 19.9 million edges. When uncompressed, the node and edge files occupy 864 MB and 852 MB, respectively.[8]

Node Properties

- *Id:* Unique identifier for each node.
- *Type:* Indicates whether the node represents an author, institution, field of study, or paper.
- *mag_id:* Identifier of the node within the Microsoft Academic Graph (MAG).
- *name:* For an author, this field contains the first and last name; for an institution, it contains the institution's name; for a field of study, it contains a descriptive string; and for a paper, it contains the paper's title.
- *PaperCount:* Applicable to all nodes except those of type "paper". This field contains the number of relevant papers associated with the author, topic, or institution.
- *CitationCount:* Similar to PaperCount, this field refers to the total number of citations that point to that node (e.g., Author X has received a total of Y citations).
- *DocType:* For paper nodes, this indicates whether the paper was published in a journal or presented at a conference.

[7] https://stark.stanford.edu/dataset_mag.html

[8] The processed dataset can be downloaded from https://drive.google.com/file/d/1mzBjgFJZs7ZctAQzkr9-S8Wkwc4WjEXZ/view?usp=sharing.

- *Year:* For paper nodes, this specifies the publication year.
- *Date:* For paper nodes, this specifies the exact publication date.
- *Publisher:* For paper nodes, this field indicates the publisher of the journal.
- *JournalId:* For paper nodes published in journals, this serves as the journal's identifier.
- *ConferenceSeriesId:* For paper nodes presented at conferences, this field identifies the conference series.
- *ConferenceInstanceId:* For paper nodes presented at conferences, this field identifies the specific annual instance of the conference.
- *PaperCitationCount:* For paper nodes, this field specifies the number of citations received by the paper.
- *Abstract:* For paper nodes, this field contains the abstract of the article.

Edge Properties

- *source:* Identifier of the source node.
- *target:* Identifier of the target node.
- *type:* The types of relationships include:
  - author_affiliated_with_institution,
  - paper_cites_paper,
  - paper_has_topic_field_of_study,
  - author_writes_paper.

Chapter 4

Representing the graph topology

In this chapter, we propose a novel space-efficient approach to representing the graph topology.

Source datasets typically identify nodes using unique identifiers (e.g., primary keys), which we term *User Node IDs*. Let V_U be the set of such User Node IDs. These identifiers may be strings or integers and generally do not fall within a predictable, compact range (like $\{0, \ldots, n-1\}$). Directly using these User Node IDs within graph data structures can be inefficient for storage and computation.

Therefore, we need to assign *Internal Node IDs* to nodes. Let V_I be the set of these Internal Node IDs. We aim for V_I to be a compact set, specifically the fixed range $\{0, \ldots, n-1\}$. We introduce a mapping $\mathrm{Id}_V : V_U \to V_I$ from these arbitrary User Node IDs to a compact set of Internal Node IDs.

While edges do not necessarily have a user identifier assigned, we still assign an Internal ID. Let E_I be the set of these Internal Edge IDs, explicitly $E_I = \{0, \ldots, m-1\}$.

Problem: We need to provide our graph database with a map $\mathrm{Id}_V : V_U \to V_I$. Such a map should be bidirectional, allowing efficient computation of both Id_V and Id_V^{-1}. The graph database should also be able to assign each edge $e \in E$ a numeric identifier $\mathrm{Id}_E(e) \in E_I$.

Once our graph database is equipped with such mappings, we can revisit Definition 1.1 in terms of these internal IDs.

Definition 4.1 *A Labeled Property Graph (LPG) is a tuple*

$$G = (V_I, E, L, \mathrm{l}_V, \mathrm{l}_E, K, W, \mathrm{p}_V, \mathrm{p}_E),$$

where:

- $V_I = \{0, \ldots, n-1\}$ *is the set of internal node IDs.*
- $E \subseteq V \times V$ *is a multiset of directed edges.*
- L *is the set of labels.*
- K *is the set of property keys.*
- W *is the set of property values.*
- $l_V : V_I \to L \cup \{\varepsilon\}$ *assigns an optional label to each node.*
- $l_E : E_I \to L \cup \{\varepsilon\}$ *assigns an optional label to each edge.*
- $p_V : V_I \times K \to W \cup \{\varepsilon\}$ *assigns properties to nodes, if defined.*
- $p_E : E_I \times K \to W \cup \{\varepsilon\}$ *assigns properties to edges ids, if defined.*

4.1 Representing Node IDs

Constructing an appropriate Id_V is equivalent to determining a suitable graph relabelling. While we will tackle that task in the following section, our current focus is on introducing two space-efficient data structures for storing Id_V and its inverse Id_V^{-1}. A trivial approach would be to keep two hash maps Id_V and Id_V^{-1}, but this would lead to a significant storage overhead.

4.1.1 Permutation-based mapping

We propose a compact data structure for mapping between User Node IDs and $V_I = \{0, \ldots, n-1\}$. It supports both integer and string keys, but offers better performance and compression ratios when keys are integers.

We call $\mathcal{K}eys$ the zero-indexed sorted array such that $\mathcal{K}eys[i]$ is the i-th User Node ID according to some total ordering over V_U (e.g., the canonical ordering). The $\mathcal{K}eys$ array acts as an intermediate mapping from a $userID \in V_U$ to its index i in the sorted sequence. We then define a permutation π on V_I, i.e, the indices $\{0, \ldots, n-1\}$, such that $userID = \mathcal{K}eys[i] \implies \pi^{-1}(i) = internalID$. Such π can be constructed as an invertible permutation as described in Section 2.2.4 using $t = \lceil \log_2 n \rceil$ where $n = \#\mathcal{K}eys$.

This setup allows us to compute both $\mathrm{Id}_V(userID)$ and its inverse Id_V^{-1} (given the corresponding *internalID*) efficiently:

- To compute $\mathrm{Id}_V(userID)$ we first perform a binary search over $\mathcal{K}eys$ and find the index i such that $\mathcal{K}eys[i] = userID$. We then map such an index to the corresponding *internalID* through π^{-1}.
- $\mathrm{Id}_V^{-1}(internalID)$ is computed as follows. We compute $i = \pi(internalID)$ and then we get its corresponding key by accessing $\mathcal{K}eys[i]$.

Clearly, this scheme supports both integer and string-based User Node IDs. However, if User Node IDs are integers, the $\mathcal{K}eys$ array can be compressed using Elias–Fano encoding to save space. On the other hand, when User Node IDs in V_U are strings, the scheme remains valid, with the only overhead being the cost of string comparisons during binary search in Id_V.

Theorem 4.2 *If V_U consists of n non-negative integers bounded by u, the proposed data structure serves Id_V in $O(\log n)$ time and Id_V^{-1} in constant-time, while taking at most $n(7 + \log_2 u) + o(n)$ total bits of space.*

Proof Id_V requires performing a binary search over $Keys$, which takes $O(\log_2 n)$ time, and computing the inverse of the index returned by the binary search through π^{-1}, which takes $O(t) = O(\log_2 n)$ time. The time complexity of Id_V^{-1} is given by accessing first π and then the i-th key, both constant-time operations.

Finally, the space taken by the map is the sum of the space taken by $\mathcal{K}\rceil\dagger\int$ and π. If encoded with Elias-Fano, $\mathcal{K}eys$ requires at most $n(2 + \lceil\log_2 u/n\rceil) + o(n)$ bits, while π requires $n\lceil\log_2 n\rceil + n + n\frac{2}{t+1}\lceil\log_2 n\rceil + o(n)$ bits. The sum of these 2 quantities can then be rewritten as follows

$$\begin{aligned}
&n(2 + \lceil\log_2 u/n\rceil) + o(n) + n\lceil\log_2 n\rceil + n + n\frac{2}{t+1}\lceil\log_2 n\rceil + o(n)\\
&= 3n + n\lceil\log_2(u/n)\rceil + n\lceil\log_2 n\rceil + n\frac{2}{\lceil\log_2 n\rceil + 1}\lceil\log_2 n\rceil + o(n)\\
&\leq 3n + n\lceil\log_2(u/n)\rceil + n\lceil\log_2 n\rceil + 2n + o(n)\\
&= n(5 + \lceil\log_2(u/n)\rceil + \lceil\log_2 n\rceil) + o(n)\\
&\leq n(5 + \log_2 u - \log_2 n + 1 + \log_2 n + 1) + o(n)\\
&= n(7 + \log_2 u) + o(n) \qquad \square
\end{aligned}$$

Alternatively, a similar result could be achieved by using two Minimal Ordered Perfect Hash (MOPH) functions [9, Section 8.6], with negligible additional space overhead. Such an approach would yield an optimal $O(1)$ time complexity for computing both Id_V and its inverse Id_V^{-1}. We opted instead for the permutation-based method due to its implementation simplicity. Moreover, this approach already provides sufficient performance guarantees. While the $O(\log n)$ complexity for computing Id_V is suboptimal, in practice the overhead is minimal since we query Id_V infrequently—for example, only at the beginning of a traversal such as a BFS from a given node. On the other hand, computing Id_V^{-1} is a more frequent operation (e.g., to translate Internal Node IDs back into User IDs throughout the visit) and is conveniently carried out in optimal $O(1)$ time by our proposed scheme.

4.1.2 FM-Index-based mapping

We now propose another approach that aggressively compresses the underlying data, with the drawback of having more expensive retrievals.

We consider the case where the User Node IDs are strings. This is not limiting since integer keys can be easily encoded as strings as well. Let then ξ be a delimiter such that $\xi \notin \Sigma$ where Σ is the alphabet in which V_U elements are drawn.

Let key_i denote the User Node ID such that $\mathrm{Id}_V(key_i) = i$. We define the string $\bar{\mathcal{T}}$ as follows:

$$\bar{\mathcal{T}} = \xi \cdot key_0 \cdot \xi \cdot key_1 \cdot \xi \cdots \xi \cdot key_{n-1} \cdot \xi$$

The structure $\mathcal{T}$ is then the representation of $\bar{\mathcal{T}}$ using a CSA based on Huffman-shaped Wavelet Trees.

Finally, we build a (sparse) binary-vector $\mathcal{B}$ such that $\mathcal{B}[i] = 1 \iff \bar{\mathcal{T}}[i] = \xi$.

Using $\mathcal{T}$ and $\mathcal{B}$, we can implement Id_V and Id_V^{-1} as follows:

$$\mathrm{Id}_V(userID) = \mathrm{rank}_{\mathcal{B}}\left(\mathrm{locate}_{\mathcal{T}}\left(\xi \cdot userID \cdot \xi\right) + 1\right) - 1$$

$$\mathrm{Id}_V^{-1}(internalID) = \mathrm{extract}_{\mathcal{T}}\Big(\begin{array}{l} \mathrm{select}^1_{\mathcal{B}}(internalID + 1) + 1, \\ \mathrm{select}^1_{\mathcal{B}}(internalID + 2) \end{array} \Big)$$

The main advantage of this approach lies in its compactness: both $\mathcal{T}$ and $\mathcal{B}$ can be represented succinctly, with entropy-compressed representations delivering space close to the information-theoretic lower bound. However, the cost of these compact representations is that querying is more expensive compared to the permutation-based implementation, especially for Id_V, which requires a full pattern search.

4.2 Compressing the graph structure

The preceding section has laid the groundwork necessary for efficiently representing Node IDs. This section now focuses on tackling the related challenge of effectively representing the adjacency lists that encode graph topology. Precisely, we need a data structure that compactly stores neighborhoods of nodes (adjacency information), and we must identify a suitable scheme for assigning internal Node IDs in a manner conducive to succinct representation.

When designing such a mechanism, three essential considerations arise:

1. The adjacency lists must allow constant-time query operations, i.e., given a node v and an integer index i, access to the i-th neighbor $N_v(i)$ should be supported in $O(1)$ time. We might also be interested in supporting efficient backwards queries (i.e., getting the parts of a given node).
2. The solution must achieve good compression regardless of the nature of the graph. Indeed, GDBs are not specifically designed for one single domain (e.g., social networks), so we aim to guarantee good compression rates on both sparse and dense graphs.
3. The node relabeling strategy should be carefully chosen to optimize compression. Intuitively, assigning adjacent nodes close internal IDs often results in more-compressible adjacency sequences.

4.2.1 Limitations of current approaches

Figure 4.1 illustrates the compression rates of existing solutions presented in Section 2.3 on the topology of the test datasets described in Chapter 3.

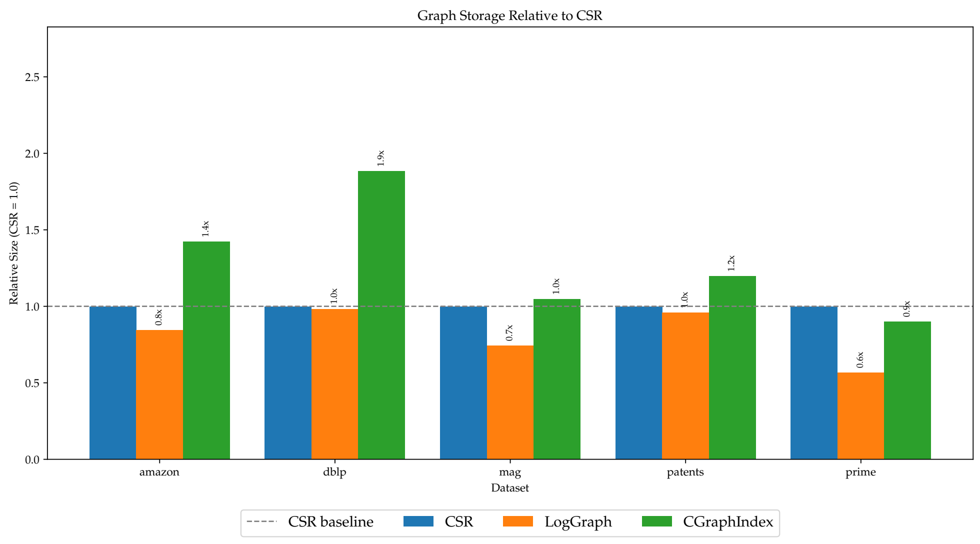

Figure 4.1: Comparison of storage sizes across different graph representations (CSR [Compressed Sparse Row], LogGraph, and CGraphIndex), normalized to the CSR size for each dataset (CSR = 1.0 baseline).

We observe that the compression performance of CGraphIndex is consistently worse than LogGraph, whereas LogGraph achieves similar or better performance than CSR when the graphs exhibit dense connectivity. However, given the simplicity of CSR's design and the fact that most labeled graphs in GDBs applications are sparse, we believe there is room in achieving a solution that is easy to implement and effective in space/time performance.

Driven by these insights, we take the CSR scheme as our baseline and introduce further optimizations, leveraging assumptions we can make on the underlying data. Concretely, we focus on the compression of the concatenated neighborhoods $\mathcal{A}$. Without loss of generality, we assume that each neighborhood N_v is sorted in increasing order within $\mathcal{A}$, and we propose a technique to compress it efficiently, exploiting possible clusters within. Actually, such a technique can be applied to any integer sequence and, in the case of $\mathcal{A}$, if each neighborhood N_v is compact (i.e., $\max N_v - \min N_v$ is small), then the neighborhoods form clusters.

Finally, we argue that an appropriate graph relabelling strategy can enhance the formation of clusters in $\mathcal{A}$, thus maximizing the effectiveness of our approach.

4.2.2 A cluster-aware integer compression technique

Let $\mathcal{S}$ be an arbitrary sequence of l non-negative integers bounded by $u > 0$, and $b = \lceil \log_2 u \rceil$ the minimum number of bits necessary to represent every element in $\mathcal{S}$.

Our compression scheme is based on the observation that clustered sequences of integers frequently share common prefixes in their binary representation. We exploit this fact by splitting each integer into two parts on a chosen *split point* $0 \leq h < b$. We denote the h most significant bits with s_i^+, and the remaining $b - h$ bits with s_i^-. Additionally, we maintain a bit-vector $\mathcal{B}$ of length l recording whether consecutive elements share identical s_i^+ values. Formally, the bit-vector $\mathcal{B}$ is defined as:

$$\mathcal{B}[i] = \begin{cases} 1 & i = 0 \vee s_{i-1}^+ \neq s_i^+ \\ 0 & \text{otherwise} \end{cases} .$$

Intuitively, a bit set to 1 indicates the start of a new cluster in the high bits sequence. Consequently, we only explicitly store high-bit values corresponding to positions with a 1-bit in $\mathcal{B}$. Thus, the high sequence $\mathcal{H}$ stores exactly as many integers as there are ones in $\mathcal{B}$, while the low bits $\mathcal{L}$ always store low-bit values for all l integers.

Lemma 4.3 *If $\mathcal{B}$ is equipped with a rank-support data structure, we can implement Access$(\mathcal{S}, i)$ to reconstruct s_i from $\mathcal{L}$, $\mathcal{H}$, and $\mathcal{B}$ in $O(1)$ time.*

Proof We define the access function as:

$$\text{Access}(\mathcal{S}, i) = \mathcal{H}[\text{rank}_{\mathcal{B}}(i+1) - 1] \times 2^h + \mathcal{L}[i].$$

Clearly, this expression can be evaluated in $O(1)$ time assuming that $\mathcal{B}$ supports rank queries in constant time.

To establish correctness, we need to demonstrate two properties: (i) $\mathcal{L}[i] = s_i^-$, and (ii) $\mathcal{H}[\text{rank}_\mathcal{B}(i+1) - 1] = s_i^+$. The first property follows immediately from the construction of the array $\mathcal{L}$, which explicitly stores the lower h bits of each element. The second property can be proven using mathematical induction, as it reflects the mapping of positions in $\mathcal{B}$ to entries in $\mathcal{H}$. □

We can also observe that for the special case $h = 0$, the proposed scheme is fully equivalent to bitpacking, therefore, we can avoid storing $\mathcal{B}$ in such a case. We summarize these results about the space occupancy in the following theorem.

Theorem 4.4 *For any split point h, the number of bits required by the proposed encoding scheme is*

$$\text{Space}(\mathcal{S}, h) = \begin{cases} rank_\mathcal{B}(l) \times h + l \times (\lceil \log_2 u \rceil - h) + l + g(l) & h > 0 \\ l \times \lceil \log_2 u \rceil & h = 0 \end{cases}$$

where $g(l)$ is typically a $o(l)$ function due to the $\mathcal{B}$'s rank support overhead.[1]

Such a quantity can also be computed in linear time without explicitly constructing $\mathcal{H}, \mathcal{L}$, and $\mathcal{B}$. Indeed, we just have to count the number of 1s we would have in $\mathcal{B}$ which can be done by traversing $\mathcal{S}$.

We are now interested in choosing the split point h that minimizes the overall space usage of the encoding.

Definition 4.5 *We define the optimal split point h^* of a sequence $\mathcal{S}$ the quantity*

$$h^*(\mathcal{S}) = argmin_{0 \leq h < b} \text{Space}(\mathcal{S}, h).$$

Clearly, we are always interested in using $h^*(\mathcal{S})$ as a split point; therefore, when omitted, the representation will implicitly refer to this optimal split.

MP: Questo teorema/algoritmo lo rimuovo? Tanto ho quello dopo che lo "surclassa"

Theorem 4.6 *The proposed representation of a sequence $\mathcal{S}$ can be built in $O(l \times \log_2 u)$ time and $O(1)$ additional space.*

Proof To construct the proposed representation, we iterate over possible partitioning parameters $0 \leq h < \lceil \log_2 u \rceil$. For each candidate value of $h > 0$, we need to compute $\text{rank}_\mathcal{B}(l)$, which is the only term of $\text{Space}(\mathcal{S}, h)$ that depends on the split given by h on $\mathcal{S}$. We can do that by noting that the number of 1s in $\mathcal{S}$ equals to the number of elements s_i in $\mathcal{S}$ such that $i = 0 \vee s_i^+ \neq s_{i-1}^+$. This computation takes $O(l)$ time per value of h and $O(1)$

[1] In our implementation, we use SDSL `rank_support_v5` and $g(l) \approx 0.0625l$.

space. Since there are at most $\lceil \log_2 u \rceil$ such values, the total time to find the optimal split point h^* (that minimizes space) is $O(l \times \log_2 u)$.

Once h^* is determined, we construct $\mathcal{L}$, $\mathcal{H}$, and $\mathcal{B}$ in $O(l)$ time. Therefore, the total time complexity remains $O(l \times \log_2 u) + O(l) = O(l \times \log_2 u)$. $\square$

Update: In most practical settings, such an algorithm is already fast given its low memory footprint and the fact that we scan $\mathcal{S}$ linearly, thus with a *cache-oblivious* behaviour. However, we can come up with a $O(l + \log_2 u)$ time construction algorithm.

Theorem 4.7 ***Update:*** *It exists a* $O(l + \log_2 u)$ *algorithm that effectively finds* $h^*(\mathcal{S})$ *and hence constructs the compressed representation of* $\mathcal{S}$*. Furthermore, the algorithm requires* $O(\log_2 u)$ *words of additional space.*

Proof Let $b = \lceil \log_2 u \rceil$. We define the function $\text{LCP}(j)$ as the length of the longest common prefix between the b-bit binary representations of elements s_{j-1} and s_j, for $j \in \{1, \ldots, l-1\}$. We compute each $\text{LCP}(j)$ in constant time by using a bitwise XOR followed by a count leading zeros instruction (e.g., `clz(s`$_{j-1}$$\oplus$`s`$_j$`)`), treating the special case $s_{j-1} = s_j$ as $\text{LCP}(j) = b$. For the sake of simplicity, we assume that the `clz` instruction counts the number of leading zeros within the fixed-width b-bit representations of the integers in $\mathcal{S}$ — that is, with each number zero-padded to $b = \lceil \log_2 u \rceil$ bits — rather than over the full machine word size.

First, we compute $\text{LCP}(j)$ for every $j \in \{1, \ldots, l-1\}$. We then store their frequencies in an array $\mathcal{F}$, where $\mathcal{F}[k]$ holds the number of indices j such that $\text{LCP}(j) = k$. Given that $\text{LCP}(j) \leq b$ for every j, $\mathcal{F}$ is long b. We compute $l-1$ LCPs, each taking $O(1)$ time, and we populate the frequency array $\mathcal{F}$. Initializing $\mathcal{F}$ takes $O(b)$ time, and filling it takes $O(l)$ time. Thus, this preprocessing takes $O(l + b)$ time.

Next, we iterate over the possible values of $h \in \{0, 1, \ldots, b-1\}$ in order to find the optimal split point. In order to do that, we need to compute $\text{Space}(\mathcal{S}, h)$. For $h = 0$, this is straight-forward, while for $h > 0$ we need to determine $\text{rank}_{\mathcal{B}}(l)$, which is the number of 1s in $\mathcal{B}$. Let this be R_h.

According to the definition of $\mathcal{B}$:

$$R_h = \mathcal{B}[0] + \sum_{j=1}^{l-1} \mathbb{1}_{s^+_{j-1} \neq s^+_j}.$$

Since $\mathcal{B}[0] = 1$ and $s^+_{j-1} \neq s^+_j$ if and only if $\text{LCP}(j) < h$, we have:

$$R_h = 1 + \sum_{j=1}^{l-1} \mathbb{1}_{\text{LCP}(j) < h}.$$

Using the frequency array $\mathcal{F}$, this sum can be efficiently calculated:

$$R_h = 1 + \sum_{k=0}^{h-1} \mathcal{F}[k].$$

We can compute then all R_h values iteratively. We detail the whole procedure to find $h^*(\mathcal{S})$ in Algorithm 1.

Algorithm 1 Find Optimal Split Point h^*

```
Require: Sequence $\mathcal{S} = (s_0, \dots, s_{l-1})$; overhead function $g(l)$.
Ensure: Optimal split point $h^*$.
 1: $b \leftarrow \lceil \log_2(\max \mathcal{S} + 1) \rceil$
 2: Initialize frequency array $\mathcal{F}[0 \dots b-1] \leftarrow 0$
                                              ▷ Populate frequency array $\mathcal{F}$
 3: for $j \leftarrow 1$ to $l - 1$ do
 4:     if $s_{j-1} \neq s_j$ then
 5:         $\ell \leftarrow \texttt{clz}(s_{j-1} \oplus s_j)$
 6:         $\mathcal{F}[\ell] \leftarrow \mathcal{F}[\ell] + 1$
 7:     end if
 8: end for
                                              ▷ Find optimal split point $h^*$
 9: min_space $\leftarrow l \cdot b$,   $h^* \leftarrow 0$,   $R_h \leftarrow 1$
10: prefix_sum $\leftarrow 0$
11: for $h \leftarrow 1$ to $b - 1$ do
12:     prefix_sum $\leftarrow$ prefix_sum $+ \mathcal{F}[h-1]$
13:     $R_h \leftarrow 1 +$ prefix_sum
14:     curr_space $\leftarrow R_h \cdot h + l \cdot (b - h) + l + g(l)$
15:     if curr_space $<$ min_space then
16:         min_space $\leftarrow$ curr_space
17:         $h^* \leftarrow h$
18:     end if
19: end for
20: return $h^*$
```

The second for-loop in Algorithm 1 runs $b - 1$ times. Each step within this loop takes constant time. Therefore, determining the R_h and Space$(\mathcal{S}, h)$ for all $h \in \{1, \dots, b-1\}$ takes $O(b)$ time, after the initial $O(l + b)$ LCP and frequency computation.

Once h^* is determined, the actual data structures $\mathcal{L}$, $\mathcal{H}$, and $\mathcal{B}$ (along with its rank support) are constructed. This involves a single pass over the input sequence $\mathcal{S}$ to extract the relevant bit parts for $\mathcal{L}$ and to determine $\mathcal{B}$, and then to populate $\mathcal{H}$. This construction takes $O(l)$ time.

Therefore, the total time complexity to find h^* and build the proposed representation is $O(l+b)+O(b)+O(l)=O(l+b)$. Regarding space complexity, the algorithm stores the array $\mathcal{F}$ of length b and a constant number of auxiliary variables, thus $O(b)$ extra space. □

Next, we may argue that the choice of one single optimal split point h^* for a possible long sequence $\mathcal{S}$ is a global optimal choice, which doesn't guarantee that each cluster is compressed optimally. Indeed, clusters might have different lengths and distributions. To address this inefficiency, we can uniformly partition $\mathcal{S}$ into blocks of fixed size and apply the proposed technique to each block. If the size of the block is appropriate, this introduces a negligible overhead while potentially improving compression of $\mathcal{S}$.

Alternatively, we could replicate the two-level design of the so-called Partitioned EF indexing [21], by adopting the technique proposed by Ferragina et al. in [12]. This algorithm approximates the optimal partition of $\mathcal{S}$ to minimize total space usage. It does so by representing the problem as visiting a graph whose edges (i,j) have costs defined as $\text{Space}(\mathcal{S}[i:j], h^*(\mathcal{S}[i,j]))$.

Finally, our method naturally invites comparisons with both bit-packing and EF compressed indexing. We begin by highlighting an immediate guarantee with respect to bit-packing:

Theorem 4.8 *The proposed representation of a sequence $\mathcal{S}$ is never larger than standard bitpacking.*

Proof Let h^* be the optimal split point of $\mathcal{S}$. From the definition of h^* and Theorem 4.4, we have that

$$\text{Space}(\mathcal{S},h^*) \leq \text{Space}(\mathcal{S},0) = l \times \lceil \log_2 u \rceil,$$

which proves that the proposed representation is never larger than standard bitpacking. □

Comparing with Elias–Fano encoding, both codes offer constant-time retrievals of arbitrary elements and exploit the redundancy of the upper part of the bits. A notable advantage of the proposed technique is that it overcomes EF's limitation regarding monotonicity requirements. Additionally, for some sorted sequences, our method may yield a more compact representation thanks to its inherent *cluster-awareness*. This advantage is particularly salient for sorted sequences characterized by extended segments of numerically proximal integers. When these segments are long (i.e., $\text{rank}_{\mathcal{B}}(l)/l$ is small) and h^* itself is large, our technique's compression of these locally dense regions can be more effective than Elias-Fano's, especially if the overall data universe u is large compared to the sequence length l (i.e., u/l is large). **MP:** Qui aggiungere qualcos'altro

4.2.3 Assigning (clustered) Node IDs

We are interested in finding a graph relabelling that favors clusters in the adjacency structure $\mathcal{A}$. Such relabelling should have the following characteristics:

- It should be easily computable (i.e., in polynomial time). This requirement implicitly rules out exact solutions, as finding an optimal graph relabelling is, in most cases, an NP-hard problem (as discussed in Section 2.1).
- The relabelling should favor clusters in both $\mathcal{A}$ and its transpose $\mathcal{A}^T$. This is because we may be interested in representing both so that we can efficiently support backward navigation.

Before proceeding, we must consider two separate situations: (i) nodes with many outgoing edges (high-degree nodes), and (ii) nodes with few outgoing edges (low-degree nodes).

In the first case, it is crucial to adopt a node relabelling strategy that minimizes the interval $\max N_v - \min N_v$. This ensures that, even in unfavorable scenarios (such as when neighborhood IDs N_v are uniformly distributed within the interval $[\min N_v, \max N_v]$), the adjacency sequence N_v has limited variability, thereby increasing opportunities for prefix-sharing compression.

In the second case, the neighborhoods are small, limiting the internal redundancy we may exploit within each adjacency list. Therefore, our primary objective becomes placing these small neighborhoods close to other nodes' neighborhoods, ideally creating clusters across consecutive adjacency lists N_{v-1}, N_v, and N_{v+1}.

Surprisingly, the graph relabelling produced by the Cuthill–McKee (CM) algorithm for undirected graphs (see Section A.1) naturally and effectively accommodates both scenarios. First, the CM ordering explicitly targets the minimization of node bandwidth, effectively limiting the global interval:

$$\max_{v \in V} \left(\max N_v - \min N_v \right) .$$

Additionally, for sparse graphs, the breadth-first search traversal inherent in the CM algorithm implicitly assigns consecutive IDs to nodes with small neighborhoods. Given the low degrees and the bandwidth reduction, chances are that consecutive neighborhoods share common prefixes, resulting in highly compressible clusters.

Together, these two properties incentivize the emergence of adjacency clusters, significantly improving the compression efficiency of the representation discussed in the previous subsection.

Given its undirected formulation, both properties also hold for $\mathcal{A}^{-1}$.

4.3 Representing and assigning Edge IDs

For nodes, we established a mapping between User Node IDs and Internal Node IDs to optimize storage and access. Edges, on the other hand, typically do not have pre-existing *User Edge IDs*; their identifiers are primarily for internal use within the graph database, allowing a unique reference to each edge $e \in E$. The goal is to assign an *Internal Edge ID*, $\mathrm{Id}_E(e)$, from the range $E_I = \{0, \dots, m-1\}$, where $m = |E|$ is the total number of edges.

We can derive these Internal Edge IDs directly from the data structures already proposed for representing the graph's adjacency information, specifically the concatenated neighborhoods list $\mathcal{A}$ and the offsets array $\mathcal{O}$. Recall that $\mathcal{A}$ is a flat list containing all target nodes of all edges, where the neighbors of a node u (i.e., N_u) are stored contiguously and monotonically. The array $\mathcal{O}$ stores the starting index in $\mathcal{A}$ for each node u's neighborhood, such that $N_u = (\mathcal{A}[\mathcal{O}[u]], \mathcal{A}[\mathcal{O}[u]+1], \dots, \mathcal{A}[\mathcal{O}[u+1]-1])$.

Consider an edge $e = (u, v) \in E$. This edge signifies that v is a neighbor of u. Within the sequence of u's neighbors, N_u, this specific edge (u, v) will correspond to v appearing at a particular local 0-indexed position, let's call this k_{uv}. That is, v is the k_{uv}-th neighbor of u in its ordered adjacency list, so $\mathcal{A}[\mathcal{O}[u] + k_{uv}] = v$. The value k_{uv} ranges from 0 to $(\#N_u - 1)$, where $\#N_u$ is the out-degree of node u.

We can then define the Internal Edge ID for $e = (u, v)$ as its global index within the concatenated adjacency list $\mathcal{A}$:

$$\mathrm{Id}_E(e) = \mathcal{O}[u] + k_{uv}.$$

Since $\mathcal{A}$ contains exactly m entries (one for each edge in the graph), and each entry corresponds to a unique source node u and a unique local index k_{uv} within N_u, this assignment ensures that $0 \le \mathrm{Id}_E(e) \le m - 1$.

This approach also naturally handles multigraphs where E is a multiset. If there are multiple edges from node u to node v, say $e_1 = (u, v)$ and $e_2 = (u, v)$, they will occupy distinct positions within u's logical neighborhood list N_u. For instance, e_1 might be the k_1-th neighbor and e_2 the k_2-th neighbor (where $k_1 \neq k_2$). Consequently, they will map to different indices in $\mathcal{A}$: $\mathcal{O}[u] + k_1$ and $\mathcal{O}[u] + k_2$, resulting in unique *Internal Edge IDs*. The specific order of parallel edges within N_u (and thus their k_{uv} values) would typically be determined by their order of appearance in the input data or some other consistent processing rule during the construction of $\mathcal{A}$ and $\mathcal{O}$.

The primary advantage of this edge identification scheme is its space efficiency: no additional data structures are required beyond $\mathcal{A}$ and $\mathcal{O}$, which are already essential for representing the graph topology.

4.4 Querying the graph representation

This section details the implementation of fundamental query operations using the proposed space-efficient graph topology representation. Most of these operations are straightforward derivations from the underlying data structures. We will demonstrate how to retrieve adjacency and incidence information by leveraging the concatenated adjacency list $\mathcal{A}$, the offset array $\mathcal{O}$ for forward traversals (outgoing edges), and their counter-parts $\mathcal{A}^T$ and $\mathcal{O}^T$ built on the inverse directed graph for backward traversals (incoming edges).

4.4.1 Forward queries

As in the previous sections, with N_v we indicate the neighbors of a node v. $N_v(i)$ represents the i-th neighbor of a v. We now introduce the notation $N_v^E(i)$ to indicate the EdgeID assigned to the i-th outgoing edge.

Out-degree of a node

$$\#N_v = \mathcal{O}[v{+}1] - \mathcal{O}[v] \quad \text{for } v = 0, \ldots, n{-}1$$

Accessing the i-th neighbor

$$N_v(i) = \mathcal{A}[\mathcal{O}[v] + i] \quad \text{for } v = 0, \ldots, n{-}1, \;\; i = 0, \ldots, \#N_v{-}1$$

Accessing the ID of the i-th outgoing edge

$$N_v^E(i) = \mathcal{O}[v] + i \quad \text{for } v = 0, \ldots, n{-}1, \;\; i = 0, \ldots, \#N_v{-}1$$

4.4.2 Backward queries

Similarly, with P_v we indicate the parents of a node v (i.e., the nodes u such that $(u, v) \in E$) and use $P_v^E(i)$ to refer to the Edge ID of the i-th incoming edge of v.

In-degree of a node

$$\#P_v = \mathcal{O}^T[v + 1] - \mathcal{O}^T[v] \quad \text{for } v = 0, \ldots, n{-}1$$

Accessing the i-th parent

$$P_v(i) = \mathcal{A}^T[\mathcal{O}^T[v] + i] \quad \text{for } v = 0, \ldots, n{-}1, \;\; i = 0, \ldots, \#P_v{-}1$$

Accessing the ID of the i-th incoming edge

The ID of the i-th incoming edge to node v, denoted $P_v^E(i)$, where $u = P_v(i)$ is the parent node, is given by:

$$P_v^E(i) = \mathcal{O}[u] + k_{uv}$$

where $u = \mathcal{A}^T[\mathcal{O}^T[v] + i]$, and k_{uv} is the index such that $\mathcal{A}[\mathcal{O}[u] + k_{uv}] = v$ that can be determined running a binary search over N_u. Specifically, if $P_v(i)$ is the j-th occurrence of u in P_v, then k_{uv} is the j-th smallest index such that $N_u(k_{uv}) = v$.

Chapter 5

Representing graph properties

In this chapter, we describe how to store node and edge properties in order to offer efficient lookups, while still aiming at compactness. We will focus on node properties, as the edge properties will be stored exactly in the same way.

Properties types

We aim to support properties of the following types: integer, date, double, enum, and string. The first 4 are *fixed-size* types, and we employ different strategies to store them. The fifth one is a *variable-size* type, and we use a more flexible encoding scheme to account for its varying length and content.

We formalize this classification, saying that the set of properties K is a partition of $K^{\text{int}}, K^{\text{date}}, K^{\text{double}}, K^{\text{enum}}, K^{\text{string}}$. Without loss of generality, we also index K with a total ordering such that

$$\forall i, j, k, l, m \quad \begin{cases} K_i \in K^{\text{int}} \\ K_j \in K^{\text{date}} \\ K_k \in K^{\text{double}} \\ K_l \in K^{\text{enum}} \\ K_m \in K^{\text{string}} \end{cases} \quad \Rightarrow \quad i < j < k < l < m$$

This partition also applies to $\mathrm{p}_N : N \times K \to W$, which is defined as follows.

Definition 5.1

$$p_N(v,k) = \begin{cases} \mathrm{p}_N^{int}(v,k) & k \in K^{int} \\ \mathrm{p}_N^{date}(v,k) & k \in K^{date} \\ \mathrm{p}_N^{double}(v,k) & k \in K^{double} \\ \mathrm{p}_N^{enum}(v,k) & k \in K^{enum} \\ \mathrm{p}_N^{string}(v,k) & k \in K^{string} \\ \varepsilon & otherwise \end{cases}$$

where $\mathrm{p}_N^{prop\_type} : K^{prop\_type} \to W$ *have obvious definitions.*

5.1 Fixed-size properties

In this Section, we first propose a scheme common to all the fixed-size properties, and then provide the implementation details tailored to the specific property type.

5.1.1 Node type system

Problem: Different nodes may have different subsets of properties defined. When representing fixed-size properties, storing null values for undefined node properties can become space-inefficient.

To address this, we introduce a notion of node types based on the set of fixed-size properties present in each node.

Definition 5.2 *The type of a node* $n \in V$ *is defined as*

$$\mathrm{Type}(v) = \left\{ k \in K \setminus K^{string} \;\middle|\; \mathrm{p}_V(v,k) \neq \varepsilon \right\}$$

Clearly, $\mathrm{Type}(v)$ can also be represented as a binary vector $\mathcal{B}$ of size $\#(K \setminus K^{\mathrm{string}})$ such that $\mathcal{B}[i] = 1 \iff K_i \in \mathrm{T}ype(v)$.

Definition 5.3 *Let* T *be the set of all types assigned to nodes in* V *(i.e.,* $T = \{\mathrm{Type}(v) \mid v \in V\}$*) and* t *its cardinality. We then define the function*

$$\mathrm{TypeID} : T \to \{0, 1, \ldots, t-1\}$$

as an injective function assigning a unique integer identifier to each distinct type.

This formulation allows nodes to reference their type compactly via a *type ID*, avoiding repeated storage of property schemas.

Finally, we observe the following fact that will be used when designing the compression scheme for fixed-size properties.

Fact: Empirical evidence shows that in real-world LPGs, the number of distinct node types is typically small and remains stable even as the graph size increases.

5.1.2 Property storage layout

Building upon the node type system described above, we now detail the structures used to store fixed-size properties efficiently. The core idea is to group nodes by their assigned type ID and store each group's property data contiguously.

First, we identify the set of distinct node types, each represented as a bit-vector. Suppose we have identified t distinct node types. We store these type bit-vectors in an array $\mathcal{T}\dagger_{\surd}]\int$, where each bit-vector $\mathcal{T}\dagger_{\surd}]\int[i]$ encodes the set of properties defined for nodes of type ID i. Thus, the space needed for all type bit-vectors is $t \times \#(K \setminus K^{\text{string}})$ bits.

Next, we assign a unique type identifier to each node. We store these type identifiers in a single bit-packed array $\mathcal{N}odeTypeId$, whose length equals the number n of nodes. Since each identifier ranges from 0 to $t-1$, this requires exactly $n\lceil \log_2 t \rceil$ bits.

To efficiently locate properties of nodes belonging to a given type, we construct t additional bit-vectors called $\mathcal{N}odeOfType[t]$. Each bit-vector is defined as:

$$\forall v \in V, \quad \mathcal{N}odeOfType[t][v] = 1 \iff \mathcal{N}odeTypeId[v] = t$$

and equipped with rank support. [1]

This representation lets us store property data of nodes sharing the same type in contiguous memory. Specifically, given a node v of type t (i.e., $\mathcal{N}odeTypeId[v] = t$), we obtain the position of v among all nodes of type t by:

$$\text{NodeOffset}(v) = \text{rank}_{\mathcal{N}odeOfType}[t](v)$$

Finally, we store properties separately by type ID t. For each fixed-size property type $prop\_type \in \{\text{int}, \text{date}, \text{double}, \text{enum}\}$, we maintain a distinct property matrix $\mathcal{P}roperties^{prop\_type}[t]$. Each matrix has as many rows as there

[1] Note the following observations: i) We could represent $\mathcal{N}odeTypeId$ using a Wavelet Tree[20, Section 6.2], thus eliminating the need for separate $\mathcal{N}odeOfType$ bit-vectors. However, when the number of node types is relatively small, explicitly storing these bit-vectors remains more space-efficient despite their redundancy. ii) When dealing with very few node types, it can be preferable to forego storing $\mathcal{N}odeTypeId$ altogether and rely exclusively on the $\mathcal{N}odeOfType$ bit-vectors.

are defined properties of $prop\_type$ for nodes of type t, and each row contains the corresponding property values for every node of type t.

Since all properties are indexed according to the global ordering previously defined, the specific index assigned to property $k_i \in K^{prop\_type}$ within nodes of type t is computed as:

$$\begin{aligned}\text{PropertyIndex}(k_i \in K^{prop\_type}) = {} & \text{rank}_{\mathcal{T}ypes[t]}(i) \\ & -\text{rank}_{\mathcal{T}ypes[t]}\left(\min\{j \mid k_j \in K^{\text{prop\_type}}\}\right).\end{aligned}$$

Putting all the pieces together, the encoded value of the property $k_i \in K^{prop\_type}$ for node v is found at:

$$w = \mathcal{P}roperties^{prop\_type}[t][\text{PropertyIndex}(k_i)][\text{NodeOffset}(v)]$$

with $t = \mathcal{N}odeTypeId[v]$. The actual property value is then obtained by applying the decoding function specific to the property's type:

$$\text{p}_V^{prop\_type}(v, k_i) = \begin{cases}\text{decode}^{t,k_i,prop\_type}(w) & \mathcal{T}ypes[t][i] = 1 \\ \varepsilon & \text{otherwise}\end{cases}.$$

5.1.3 Integers and dates

Thanks to the proposed layout, we have the guarantee that nodes with the same properties have consecutive IDs. Storing such properties is now straightforward. We are just left with coming up with a compression scheme tailored for each property type.

Unfortunately, the assumption that we can do over the values assumed by the nodes of the same type is minimal. For instance, we cannot assume any existing ordering on property values given by the node IDs, so we cannot use any compression techniques specific for sorted sequences.

For the integer case, we can still hope the values assumed by properties rarely cover the whole range given by standard int (i.e., either $[-2^{31}, 2^{31}-1]$ or $[-2^{63}, 2^{63}-1]$).

For each integer property (and for each type), we can store its minimum, and we express the other values as differences from it. In this way, we obtain a non-negative integer sequence that can be compressed through bit-packing. Clearly, the more compact the interval of possible values, the more effective this compression is.

Therefore, we define decode^{int} as follows

$$\text{decode}^{int}(t, k_i, w) = \min(t, k_i) + w$$

where $\min(t, k_i)$ is the minimum assumed by nodes of type t by the property k_i.

We can easily generalize such a method to date properties (i.e., properties with values in the form (day, month, year)). The key idea is to map each date to a compact integer representation, so that we can once again apply the same difference-based encoding used for standard integers. We propose two techniques to perform such mapping.

The first technique relies on counting the number of days elapsed since a fixed reference date (e.g., January 1st, 1900). Each date can thus be uniquely represented by the number of days since this epoch. After selecting a fixed and reasonably recent epoch and a valid maximum date (e.g., December 31st, 2100), we can ensure the resulting range of integers covers only actual calendar dates and remains relatively small—around 73,000 values for the 1900–2100 interval, therefore using 17 bits for each date value.

Alternatively, we can encode dates using a simple multiplier-based scheme. For instance, a date (d, m, y) can be mapped to an integer using an expression like:

$$\begin{aligned}\text{date\_int}(m, d, y) = {} & (y - \text{YEAR\_OFFSET}) \times \text{YEAR\_MULT} \\ & + m \times \text{MONTH\_MULT} \\ & + d\end{aligned}$$

where the constants YEAR_OFFSET, YEAR_MULT, and MONTH_MULT are tuned so that the encoding remains compact and the separation between components avoids overlaps (e.g., YEAR_MULT = 450, MONTH_MULT = 35).

Both approaches yield integer representations that can be handled with the same difference-encoding and bit-packing pipeline described above for integer properties. The choice between the two primarily depends on correctness requirements, allowed date ranges, and implementation constraints.

5.1.4 Enums and labels encoding

Enum properties refer to attributes of various types (e.g., integer, string, date, or double) where the set of distinct values across nodes of the same type is significantly smaller than the total number of nodes. This offers us the chance to store such values in order to assign them an index so that we can refer to them through their index.

We denote by $\mathcal{V}(t, p)$ the array that contains all the unique values assumed by nodes of type t for property k_i. Thus, the length of $\mathcal{V}(t, k_i)$ corresponds to the number of distinct values for that property among those nodes.

Next, instead of storing each node's actual property value, we encode it using the index of the corresponding value in $\mathcal{V}(t, k_i)$. These indices are stored in a contiguous bit-packed array, using the minimum number of bits necessary to represent all possible indices.

This compact representation significantly reduces memory usage while enabling fast access to the actual property values via decode^{enum}, which is defined as follows

$$\text{decode}^{enum}(t, k_i, w) = \mathcal{V}(t, k_i)[w].$$

Given that labels usually identify a class of nodes in the dataset, and therefore the number of their unique values is a small fraction of the number of nodes, it is natural to think of them as enum properties. We can therefore implement l_V in terms of p_V or use a stand-alone logic replicating the design just proposed.

5.2 Variable-size Properties

Properties of variable size, primarily strings, require a distinct storage strategy compared to their fixed-size counterparts. The inherent variability in the length of string data makes fixed-width allocation schemes space-inefficient. To address this, we adopt an approach centered around the concatenation of all string property data into a unified indexed structure.

Although this design is primarily intended for storing string properties, it can also be used for fixed-size properties to achieve higher compression ratios[2]. This, however, comes at the cost of slower access speed.

The core idea is to serialize all variable-size properties associated with nodes into a single, comprehensive text string, denoted by $\mathcal{T}$. This string is constructed using two distinct delimiters ξ_0, ξ_1, ξ_2, and ξ_3. These delimiters are chosen such that they do not appear in the alphabet from which property names and values are drawn. After its construction, the string $\mathcal{T}$ is compressed using its FM-index representation, a technique also employed in Section 4.1.2.

We define four delimiter-based markers to structure $\mathcal{T}$:

$$\begin{aligned}
\text{NodeStart}(v) &= \xi_0 \cdot v \\
\text{NodeEnd}(v) &= \xi_1 \cdot v \\
\text{PropertyStart}(v, key) &= \xi_2 \cdot v \cdot \xi_2 \cdot key \cdot \xi_2 \\
\text{PropertyEnd}(v, key) &= \xi_3 \cdot v \cdot \xi_3 \cdot key \cdot \xi_3
\end{aligned}$$

[2]Indeed, the authors of CGraphIndex proposed a similar approach and employed it for most graph properties, regardless of their type.

Let then $K^{var}(v)$ denote the sequence of all variable-size property key-value pairs for a given node v. Suppose the variable-size properties for node v are given by $K^{var}(v) = [(key_1, val_1), (key_2, val_2), \ldots, (key_k, val_k)]$. We then append the following substring to $\mathcal{T}$:

$$\begin{aligned}
&\text{NodeStart}(v)\cdot\\
&\quad \text{PropertyStart}(v, key_1) \cdot val_1 \cdot \text{PropertyEnd}(v, key_1)\cdot\\
&\qquad\qquad\qquad\qquad \vdots\\
&\quad \text{PropertyStart}(v, key_k) \cdot val_k \cdot \text{PropertyEnd}(v, key_k)\cdot\\
&\text{NodeEnd}(v).
\end{aligned}$$

Thus, the full string $\mathcal{T}$ is simply the concatenation of these structured substrings for all nodes that possess variable-size properties.

The patterns NodeStart(v) and NodeEnd(v) allow quick identification of the substring containing all properties of node v. Using the FM-index operation locate$_{\mathcal{T}}$ with these delimiters, we obtain the position range corresponding to node v and then extract the relevant property data with extract$_{\mathcal{T}}$. Similarly, we can directly find the boundaries of a specific property by searching for PropertyStart(v, key) and PropertyEnd(v, key), allowing efficient retrieval of any individual variable-size property value via the same methodology.

Depending on the average length of property values, the frequent inclusion of PropertyStart and PropertyEnd markers may introduce significant overhead. Thus, for datasets dominated by shorter property values, omitting these property-boundary markers entirely at construction time can be more space-efficient. However, when property values are typically long, explicit property delimiters significantly speed up single-property retrieval by providing precise positional boundaries in the extract$_{\mathcal{T}}$ operation.

Chapter 6

Benchmarks

In this chapter, we first present performance evaluations in terms of both space and query efficiency. We then compare our method to alternative compressed graph representations and popular graph databases.

6.1 Experimental Setup

All experiments were run inside a virtual machine configured with an Intel Xeon Gold 6348 CPU (4 cores at 2.60 GHz), 62 GB of RAM, and 1 TB of storage. The VM was hosted on a Microsoft Hyper-V system and ran Ubuntu 22.04.4 LTS. While this setup provides sufficient computational resources for our evaluation, we observed that a standard 2020 consumer laptop delivered up to 2x faster performance on the same benchmarks, underscoring the impact of virtualization. Therefore, the absolute time values should be taken with a grain of salt, focusing instead on relative performance comparisons between our Compressed Graph Database and competing approaches.

We implemented the Compressed Graph Database described in previous chapters, using the `sdsl-lite` library for bit-packing, rank-select bitvectors, Elias-Fano encoding, and FM-index data structures. For LogGraph and CGraphIndex, we used implementations provided by their respective authors. Instead, we implemented CSR ourselves from scratch, again relying on the `sdsl-lite` library for internal structures. Finally, we used ArangoDB 3.12.0-2 and Neo4j 2025.02.0.

In the following sections, space measurements reflect the exact on-disk size of each database. Since our compressed structures support direct querying without decompression, the memory footprint matches disk usage. The sole exception is the parent adjacency lists, which are not explicitly stored on disk because we reconstruct them easily at load time.

Additionally, we normalize all time measurements by the number of nodes retrieved.

6.2 Overall Performance

Table 6.1 summarizes the space and time performance of our approach on the five datasets. The columns show space usage in MB, single-hop and two-hop neighbor query times in nanoseconds per neighbor, fixed-size property retrieval times in nanoseconds per property, variable-size property retrieval times in milliseconds per property, and database build and load times in seconds. Build time refers to the process of constructing the compressed database from scratch, while load time represents the process of resuming the pre-built structures from disk. Although build times are not instantaneous, this aligns well with our assumption of static or slightly dynamic datasets where construction occurs infrequently. Furthermore, both build and load procedures could benefit from parallelization, which we did not implement in this work.

Table 6.1: Space and query-time performance results.

Dataset	Space (MB)	1-hop (ns)	2-hop (ns)	Fixed-size (ns)	Var-size (ms)	Build (s)	Load (s)
amazon	177.7	56.5	**15.8**	360.5	4248.5	234.5	0.7
dblp	563.9	259.5	490.2	415.2	4936.2	1060.0	12.5
mag	27.5	52.6	30.4	364.5	5251.0	343.5	3.5
patents	192.6	88.4	54.4	322.1	**3277.5**	194.1	7.3
prime[1]	**13.1**	**12.0**	8.2	**186.3**	-	5.2	0.3

We further analyzed the space usage by measuring each component of the database separately.

These results also support our design discussed in Chapter **??**, indicating minimal overhead from our type system. This type system provides the additional benefit of avoiding the storage overhead of null fixed-size properties. For instance, we observed a 70% reduction in space required for node fixed-size properties in the prime dataset.

As emphasized earlier in Section **??**, node relabeling plays a crucial role on the compression effectiveness.

Figure 6.2 evidences such intuition by comparing various relabeling methods, showing that traversal-based strategies consistently outperform default and random node orders. This aligns precisely with our theoretical expectations, given that traversal-based techniques empirically minimize graph bandwidth. Among traversal-based methods, we chose CM ordering[2] due to its long-standing effectiveness and widespread adoption in practical scenarios.

[2]For directed graphs, the topological sort required by CM ordering was performed after removing any cycles.

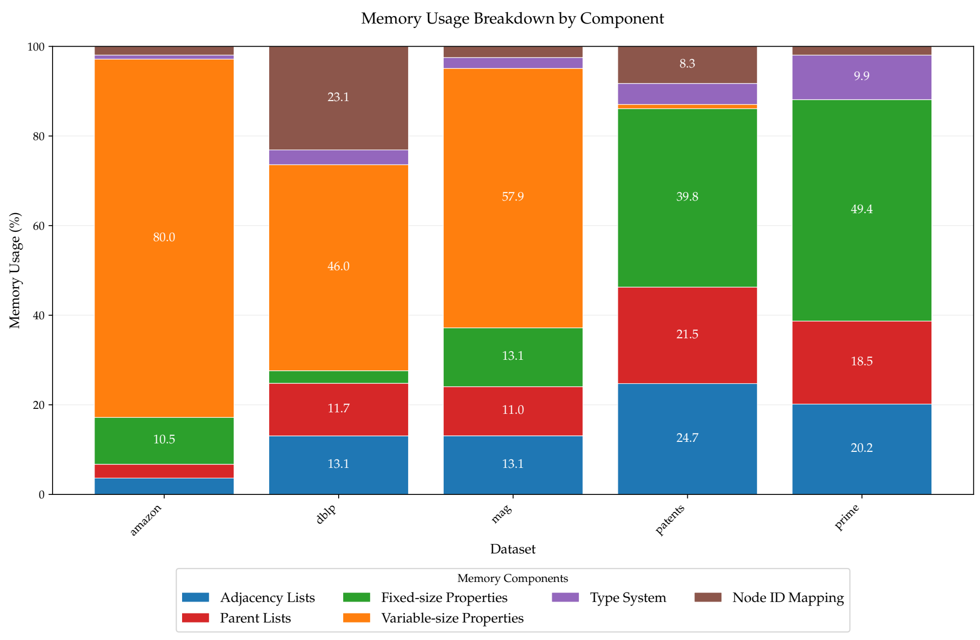

Figure 6.1: Memory usage breakdown by component in our graph database.

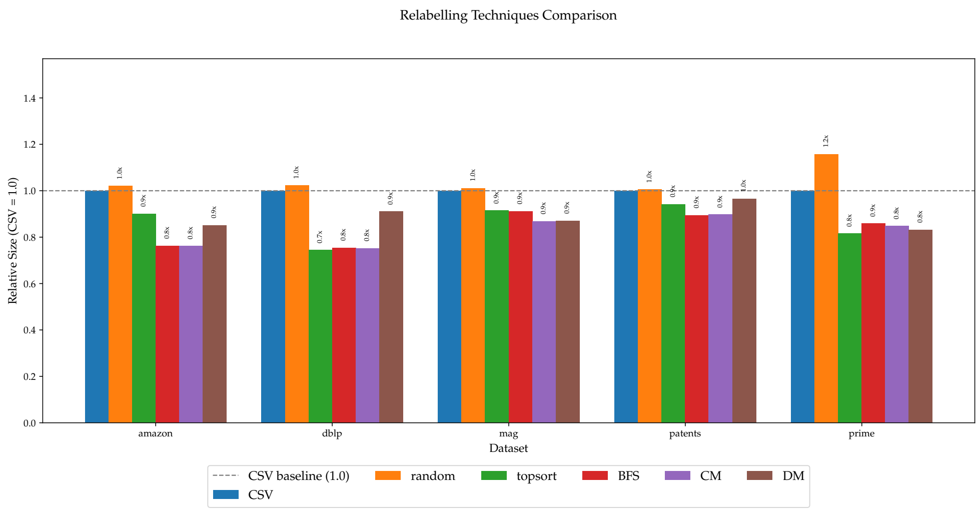

Figure 6.2: Impact of relabeling strategies on compression, relative to default CSV order.

6.3 Comparisons With Compressed Graphs

We first evaluate space and query-performance of our adjacency-list representation against existing compressed graph structures (CSR, LogGraph, CGraphIndex).

Our approach outperforms alternatives in space usage for most datasets,

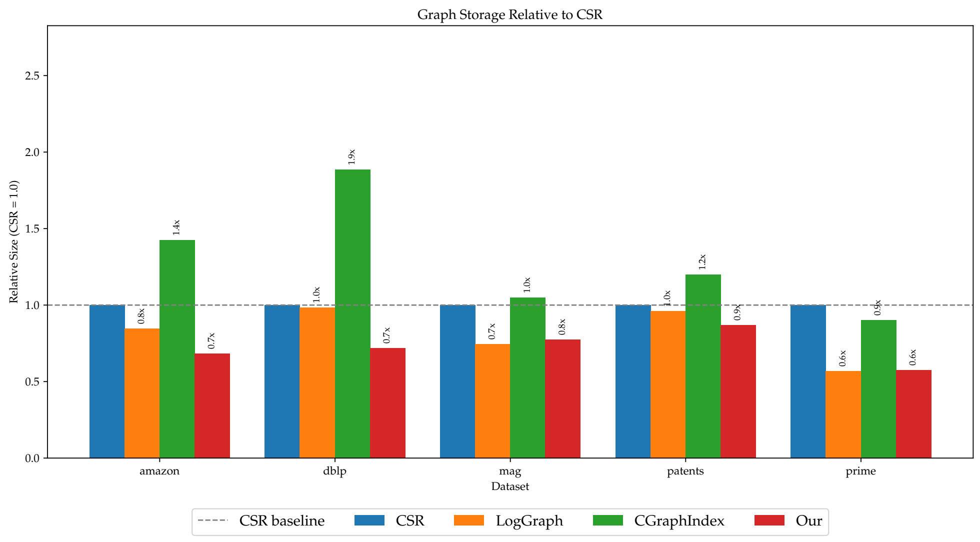

Figure 6.3: Adjacency-list storage sizes normalized to CSR, including our approach.

Table 6.2: Storage (in MB) comparison with different graph representations.

Dataset	CSR	LogGraph	CGraphIndex	Our Approach
amazon	20.1	17.0	28.6	**13.7**
dblp	218.2	214.7	411.2	**156.7**
mag	102.7	**76.4**	107.6	79.5
patents	143.3	137.6	171.7	**124.6**
prime	16.7	**9.5**	15.0	9.6

particularly excelling on sparse graphs where CSR already showed good compression.

As shown by Figure 6.3 and Table 6.2, LogGraph is the second best alternative and provides slightly better performances than our method on dense graphs like prime and mag.

Table 6.3: Average single-hop neighbor query times (ns per neighbor).

Dataset	CSR	LogGraph	CGraphIndex	Our Approach
dblp	**147.8**	602.9	-	259.5
mag	33.1	82.9	**23.2**	52.6
patents	35.4	121.8	**3.8**	88.4
prime	6.3	17.1	**6.0**	12.0

The Pareto-frontier analysis in Figure 6.5, incorporating these results, con-

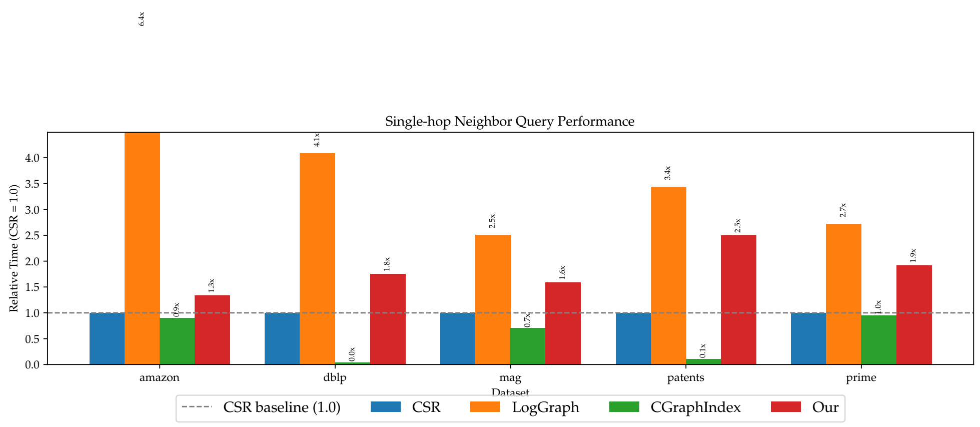

Figure 6.4: Single-hop neighbor query performance comparison between compressed graph representations.

firms our method offers competitive or superior space-time trade-offs compared to LogGraph and CGraphIndex.

For completeness, we note that for 2-hop neighborhood queries, LogGraph and CGraphIndex outperform both CSR and our method, particularly on sparse graphs (Figure 6.6). Additionally, we evaluated a variant of CSR that uses bitpacking for the offset representation, obtaining negligible improvements. This last variant, despite minimal decompression overhead and cache-friendly behavior, still performs worse than LogGraph and CGraphIndex. Thus, we reasonably attribute the performance gap we observed to internal implementation details of the `sdsl` library or to a lack of low-level optimizations. In any case, these factors lie beyond the algorithmic scope of this analysis and we are confident that such low-level optimizations can be replicated to both CSR and our method, thus potentially narrowing the identified performance gap.

6.4 Comparison with Graph Databases

We now compare storage usage and query times versus Neo4j and ArangoDB. Following recent literature, we primarily compare to Neo4j as it outperforms ArangoDB in graph processing tasks [**?**]. All experiments on Neo4j use batched queries for improved performance.

In terms of storage usage, our approach substantially outperforms both Neo4j and ArangoDB, achieving database sizes significantly smaller than even zipped versions of the original datasets.

Our solution also significantly surpasses Neo4j in performance for both single-hop and two-hop neighbor queries.

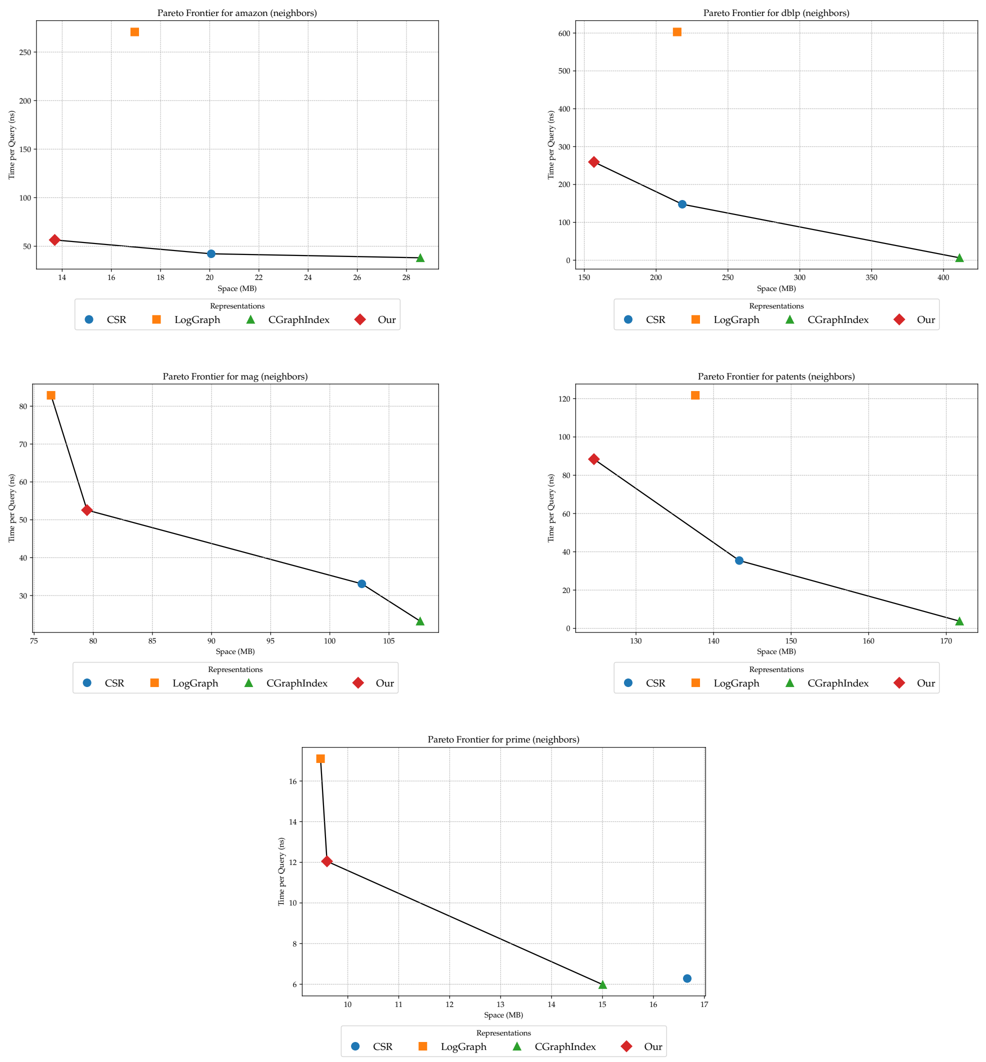

Figure 6.5: Pareto frontier analysis for single-hop neighbor queries showing space-time trade-offs across all datasets.

While we also achieve faster fixed-size property retrievals, our variable-size property retrievals are comparatively slower. This reflects an intentional trade-off: highly compressed FM-index encoding significantly saves space for infrequently accessed data, at a known performance cost. These parameters could be relaxed to favor faster retrieval if needed.

In summary, our compressed approach delivers strong benefits in terms of both space efficiency and query speed for frequently executed graph operations, making it particularly attractive in settings where storage constraints exist and variable-sized properties are rarely accessed.

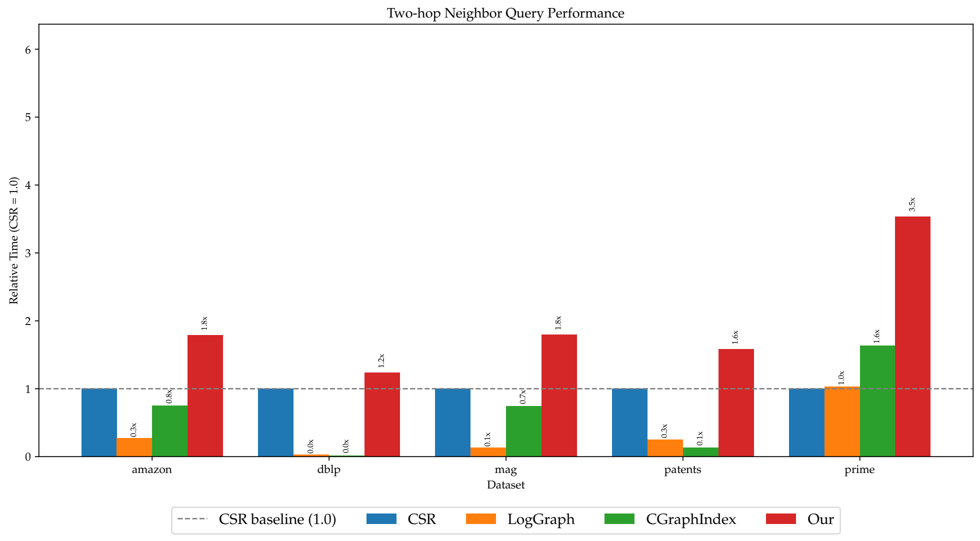

Figure 6.6: Two-hop neighbor query performance comparison between compressed graph representations.

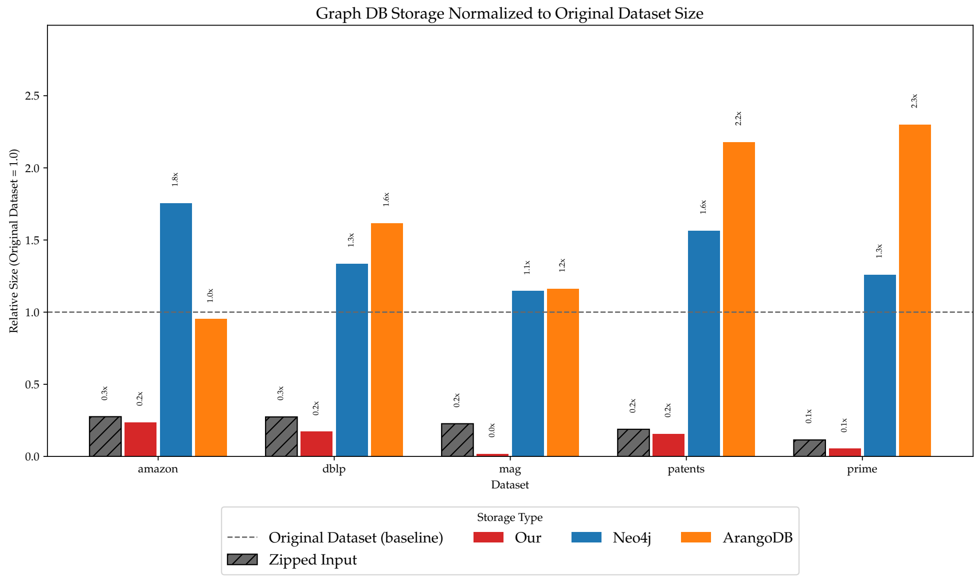

Figure 6.7: Database size comparison normalized against original datasets.

Table 6.4: Database sizes (MB) for our system vs. competing databases.

Dataset	Ours	Neo4j	ArangoDB	Zipped data	Unzipped data
amazon	**177.7**	1331.2	721.2	208.8	758.3
dblp	**563.9**	4403.2	5336.7	903.1	3301.1
mag	**27.5**	1965.7	1991.1	388.9	1716.4
patents	**192.6**	1945.6	2709.1	233.9	1244.4
prime	**13.1**	312.0	570.0	28.3	248.0

Table 6.5: Single-hop neighbor query times (ns/neighbor).

Dataset	Our approach	Neo4j
amazon	**56.5**	210.8
dblp	**259.5**	1250.3
mag	**52.6**	180.7
patents	**88.4**	320.1
prime	**12.0**	45.2

Table 6.6: Two-hop neighbor query times (ns/neighbor).

Dataset	Our approach	Neo4j
amazon	**15.8**	450.2
dblp	**490.2**	2850.7
mag	**30.4**	420.3
patents	**54.4**	680.9
prime	**8.2**	95.4

Table 6.7: Fixed-size property retrieval times (ns/property).

Dataset	Our approach	Neo4j
amazon	**360.5**	720.3
dblp	**415.2**	890.4
mag	**364.5**	750.2
patents	**322.1**	680.9
prime	**186.3**	520.7

Table 6.8: Variable-size property retrieval times (ms/property).

Dataset	Our approach	Neo4j
amazon	4248.5	**8.1**
dblp	4936.2	**12.4**
mag	5251.0	**10.8**
patents	3277.5	**11.3**

Chapter 7

Conclusions

Bibliography

[1] ISO/IEC 39075:2024 Information technology — Database languages — GQL. International Standard ISO/IEC 39075:2024, International Organization for Standardization and International Electrotechnical Commission, Geneva, Switzerland, 2024. Edition 1, published April 2024. Standard to be revised (Stage: 90.92).

[2] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1), February 2008.

[3] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *ACM Computing Surveys*, 56(2):Article 31, 2023.

[4] Maciej Besta, Dimitri Stanojevic, Tijana Zivic, Jagpreet Singh, Maurice Hoerold, and Torsten Hoefler. Log(graph): A near-optimal high-performance graph representation, 2020.

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

[6] Payal Chandak, Kexin Huang, and Marinka Zitnik. Building a knowledge graph to enable precision medicine. *Scientific Data*, 10(1):67, Feb 2023.

[7] Andrea Colombo, Anna Bernasconi, and Stefano Ceri. Modelling legislative systems into property graphs to enable advanced pattern detection. *arXiv*, 2024.

[8] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, page 157–172, New York, NY, USA, 1969. Association for Computing Machinery.

[9] Paolo Ferragina. *Pearls of Algorithm Engineering*. Cambridge University Press, 2023.

[10] Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of wavelet trees. *Information and Computation*, 207(8):849–866, 2009.

[11] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, July 2005.

[12] Paolo Ferragina, Igor Nitto, and Rossano Venturini. On optimally partitioning a text to improve its compression, 2009.

[13] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1433–1445, New York, NY, USA, 2018. Association for Computing Machinery.

[14] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[15] Bronwyn H Hall, Adam B Jaffe, and Manuel Trajtenberg. The nber patent citation data file: Lessons, insights and methodological tools. Working Paper 8498, National Bureau of Economic Research, October 2001.

[16] Nicolas Heist, Sven Hertling, Daniel Ringler, and Heiko Paulheim. Knowledge graphs on the web – an overview, 2020.

[17] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia D'amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. Knowledge graphs. *ACM Comput. Surv.*, 54(4), July 2021.

[18] Hongwei Huo, Yongze Yu, Zongtao He, and Jeffrey Scott Vitter. Indexing Labeled Property Multidigraphs in Entropy Space, with Applications . In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, pages 2478–2492, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[19] Giovanni Manzini. The Burrows-Wheeler Transform: Theory and practice. 09 1999.

[20] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

[21] Giuseppe Ottaviano and Rossano Venturini. Partitioned elias-fano indexes. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '14, page 273–282, New York, NY, USA, 2014. Association for Computing Machinery.

[22] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[23] Ch. H. Papadimitriou. The NP-Completeness of the bandwidth minimization problem. *Computing*, 16(3):263–270, 1976.

[24] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43–es, November 2007.

[25] Johan Sandell, Einar Asplund, Workneh Yilma Ayele, and Martin Duneld. Performance comparison analysis of arangodb, mysql, and neo4j: An experimental study of querying connected data, 2024.

[26] Guus Schreiber and Yves Raimond. RDF 1.1 Primer, June 2014. W3C Working Group Note, 24 June 2014.

[27] Jennifer Scott and John Reid. Reducing the total bandwidth of a sparse unsymmetric matrix. *SIAM Journal on Matrix Analysis and Applications*, 28, 01 2006.

[28] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June (Paul) Hsu, and Kuansan Wang. An overview of microsoft academic service (mas) and applications. In *Proceedings of the 24th International Conference on World Wide Web*, 2015.

[29] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. Arnetminer: extraction and mining of academic social networks. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, page 990–998, New York, NY, USA, 2008. Association for Computing Machinery.

[30] Sebastiano Vigna. Quasi-succinct indices, 2012.

[31] Kuansan Wang, Zhihong Shen, Chiyuan Huang, Chieh-Han Wu, Yuxiao Dong, and Anshul Kanakia. Microsoft academic graph: When experts are not enough. *Quantitative Science Studies*, 2020. MIT Press Direct.

[32] Shirley Wu, Shiyu Zhao, Michihiro Yasunaga, Kexin Huang, Kaidi Cao, Qian Huang, Vassilis N. Ioannidis, Karthik Subbian, James Zou, and Jure Leskovec. Stark: Benchmarking llm retrieval on textual and relational knowledge bases, 2024. arXiv preprint.

[33] Zhentao Xu, Mark Jerome Cruz, Matthew Guevara, Tie Wang, Manasi Deshpande, Xiaofeng Wang, and Zheng Li. Retrieval-augmented generation with knowledge graphs for customer service question answering. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, page 2905–2909. ACM, July 2024.

Appendix A

Appendix

A.1 Cuthill-McKee algorithm

Algorithm 2 Cuthill–McKee Relabelling

```
Require: A set of edges $E$, where each edge is a pair $(u, v)$.
Ensure: A mapping $\pi : V \to \{0, 1, \dots, |V| - 1\}$, where $V$ is the set of all vertices.
 1: Initialize $G \leftarrow \emptyset$, $\pi \leftarrow \emptyset$, and $Nodes \leftarrow \emptyset$.
 2: for each edge $(u, v)$ in $E$ do
 3:     Append $v$ to $G[u]$ and $u$ to $G[v]$.
 4:     Insert $u$ and $v$ into $Nodes$.
 5: end for
 6: for each vertex $v \in Nodes$ do
 7:     if $v \notin G$ then    ▷ Ensure all nodes have an entry in G, even if isolated
 8:         Set $G[v] \leftarrow \emptyset$.
 9:     end if
10: end for
11: for each vertex $v \in Nodes$ do
12:     Compute degree $deg[v] \leftarrow |G[v]|$.
13: end for
14: Initialize a priority queue $PQ$, ordering vertices by increasing $deg$ (with tie-breaking by vertex id).
15: for each vertex $v \in Nodes$ do
16:     Insert $v$ into $PQ$.
17: end for
18: Initialize $Visited \leftarrow \emptyset$ and $rank \leftarrow 0$.
19: while $|Visited| < |Nodes|$ do
20:     Select a vertex $s \notin Visited$ from $PQ$ (or any unvisited vertex if $PQ$ is exhausted).
21:     Initialize an empty queue $Q$.
22:     Enqueue $s$ into $Q$ and mark $s$ as visited.
23:     while $Q$ is not empty do
24:         $curr \leftarrow$ dequeue($Q$).
25:         Assign label: $\pi[curr] \leftarrow rank$; $rank \leftarrow rank + 1$.
26:         Let $N \leftarrow \{ w \in G[curr] : w \notin Visited \}$.
27:         Sort the list $N$ in increasing order of $deg$ (using vertex id to break ties).
28:         for each vertex $w$ in sorted $N$ do
29:             Mark $w$ as visited.
30:             Enqueue $w$ into $Q$.
31:         end for
32:     end while
33: end while
34: return $\pi$.
```

A.2 When bitpacking beats EF

We want to identify when the Elias–Fano (EF) encoding of a general, non-sorted integer sequence $S = (S_1, \ldots, S_n)$ becomes more compact than its simple bit-packed representation.

A concise analysis, considering minor rounding artifacts due to ceiling operations, shows that bit-packing is at least as compact as EF precisely when:

$$\lceil \log_2(M+1) \rceil \leq 2 + \lceil \log_2(A + 1/n) \rceil,$$

where $M = \max S$ and $A = (1/n) \sum_i S_i$ is the arithmetic mean.

If we ignore these minor rounding issues—easily justified by assuming power-of-two conditions—the inequality simplifies neatly to:

$$M \leq 4 \cdot A.$$

This gives a straightforward and practical rule of thumb: the bit-packed representation is more compact exactly when the ratio between the largest value and the average satisfies:

$$\frac{\max S}{\operatorname{avg} S} \leq 4.$$

Otherwise, the EF encoding of the prefix sums will be smaller and more compact.