GraphAr: An Efficient Storage Scheme for Graph Data in Data Lakes

Xue Li
Alibaba Group
youli.lx@alibaba-inc.com

Weibin Zeng
Alibaba Group
qiaozi.zwb@alibaba-inc.com

Zhibin Wang
Nanjing University
wzbwangzhibin@gmail.com

Diwen Zhu
Alibaba Group
diwen.zdw@alibaba-inc.com

Jingbo Xu
Alibaba Group
xujingbo.xjb@alibaba-inc.com

Wenyuan Yu
Alibaba Group
wenyuan.ywy@alibaba-inc.com

Jingren Zhou
Alibaba Group
jingren.zhou@alibaba-inc.com

ABSTRACT

Data lakes, increasingly adopted for their ability to store and analyze diverse types of data, commonly use columnar storage formats like Parquet and ORC for handling relational tables. However, these traditional setups fall short when it comes to efficiently managing graph data, particularly those conforming to the Labeled Property Graph (LPG) model. To address this gap, this paper introduces GraphAr, a specialized storage scheme designed to enhance existing data lakes for efficient graph data management. Leveraging the strengths of Parquet, GraphAr captures LPG semantics precisely and facilitates graph-specific operations such as neighbor retrieval and label filtering. Through innovative data organization, encoding, and decoding techniques, GraphAr dramatically improves performance. Our evaluations reveal that GraphAr outperforms conventional Parquet and Acero-based methods, achieving an average speedup of 4452× for neighbor retrieval, 14.8× for label filtering, and 29.5× for end-to-end workloads. These findings highlight GraphAr's potential to extend the utility of data lakes by enabling efficient graph data management.

PVLDB Reference Format:
Xue Li, Weibin Zeng, Zhibin Wang, Diwen Zhu, Jingbo Xu, Wenyuan Yu, and Jingren Zhou. GraphAr: An Efficient Storage Scheme for Graph Data in Data Lakes. PVLDB, 18(3): 530 - 543, 2024.
doi:10.14778/3712221.3712223

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/apache/incubator-graphar/tree/research.

1 INTRODUCTION

Data lakes have quickly become an essential infrastructure for organizations looking to store and analyze diverse datasets in their raw formats [27, 33, 48, 55, 56, 63, 74]. As centralized repositories, they offer unparalleled flexibility in accommodating a wide array of data types, from structured relational tables to unstructured logs and text files. Crucially, they serve as a cost-effective solution for archiving data at scale while still allowing for queries on archived or rarely accessed data. This dual utility makes them invaluable for both real-time analytics and long-term data management. Columnar storage formats like Parquet [7] and ORC [6] have become standard for storing tabular data in data lakes due to their robust compression and efficient query capabilities.

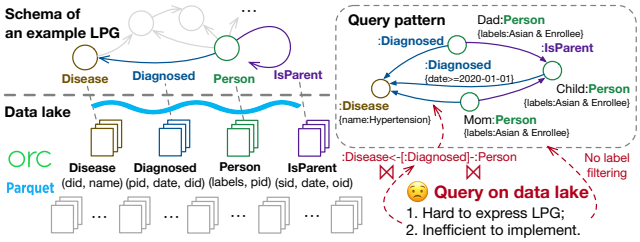


Figure 1: A graph-related query within the data lake.

In sync with these trends, graph data has become increasingly importance, especially for modeling complex relationships. Leading graph-related systems like Neo4j [19], TigerGraph [36], JanusGraph [16] and GraphScope [38] leverage the Labeled Property Graph (LPG) model [25, 26, 28] for this purpose. The recent ISO SQL:2023 standard includes a SQL/PGQ extension that not only facilitates querying LPGs but also enables the creation of LPG views from relational tables [1]. This groundbreaking inclusion highlights the growing convergence of relational and graph data models and emphasizes the need to integrate LPGs into data lakes. Consequently, LPGs are making their way into data lakes for multiple uses, from backups and archives for existing graph databases to natural extensions of transactional, log, and tabular data.

Managing and analyzing LPGs in data lakes offers several significant benefits. Graph-specific queries are often more naturally articulated in languages like Cypher [42], Gremlin [65], GQL [41], or SQL/PGQ, providing an intuitive framework for conducting comprehensive analysis of entity relationships, facilitating the discovery of valuable insights. Data lakes also provide computational flexibility for running complex graph algorithms, enabling the exploration of intricate patterns. Additionally, they offer cost-effective storage solutions, allowing organizations to utilize more affordable and colder storage options without sacrificing query performance. Most

Proceedings of the VLDB Endowment, Vol. 18, No. 3 ISSN 2150-8097.
doi:10.14778/3712221.3712223

notably, data lakes enable seamless querying across both graph and relational data, ushering in a holistic approach to data analytics.

As shown in Figure 1, the example workload illustrates a scenario of immense relevance to public health researchers. The query aims to count the number of families—comprising a father, mother, and child—each labeled as Asian and Enrollee (indicating their participation in a health study), and diagnosed with hypertension since 2020. Such queries hold significant utility for public health studies as they allow for the analysis of correlations between familial relationships, racial groups, and specific health conditions like hypertension among study participants. Understanding these relationships can be critical for targeted health interventions and for identifying possible social or genetic factors contributing to disease prevalence. Within the context of this research query, data lakes offer an economical and scalable solution for storing diverse, multi-source, and often historical health-related data. More importantly, the intricate relationships and multiple attributes required by this research are more naturally and efficiently captured through property graph queries than through traditional SQL. However, integrating LPGs into data lakes introduces unique challenges:

Challenge 1: There is no standardized way to encapsulate an LPG within the existing data lake architecture. While columnar formats like Parquet and ORC excel at storing individual tables, they fall short in representing the complex relationships and semantics across these tables, which are inherent to LPGs.

Challenge 2: Graph-specific operations can be highly inefficient in this setup. The foundational operation, neighbor retrieval, might require multiple joins, significantly impacting performance.

Challenge 3: Label filtering, another essential graph-specific operation, also introduces inefficiency due to the lack of native support in columnar formats.

GraphAr. To address these challenges, we introduce GraphAr[1], an efficient storage scheme for graph data in data lakes. It is designed to enhance the efficiency of data lakes utilizing the capabilities of existing formats, with a specific focus on Parquet in this paper. GraphAr ensures seamless integration with existing tools and introduces innovative additions specifically tailored to handle LPGs.

Firstly, Parquet provides flexible and efficient support for various datatypes, including atomic types (e.g., bools and integers), and nested and/or repeated structures (e.g., arrays and maps). Leveraging Parquet as its fundamental building block, GraphAr further introduces standardized YAML files to represent the schema metadata for LPGs, alongside a hierarchical data layout to store the data. This innovative combination of data organization with metadata management enables the complete expression of LPG semantics, while ensuring compatibility with both data lake ecosystems and existing graph-related systems, addressing **Challenge 1**.

GraphAr incorporates specialized optimization techniques to improve the performance of critical graph operations, which are not inherently optimized in existing formats. To address **Challenge 2**, GraphAr facilitates neighbor retrieval by organizing edges as sorted tables in Parquet to enable an efficient CSR (Compressed Sparse Row) or CSC (Compressed Sparse Column)-like representation, and leveraging Parquet's delta encoding to reduce overhead in data storage and loading. GraphAr also introduces an innovative decoding algorithm that utilizes instruction sets such as BMI (Bit Manipulation Instructions) and SIMD (Single Instruction, Multiple Data), along with a unique structure named PAC (Page-Aligned Collections), to further accelerate the neighbor retrieval process.

In addressing **Challenge 3** of label filtering, GraphAr adapts the RLE (Run-Length Encoding) technique from Parquet and introduces a novel interval-based decoding algorithm. Through integrating proven methods (CSR/CSC, delta encoding, RLE) with novel decoding algorithms, GraphAr delivers a comprehensive and efficient solution for optimizing LPG-specific operations.

Our key contributions can be summarized as follows:

- Elucidation of challenges and limitations in existing tabular formats for managing LPGs in data lakes (Section 2).
- A strategic choice of Parquet compatibility, a standardized YAML to fully express LPG semantics, and detailed specification for organizing LPGs in Parquet (Section 3).
- Development of specialized optimization techniques for enhancing performance in neighbor retrieval and label filtering operations (Sections 4 and 5). These are built upon Parquet's advanced encoding features and are complemented by two innovative and efficient decoding algorithms.
- Comprehensive performance evaluation of GraphAr compared to Parquet and Acero-based implementations, highlighting substantial speed gains: on average 4452× for neighbor retrieval, 14.8× for label filtering, and 29.5× for end-to-end workloads. And the potential for integrating GraphAr into existing graph systems (Section 6).

2 BACKGROUND AND KEY CHALLENGES

In this section, we discuss the limitations of using tabular file formats like Parquet and ORC in data lakes for LPGs, a common graph data model. We explore how these formats inadequately support LPG representation and efficient graph queries, laying the groundwork for the challenges that GraphAr tackles.

2.1 Tabular Formats in Data Lakes

Tabular data is key to data lakes, aiding efficient organization, analysis, and data extraction from large sets. Columnar formats like Parquet [7] and ORC [6] are popular due to their robust features. Unlike row-based formats such as CSV, they allow faster queries by enabling selective column reading, avoiding unnecessary data. Additionally, they offer diverse and efficient compression and encoding strategies, such as delta encoding to compress the variance between consecutive values, and run-length encoding to compress repetitive values. These techniques not only reduce storage needs but also enhance processing speeds. Another advantage of Parquet and ORC is predicate pushdown, which enhances query performance by moving filters closer to the storage layer, thus minimizing reads.

The combination of selective column reading, efficient compression, and predicate pushdown positions Parquet and ORC as the go-to choices for managing tabular data in data lakes. Previous studies have demonstrated the importance of leveraging their capabilities for optimizing relational data management [27, 40, 47]. Recent research [10, 52, 70–73] has also explored enhancements to

[1] Apache GraphAr is an effort undergoing incubation at the Apache Software Foundation (ASF), sponsored by the Apache Incubator.

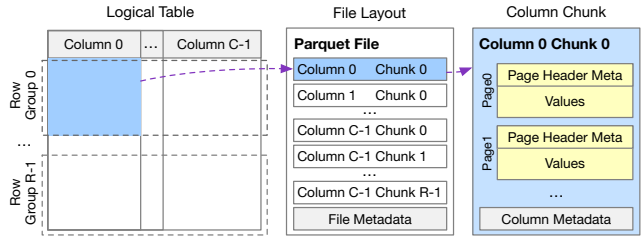
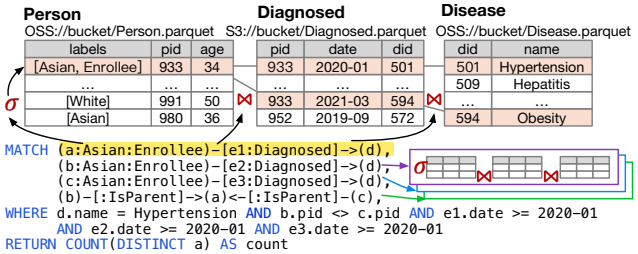
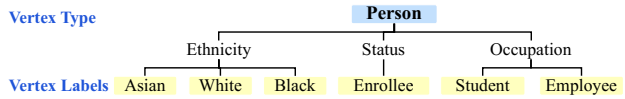


Figure 2: The internal structure of a Parquet file for a logical table with C columns and R row groups.



(a) Execution workflow of the query on tabular formats.



(b) Inherent multi-labeling characteristics inside Person vertices.

Figure 3: An example of querying LPGs on tabular formats.

tabular formats, utilizing CPU instruction sets like BMI and SIMD. In this paper, we will focus on Parquet, but the techniques discussed can be seamlessly adapted to other columnar formats such as ORC.

Parquet. Figure 2 illustrates the internal structure of a Parquet file. Structurally, a Parquet file represents a table, organized into row groups for logical segmentation. Within a row group, the data of a column is stored in a column chunk, which is guaranteed to be contiguous in the file. Column chunks are further divided into pages, the indivisible units for compression and encoding. These pages, which can vary in type, are interleaved in a column chunk.

Parquet files contain three layers of metadata: file metadata, column metadata, and page header metadata. The file metadata directs to the starting points of each column's metadata. Inside the column chunks and pages, the respective column and page header metadata are stored, offering a detailed description of the data. This includes data types, encoding, and compression schemes, facilitating efficient and selective access to data pages within columns.

2.2 Labeled Property Graphs

Labeled Property Graphs (LPGs) [25, 26, 28] excel at representing complex relationships and semantics in a natural manner. Their flexible schema allows for accommodating the diverse and evolving nature of big data within repositories, making them integral to data lakes. LPG serves as the canonical data model in many graph systems [19, 36, 43] and graph query languages [11, 35, 41, 42, 65, 69], enabling queries and analytics to uncover valuable insights and patterns. Formally, an LPG is defined as $G = (V, E, T_V, T_E, P, L)$, where V is a set of vertices, E a set of edges, T_V and T_E the types of vertices and edges respectively, P the properties, and L the labels.

For each vertex $v \in V$ and edge $e \in E$, they are associated with a type $t_v \in T_V$ and $t_e \in T_E$ respectively, and can have optional properties. A property $p \in P$ is specific to a vertex or an edge type, with a unique identifier within its type and a pre-defined datatype for its values. This implies that vertices or edges of the same type share the same set of properties, and can be stored in a tabular format, as shown in Figure 3a.

Furthermore, each label $l \in L$ has a unique identifier, usually a string. Each vertex type t_v is linked with a set of candidate labels $L(t_v) \subseteq L$, allowing each vertex of this type to be assigned zero or more labels. Labels hold significant importance in LPGs as they represent classifications and characteristics of entities, while properties serve as attributes to store additional information. LPGs allow vertices to have multiple labels, offering a flexible and expressive way to describe entities. For instance, Figure 3b illustrates an inheritance hierarchy of labels within the *Person* vertices, where a vertex can be labeled as both Asian and Enrollee. Although edge types could technically also be labeled, we focus solely on vertex labels in this paper, aligning with common graph query practice[2].

While Parquet is highly effective for storing individual vertex and edge types T_V and T_E along with their associated properties due to the columnar structure and data compression capabilities, it falls short in capturing the interconnected schema essential for linking vertices with edges, e.g., to express the relationships across the three tables of Figure 3a, which represent two vertex types and one edge type. This limitation is crucial for efficient graph traversal and pattern matching. Moreover, Parquet lacks native support for the multi-labeling capability of LPGs, resulting in a loss of complex semantics and relationships inherent to LPGs.

2.3 Querying Labeled Property Graphs

The core feature shared among graph query languages is the facility for pattern matching [11, 35]. This capability allows for in-depth analysis of the relationships between entities, uncovering valuable insights and patterns that may not be readily apparent in other data models. Given the fact that a LPG consists of topology, labels, and properties, a graph pattern is then defined as vertices and their connections through edges, filtered based on labels and property values [41]. Figure 3a illustrates the example workload mentioned in Figure 1, expressed in Cypher. And the steps for matching `(a:Asian:Enrollee)-[e1:Diagnosed]->(d)` are highlighted in the figure.

When it comes to properties, a viable approach is to use native tabular data, leveraging existing formats for efficient storage and property-related operations. However, this approach struggles with two crucial aspects of pattern matching.

Firstly, tabular formats lack native support for representing graph topology, making it difficult to efficiently fetch the neighboring vertices and edges for a given vertex. A common workaround is to store edge endpoints as properties and use the join operations across multiple tables to retrieve neighbors, as shown in Figure 3a. However, this approach is often inefficient due to the computational overhead of multiple joins.

[2]Graph query languages like Cypher and Gremlin typically adhere to the convention that an edge can have only one classification, corresponding to the edge type in LPG model. Nevertheless, the strategies for vertex labels discussed in this paper can be seamlessly extended to support edge labels.

Secondly, label filtering is a unique feature in graph queries, to enable the selection of specific subsets of vertices, making it an essential element in all graph query languages [11, 41, 65, 69]. Existing formats do not natively accommodate this flexibility and do not provide a foundation for label-based optimizations. Encoding labels as ordinary properties and performing filtering by string matching, as seen in the initial step of Figure 3a, limits the expressive power of vertex representation and hampers efficient label handling.

2.4 Key Challenges Addressed by GraphAr

The development of GraphAr is motivated by the specific limitations of existing tabular formats for both representing LPGs and supporting efficient graph queries.

Challenge 1: Effective LPG representation. LPGs use type-based organization and specific label/property definitions to form a cohesive graph structure. This enables precise and targeted querying. Existing tabular formats fall short of capturing these intricate semantics, necessitating a specialized solution. This challenge is addressed in Section 3.

Challenge 2: Efficient neighbor retrieval. A fundamental aspect of graph queries is the operation known as *neighbor retrieval*. This is vital for quick access to adjacent vertices and edges, thus accelerating graph traversal. Existing tabular formats, however, do not natively or efficiently support this crucial operation. This issue is tackled in Section 4.

Challenge 3: Optimized label filtering. *Label filtering* is a primary filtering mechanism in graph queries, allowing for the early elimination of irrelevant data. Existing tabular formats do not natively support this operation, making it a ripe area for optimization. This is the focus of Section 5.

Each of these challenges represents a gap in the capabilities of current tabular formats for graph data, which serve as the focus areas for the technical contributions of GraphAr.

3 REPRESENTING LPGS IN GRAPHAR

This section provides an overview of how GraphAr customizes the representation of LPGs in data lakes. It begins by outlining its goals and non-goals, providing clarity on the rationale behind its design. Next, it explains the strategies employed for data organization and layout. Lastly, the section describes how GraphAr seamlessly integrates into the data lake ecosystem.

3.1 Goals and Non-Goals

Goals. GraphAr's primary goal is to provide an efficient storage and management scheme for LPGs in data lakes, specifically targeting the three main challenges outlined in Section 2.

GraphAr also seeks compatibility with both data lake and graph processing ecosystems for smooth integration with a variety of existing tools and systems.

Non-Goals. GraphAr does not intend to replace existing data lake formats like Parquet and ORC, but to maximize their benefits and offer additional features for LPGs.

In line with the established practices of data warehousing and lake house architectures, both Parquet/ORC and GraphAr adhere to data immutability norms, treating batch-generated data as immutable once created. Higher-level systems such as graph databases manage mutation (e.g., adding, deleting, or updating vertices) through specialized, non-standardized file and in-memory versioning methods.

GraphAr itself is not a graph computing engine; rather, it can be non-intrusively integrated with graph processing systems, either serving as the archival format or acting as a data source.

3.2 Data Organization and Layout

In GraphAr, vertices and edges are organized according to their types, which aligns with the principles of the LPG model. Parquet is utilized as the payload file format for storing the data in the physical storage, while YAML files are used to capture schema metadata.

Schema metadata. A YAML file (Figure 4a) stores the metadata for a graph. It specifies important attributes such as file path prefixes and vertex/edge types. This file serves as a nimble yet effective way to capture metadata that is not accommodated by Parquet, while Parquet files include specific details about properties and labels within their internal schemas. It can optionally include partition sizes, allowing for data to be segmented into multiple physical Parquet files, thereby enabling parallelism at the file level.

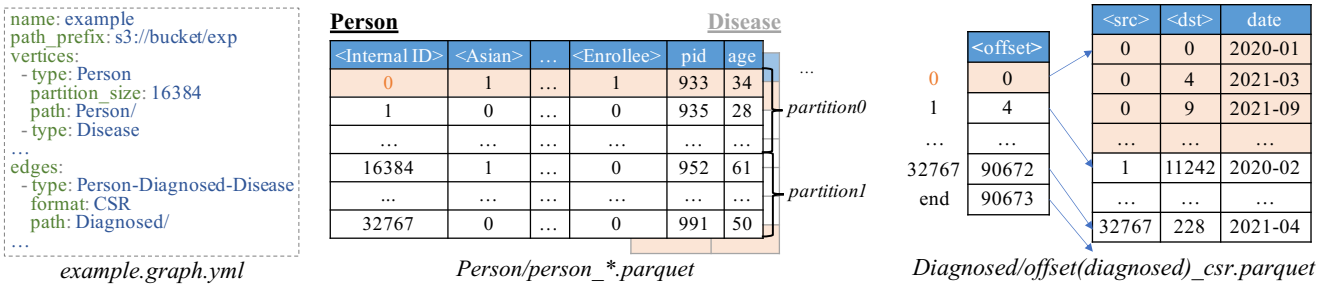
Vertex table. As depicted in Figure 4b, each row in the vertex table represents a unique vertex, identified by a 0-indexed internal ID, stored in the *<Internal ID>* column. When partitioning is enabled, for the i-th partition, its internal IDs start at partition_size $\times\, i$, and within each partition, IDs are sorted in ascending order. Bubbles[3] are allowed at the end of each partition, meaning the actual number of rows can be less than or equal to the partition size.

Property columns (*pid* and *age*) are named after their respective properties and hold the corresponding values with specified datatypes. In terms of labels, a set of candidate labels is defined for each vertex type. Then a vertex can have an arbitrary number of labels from the corresponding set. For example, the vertex type *Person* may have labels to represent ethnicity. For efficient storage and filtering of labels, GraphAr uses a binary representation to maintain each label in an individual column named with angle brackets, e.g., *<Asian>* and *<Enrollee>*. Additionally, advanced encoding/decoding techniques are applied, which will be discussed in Section 5.

Edge table. Edges are also organized and stored in Parquet files, similar to vertices. Figure 4c showcases the layout of the edge table for type *Person-Diagnosed-Disease*, where *Person* and *Disease* represent the source and destination vertex types, while *Diagnosed* signifies the classification of the relationships. Each edge is associated with the internal IDs of its source and destination vertices, stored in columns named *<src>* and *<dst>*. Edge properties, and optional partitioning, are handled in the same way as vertices.

Optimized access patterns for neighbor retrieval. The layout strategy in GraphAr leverages the columnar storage capabilities of Parquet to facilitate efficient graph traversal. Edges are sorted first by source vertex IDs and then by destination vertex IDs. This sorting strategy optimizes various access patterns. For row-wise access, the layout closely resembles the Coordinate List (COO) format, making it well-suited for edge-centric operations. On the other hand, an auxiliary index table, denoted as *<offset>*, is introduced to enable more efficient vertex-centric operations. The *<offset>* table aligns with the partitions in the vertex table, and when applied to the source

[3] "Bubbles" refer to the allowance for some ranges of internal IDs or edge segments not to correspond to any vertices or edges.



(a) Metadata. **(b) Vertex table of Person (and Disease).** **(c) Edge table of Person-Diagnosed-Disease.**

Figure 4: The metadata and data layout for the example graph in GraphAr.

vertices, facilitates retrieval patterns similar to Compressed Sparse Row (CSR). Likewise, a similar approach can be applied to enable Compressed Sparse Column (CSC)-like access. GraphAr allows for efficient bidirectional neighbor retrieval through two sorted tables for the same edge type. CSR, CSC, and COO are widely adopted for representing graphs, thus GraphAr ensures compatibility with existing graph-related systems.

These layout strategies are complemented by encoding and decoding optimizations (Section 4). Collectively, these strategies enhance both the data management and query capabilities of GraphAr.

3.3 Incorporation with Data Lakes

The design of GraphAr makes it especially well-suited for integration with data lakes, largely due to its reliance on widely adopted standards such as Parquet and YAML.

Data transformation and construction. The GraphAr format is essentially a specialized layout of Parquet files accompanied by a YAML metadata file. This enables the use of existing data processing frameworks like Apache Spark, Acero, and Hadoop, which can access various graph systems like Neo4j, TigerGraph and Nebula, or other types of database systems through their respective connectors. These frameworks can also ingest a multitude of data formats including logs, relational tables, JSON, and more. Such flexibility provides users with the ability to construct, transform, and store LPGs in data lakes from a wide array of data sources.

Downstream system integration. Since GraphAr is fundamentally based on Parquet and YAML, it is straightforward to use it as a data source for downstream systems. Many systems already have the capability to ingest Parquet files, making GraphAr a convenient and efficient data storage scheme.

Graph-specific optimizations. In addition to serving as a flexible storage format, GraphAr is also optimized for graph-specific operations. These optimizations, include advanced query pushdown techniques and other performance enhancements that are particularly useful for graph-specific tasks and queries within data lakes (see more from Sections 4 and 5).

4 EFFICIENT NEIGHBOR RETRIEVAL

In this section, we address the critical challenge of efficient neighbor retrieval in graphs. We leverage Parquet's data pages and introduce page-aligned collections (PAC) for streamlined neighbor identification. Additionally, we utilize delta encoding and introduce an innovative decoding strategy that leverages BMI and SIMD.

4.1 Workflow of Neighbor Retrieval

Parquet use data pages to match the data storage with the access granularity of underlying storage, where a page is the minimum unit of data that can be read from or written to the storage layer (Figure 2). Encoding and decoding are applied at the page level.

For LPG queries, a common operation is to retrieve specific property values of neighboring vertices, given a queried vertex, e.g., obtaining the *name* values of *Disease* vertices connected to a particular *Person* vertex. Assuming the CSR format utilized for storing edge table *Person-Diagnosed-Disease*, the typical workflow for this operation involves: 1) Using the *<offset>* index and *<dst>* column of the edge table to identify and fetch the first relevant page from the target vertex table (a page from the *name* column in the *Disease* table). This page contains at least one neighboring vertex pertinent to the query, and may also include other irrelevant vertices; 2) Selectively fetching the property values corresponding to the neighboring vertices within that page. This step is repeated iteratively for each subsequent page containing the targeted neighbors.

This workflow highlights the two primary steps during neighbor retrieval. The first is to identify which pages in the vertex table contain the neighboring vertices relevant to the query. The second is to fetch the relevant property values within each of these pages efficiently. Firstly, we formalize the neighbor retrieval operation:

Definition 1 (Neighbor Retrieval)**.** Given a vertex v, the operation of neighbor retrieval returns a data structure C representing the internal IDs of the neighboring vertices connected to v.

Requirement of C. The data structure C should facilitate the retrieval of the relevant property values of neighboring vertices, while minimizing space and processing overhead. Accordingly, we introduce the concept of page-aligned collections (PAC).

4.2 Page-Aligned Collections

Definition 2 (Page-aligned collections (PAC))**.** Given a column in vertex table that includes m pages, the PAC $C = [C_0, \ldots, C_{m-1}]$ is a list of up to m collections. Each C_i stores a set of internal IDs in the corresponding page. Non-empty collections in C are retained, while empty ones are omitted.

Intuition. Each collection C in PAC returned by neighbor retrieval corresponds to a data page of the target vertex table. To save space and avoid unnecessary processing, empty collections are omitted. This is based on the sparsity and locality of real-world graphs, which often results in irrelevant pages. Subsequently, the internal

IDs within each collection enable the retrieval of only the relevant property values through a *selection* process. The remaining challenges then involve 1) optimizing each collection's representation for quick value retrieval and 2) efficiently generating the PAC C.

A pioneering solution [52] underscores the transformation of indices into a bitmap representation to enable selection pushdown in columnar storages. To adopt this approach to efficiently retrieve the properties of neighbors, which addresses the first challenge, we adopt a bitmap representation B for each *non-empty* collection C in PAC, where $B[i] = 1$ indicates the existence of i-th element.

Figure 5a illustrates the neighbor retrieval of a source vertex to obtain a PAC, and Figure 5b demonstrates the usage of PAC to get the properties of its neighbors. For this illustrated example, only C_0 is non-empty, and B_0 is the bitmap representation of C_0. The bitmap representation can be used to facilitate the selection pushdown of vertex properties or labels, e.g., fetching the properties of *name* in the target vertex table *Disease*, for the neighbors of *Person* vertex$_0$. For the second challenge of efficiently generating PAC (i.e., Figure 5a), we utilize Parquet's delta encoding and a novel decoding strategy, which we detail in the following.

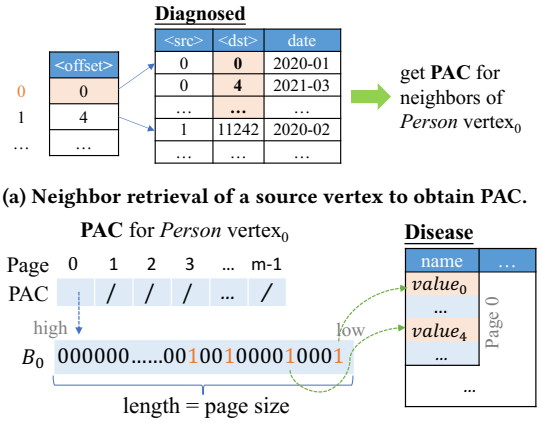
4.3 Delta Encoding

To compute the PAC C, the encoded internal IDs of neighboring vertices need to be loaded, sourced from the edge table. In a data lake scenario, where data can be stored remotely, the loading process can be more time-consuming than processing due to I/O limitations. To address this issue, we investigate the use of delta encoding for data compression, consequently reducing data load volume.

Delta encoding. Previous research [68, 77] has demonstrated that real-world graphs often exhibit both sparsity and locality. This means that while a vertex's neighbors might be spread across the entire graph, they are more likely to cluster within certain ID ranges. Such patterns arise from various factors, such as the inherent clustering in real-life graphs, where vertices within a cluster are more interconnected, and the methods used for data collection (e.g., crawlers or the organic/viral growth patterns of social networks like Facebook or TikTok). Systems [31, 77] have utilized such sparsity and locality to enable efficient partitioning or compression.

In GraphAr, such inherent locality, reinforced by our meticulous dual-key sorting of edges in the carefully designed layout, to enable incremental arrangement of internal IDs for a vertex's neighbors, serves as the basis for delta encoding, which is highly effective for both the *<src>* and the *<dst>* columns in the edge table. Delta encoding works by storing the deltas between consecutive values rather than each value separately. The deltas, which often have small values, can be stored more compactly, requiring fewer bits.

Implementations. We utilize Parquet's built-in support for delta encoding [50], which is implemented based on miniblocks. Each miniblock (with a size of 32 values) is binary packed using its own bit width, which should be a power of 2 for data alignment purposes. This design allows us to adapt to changes in the data distribution, as the bit width of each miniblock is dynamically adjusted to minimize storage consumption. According to our evaluation across various real-world graphs, as detailed in Section 6.2, the delta encoding technique can reduce the expected loaded data volume by 58.1% to 81.0% compared to without delta encoding. As a result, it brings an individual speedup of 2.7× for neighbor retrieval.



(a) Neighbor retrieval of a source vertex to obtain PAC.

(b) Using the PAC to get the properties of its neighbors.

Figure 5: An example of PAC and its usage.

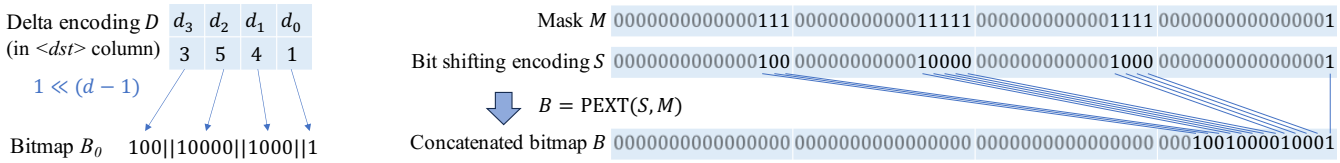
4.4 BMI-based Decoding

Challenges. Whiles delta encoding effectively reduces loading costs, it introduces additional decoding computation. Some existing works [50, 52, 59–61, 71] have explored the use of BMI (Bit Manipulation Instructions) and SIMD (Single Instruction, Multiple Data) to accelerate the data compression, decoding, scanning, or management. However, delta encoding involves data dependencies that make vectorization challenging. The decoding of the $(i+1)$-th neighbor depends on the prior decoding of the i-th neighbor.

In our context, the critical challenge is to generate the bitmap representation of PAC efficiently from the delta-encoded neighbor IDs, which are stored in the *<dst>* column of the edge table (as shown in Figure 6a). Existing techniques are not suitable for our context due to the data dependencies involved. However, by taking advantage of the sophisticated instruction sets offered by modern CPUs, we can exploit the functionalities of BMI together with SIMD operations to overcome this challenge, through an innovative decoding strategy.

Intuition. To ensure clarity, we initially consider a simplified scenario where each delta value is compressed to a 4-bit size. Conventionally, a two-step approach is used to decode the delta-encoded data, in which the current encoded value is added to the previously decoded ID to obtain the current ID, and then the bitmap is updated bit by bit based on the decoded IDs. However, our analysis reveals that this two-step process is redundant. By leveraging the bit-shifting encoding $1 << (d-1)$ for each delta value d, we can generate the bitmap by concatenating the bit-shifting encodings: $1 << (d_{n-1}-1)||\ldots||1 << (d_1-1)||1 << (d_0-1)$, where $d_0, d_1, \ldots, d_{n-1}$ represent n delta values, and $||$ represents the concatenation operator. This principle is visually depicted in Figure 6a.

Acceleration via BMI and SIMD. In practical implementation, the bit-shifting encoding is maintained using a fixed-length datatype, characterized by zero-padding on the left side. In our example, 16 bits are sufficient to accommodate the 4-bit deltas. The bit-shifting encodings of 4 values are stored in a 64-bit register, allowing for parallel generation and processing using SIMD instructions. Then, the focus shifts to the compaction of these encodings. Fortunately, the Parallel Bit Extract (PEXT) operation, a specialized CPU instruction in BMI, facilitates efficient aggregation of discrete bits from source



(a) Intuition. **(b) Implementation.**

Figure 6: An example of accelerating decoding via BMI.

positions into contiguous bits within the destination, governed by a selector mask. This process is illustrated in Figure 6b.

The subsequent challenge is to generate the required mask, achieved by deriving the i-th mask m_i from the i-th bit-shifting encoding s_i using the equation $m_i = (s_i << 1) - 1$. The advantage of this operation is its potential for convenient parallel execution, facilitated by direct manipulation of the mask sequence residing within a 64-bit register. Specifically, M can be acquired via the following steps: 1) a bitwise (in our example, 16 bits) shift of each s_i to the left by 1 bit, parallelized through SIMD instructions like _mm_slli_epi16; and 2) a bitwise subtraction of 1 from the result of the previous step, which can be accelerated via instructions like _mm_sub_epi16. All these SIMD instructions utilized in our implementations are widely available in modern CPUs, included in SSE2 (which we use) and more recent sets such as AVX2 and AVX-512.

In general, vectorization demonstrates greater efficiency when the bit width is smaller, as it allows for more significant parallelism. Our extensive evaluations have confirmed that the BMI-based decoding approach outperforms the default decoding approach in Parquet when the bit width is within 4 bits, with performance improvements ranging from 3.3% to 110%. Therefore, we utilize this BMI-based approach for miniblocks with a bit width of 1 to 4 bits, while resorting to the default delta decoding of Parquet for larger bit widths. The combination of data layout, delta encoding, and this adaptive decoding strategy results in an advanced topology management paradigm, enabling efficient neighbor retrieval.

5 OPTIMIZED LABEL FILTERING

Labels serve as a representation of the classification or characteristics of vertices in a graph. Filtering vertices by labels is a fundamental syntax in graph query languages, as it allows querying specific subsets of vertices. Existing approaches [12, 21] of fitting graphs into tabular data often treat labels as regular properties, encapsulating them within a string or list, as seen in Figure 3a. This approach overlooks the inherent differences between labels and properties, leading to inefficient label filtering, due to the need for decoding string representations and conducting string matching.

Recognizing the widespread use of labels as filter conditions and their unique nature, we develop a specialized format for labels leveraging binary representation and run-length encoding (RLE), for handling simple conditions. To support complex conditions introduced by user-defined functions that involves multiple labels, we enhance our methodology with a novel merge-based decoding algorithm, further improving efficiency and adaptability.

5.1 Handling Simple Conditions

We start by considering the simple condition that focuses on the existence of a single label. In essence, the existence or absence of a label can be effectively represented using binary notation, where the value 1 indicates the existence of the label and 0 indicates its absence, as demonstrated in Figure 4b. This binary representation offers two significant advantages: 1) it reduces the computational burden associated with decoding and matching as well as simplifies the filtering process as follows; 2) it enables efficient compression.

Definition 3 (Simple Condition Filtering)**.** Given a label l and an existence/absence indicator e, the simple condition label filtering returns the PAC C, where

$$\begin{cases} v \in C, & \text{if } v.label[l] = e \\ v \notin C, & \text{if } v.label[l] \neq e \end{cases} \tag{1}$$

Encoding. To compress consecutive runs of 0s or 1s, we utilize the technique of run-length encoding (RLE), which represents them as a single number. This run-length format naturally transforms the binary representation of a label into an interval-based structure. We then adopt a list P to define the positions of intervals. The i-th interval is represented by $[P[i], P[i+1])$, where $P[i]$ refers to the i-th element within P. Besides, it is required to record whether the vertices of the first interval $[P[0], P[1])$ contain the label or not, i.e., the first value. By leveraging this technique, the storage consumption of labels can be significantly reduced.

Decoding. Beyond efficient compression, the RLE approach seamlessly accommodates the decoding requirements for filter conditions. Specifically, to filter vertices with (or without) a specific label, we can simply select all odd intervals or all even intervals from the list P, based on the condition and the first value, instead of evaluating each vertex individually. It reduces the time complexity from $O(n)$ to $O(|P|)$, where n represents the number of vertices and $|P|$ represents the size of the interval list P. In real-world graphs, $|P| \ll n$ is often observed, due to the sparsity of labels and natural clustering of vertices with similar labels.

5.2 Extending to Complex Conditions

Expanding beyond the realm of simple label existence, we encounter the intricacies of dealing with complex conditions involving multiple labels. Consider a scenario where we need to find vertices with specific label combinations, such as the GQL pattern `MATCH (person:Asian&Enrollee)` (or in Cypher, `MATCH (person:Asian:Enrollee)`), which retrieves vertices labeled as Asian and Enrollee. A more complex pattern can be `MATCH (person:(Asian&!Enrollee)|Student)`, which retrieves vertices labeled as Asian but not Enrollee, or labeled as Student. To handle such scenarios, we employ user-defined functions (UDFs) to represent complex filter conditions. The UDF f takes a vertex v as input and returns a boolean value $f(v)$, indicating whether the vertex satisfies the condition or not. Formally, we define the complex condition filtering as follows.

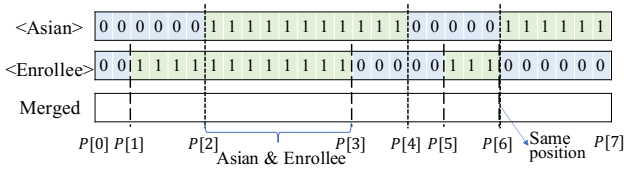


Figure 7: An example of merging intervals.

Definition 4 (Complex Condition Filtering)**.** Given a UDF f, the filtering returns the PAC C, where

$$\begin{cases} v \in C, \text{ if } f(v) = \text{true} \\ v \notin C, \text{ if } f(v) = \text{false} \end{cases} \tag{2}$$

The intuitive approach would be to tackle each vertex independently, decoding RLE into the binary representation. However, directly evaluating the UDF for every vertex proves impractical, as it retains the same complexity as the most straightforward approach.

Intuition. Inspired by the concept of discretization, two key questions arise: 1) Can we solely evaluate the condition for one representative vertex within each interval? 2) How can we efficiently identify these intervals where the encompassed vertices share the same labels? The affirmative answer to the first question emerges through the following theorem:

Theorem 1. *Consider k interval lists $P_0, P_1, \ldots, P_{k-1}$, where the vertices in $[P_i[j], P_i[j+1])$ share the same value for the i-th label. If an iterval $[s, e)$ is not broken by any position, i.e.,*

$$\nexists i \in [0, k), j \in [0, |P_i|) \quad s < P_i[j] < e, \tag{3}$$

the vertices within the interval $[s, e)$ have the same labels, i.e.,

$$\forall u, v \in [s, e), l \in [0, k) \quad u.label[l] = v.label[l]. \tag{4}$$

Consequently, it is sufficient to call the UDF for the vertex s alone, as for any vertex v in the interval $[s, e)$, v and s share the same labels, thus $f(v) = f(s)$.

Merge-based algorithm. Partitioning an interval into multiple segments proves unnecessary and counterproductive as it would escalate complexity. Therefore, our focus narrows down to intervals formed by existing positions, which also addresses the second question. To obtain the exact intervals, we can sort the positions in all interval lists $P_0, P_1, \ldots, P_{k-1}$. This sorting can be accelerated by leveraging the inherent order within the k lists, allowing for seamless merging of k sorted lists into one list P. Figure 7 demonstrates an example of interval determination for a complex condition containing two labels, Asian and Enrollee. Within the interval $[P[i], P[i+1])$, the vertices share identical labels, necessitating the UDF to be invoked solely for one representative vertex. Additionally, the presence of position $P[6]$ for both labels underscores the importance of merging to avoid redundant computations.

By employing innovative label treatment, interval-based encoding/decoding, and complex condition handling, GraphAr is able to achieve highly efficient label filtering.

6 EVALUATION

In this section, we evaluate GraphAr on various graphs, through micro-benchmarks and end-to-end graph query workloads. We also highlight GraphAr's potential to enhance performance and broaden the applicability of current graph processing systems.

Table 1: Statistics of the graphs in our evaluation.

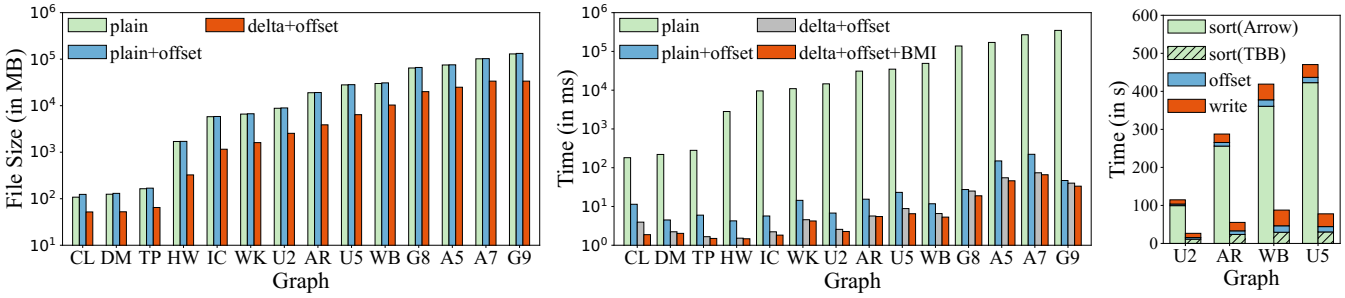
Abbr.	Graph	|V|	|E|
A5	Alibaba synthetic (scale 5)	75.0M	4.93B
A7	Alibaba synthetic (scale 7)	100M	6.69B
AR	arabic-2005 [34]	22.7M	1.27B
BL	bloom [20]	33.0K	29.7K
CF	com-friendster [51]	65.6M	1.81B
CI	citations [20]	264K	221K
CL	cont1-l [66]	1.92M	7.03M
DM	degme [66]	659K	8.13M
G8	Graph500-28 [14]	268M	4.29B
G9	Graph500-29 [14]	537M	8.59B
HW	hollywood-2009 [34]	1.14M	113M
OL	icij-offshoreleaks [20]	1.97M	3.27M
PP	icij-paradise-papers [20]	163K	364K
IC	indochina-2004 [34]	7.41M	384M
NM	network-management [20]	83.8K	181K
AX	ogbn-arxiv [46]	169K	1.17M
MA	ogbn-mag [46]	736K	21.1M
OS	openstreetmap [20]	71.6K	76.0K
PO	pole [20]	61.5K	105.8K
SF30	SNB Interactive SF-30 [37]	99.4M	655M
SF100	SNB Interactive SF-100 [37]	318M	2.15B
SF300	SNB Interactive SF-300 [37]	908M	6.29B
TP	tp-6 [66]	1.01M	10.7M
TT	twitter-trolls [20]	281K	493K
U2	uk-2002 [34]	18.5M	589M
U5	uk-2005 [34]	39.5M	1.85B
WB	webbase-2001 [34]	118M	2.01B
WK	wiki [49]	13.6M	437M

6.1 Experimental Setup

Platform. If not otherwise mentioned, our experiments are conducted on an Alibaba Cloud r6.6xlarge instance, equipped with a 24-core Intel(R) Xeon(R) Platinum 8269CY CPU at 2.50GHz and 192GB RAM, running 64-bit Ubuntu 20.04 LTS. The data is hosted on a 200GB PL0 ESSD with a peak I/O throughput of 180MB/s. Additional tests on other platforms and S3-like storage yield similar results. For timing metrics, we use single-threaded executions and report either average or distribution times based on multiple runs for accuracy. Exceptionally, the integration experiments utilize a cluster of 8 separate instances to emulate a distributed environment.

Baselines. GraphAr is developed in C++ on Apache Arrow [3], an open-source, high-performance library that supports columnar formats like Parquet and ORC. For the micro-benchmarks, we compare GraphAr against Arrow/Parquet (version 13.0.0), due to the popularity and high-performance of Parquet. Both GraphAr and the baseline follow Parquet's default configurations, which include a row group length of 1024×1024 and a 1MB page size. For end-to-end workloads, we compare GraphAr against widely-used frameworks including Apache Acero [2], Apache Pinot [8] and Neo4j (Community 5.21.0) [19]. In the integration experiments, we incorporate GraphAr into GraphScope [38], a widely-used graph processing system, and compare its integrated performance against GraphScope's original implementation.

Datasets. Table 1 summarizes the graphs we used, which span different sizes and domains. We also use synthetic graphs generated by data generators of the LDBC SNB [37] and Graph500 [14], both of which are widely recognized benchmarks. Additionally, we utilize graphs (A5 and A7) generated that mimic the characteristics of graphs in Alibaba's e-commerce production environment.



(a) Storage consumption for topology. **(b) Neighbor retrieval time.** **(c) Transformation time.**

Figure 8: Micro-benchmark of neighbor retrieval.

6.2 Micro-Benchmark of Neighbor Retrieval

We evaluate GraphAr's optimizations in neighbor retrieval through micro-benchmarks on selected graphs characterized by a large edge set ($|E|$). Our results substantiate its efficacy in enhancing storage efficiency and retrieval performance.

Storage efficiency. We compare GraphAr with baseline methods by measuring the storage consumed by encoded *Parquet* files that store the graph's topological data. Two baseline approaches are considered: 1) "plain", which employs plain encoding for the source and destination columns, and 2) "plain + offset", which extends the "plain" method by sorting edges and adding an offset column to mark each vertex's starting edge position. As Figure 8a depicts, the inclusion of offsets results in a modest increase in storage requirements, with space usage growing by 0.5% to 14.8%, as the number of vertices is typically much smaller than the number of edges.

GraphAr leverages delta encoding for source and destination columns and plain encoding for offsets. The result is a notable storage advantage: on average, it requires only 29.2% of the storage needed by the baseline "plain + offset". This efficiency in storage is particularly beneficial for query performance, as data lake queries are often I/O-bound. The transition from storage efficiency to retrieval performance is elaborated further in the next experiment.

Performance of neighbor retrieval. To evaluate GraphAr's efficiency in neighbor retrieval, we query vertices with the largest degree in selected graphs, maintaining edges in CSR-like or CSC-like formats depending on the degree type. Figure 8b shows that GraphAr significantly outperforms the baselines, achieving an average speedup of 4452× over the "plain" method, 3.05× over "plain + offset", and 1.23× over "delta + offset". These gains are attributed to the offset integration and delta encoding, as well as our BMI-based decoding. The offset integration alone accounts for an average speedup of 1993×, and delta encoding provides an additional 2.48× speedup. Our innovative decoding method, which leverages BMI and SIMD, further enhances performance within this optimized context, achieving a 1.23× speedup on top of "delta + offset".

Performance of data transformation. Given that GraphAr is designed for storing LPGs in data lakes, the efficiency of converting original graph data into the GraphAr format is crucial. Graphs generally have significantly more edges than vertices and GraphAr employs CSR/CSC-like layouts requiring edge sorting. Thus, generating topological data becomes the most time-intensive part. To assess the overhead, we analyze the time to convert four real-world graphs (U2, AR, WB, and U5), each initially in the form of Arrow Tables, a standardized in-memory format in big data systems. Figure 8c illustrates the time breakdown. The process involves three steps: 1) sorting the edges, using Arrow's *order_by* operator, labeled as "sort (Arrow)"; 2) generating vertex offsets, labeled as "offset"; and 3) writing the sorted and offset data into Parquet files with specific encoding, labeled as "write". The sorting step is most time-consuming, which has an average time complexity of $O(|E|log|E|)$ when executed sequentially, while the steps of generating offset and writing with encoding both have a lower time complexity of $O(|E|)$. For further optimization, we leverage the parallel sorting algorithms provided by Intel(R) Threading Building Blocks [15], which significantly reduces the sorting time, labeled as "sort (TBB)". By employing 24 threads in our test setup, the sorting time is only 8.9% of the original, on average. Given this transformation is a one-time, offline operation that substantially reduces future data retrieval times, the associated overhead—which is within 1 minute for generating topological data for over 1 billion edges—is acceptable.

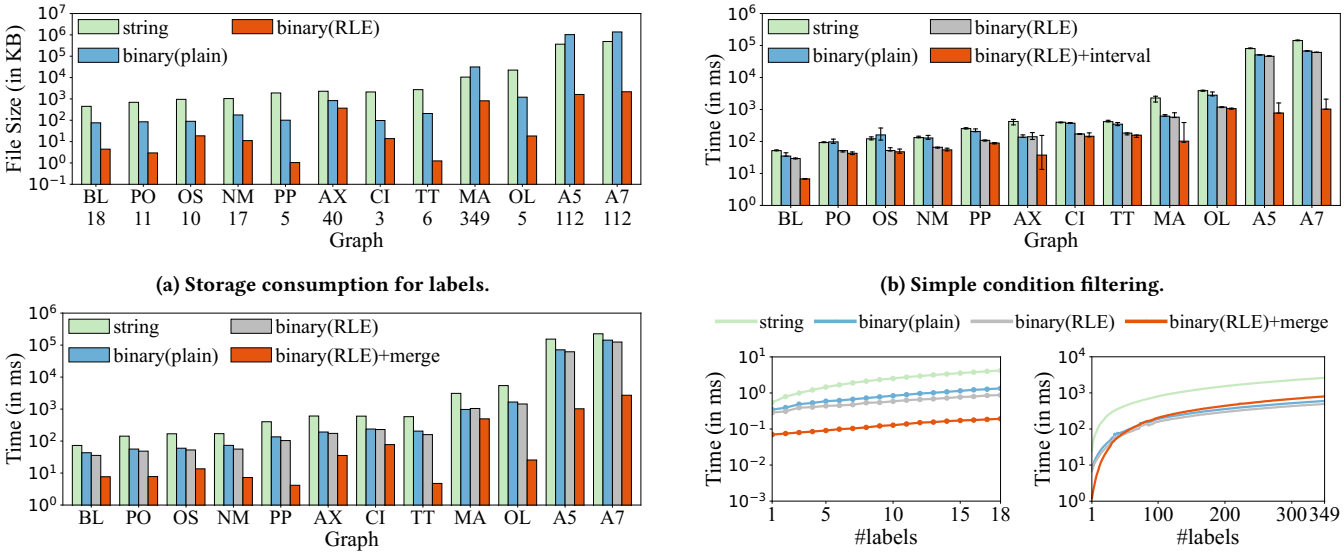
6.3 Micro-Benchmark of Label Filtering

This section evaluates GraphAr's efficiency in storing and filtering vertex labels. We employ datasets from OGB [46], Neo4j [20] and Alibaba's synthetic data generator, which feature property graphs with multiple vertex labels, with the number of labels (ranging from 3 to 349) indicated under the graph name in Figure 9a.

Storage efficiency. We assess storage efficiency by measuring the size of encoded Parquet files used for storing vertex labels. Two baseline methods serve for comparison: The first, termed "string", concatenates all labels of a vertex into a single string column using *BYTE_ARRAY* datatype and plain encoding. The second, named "binary (plain)", represents each label in a separate binary column using *BOOLEAN* datatype and plain encoding. Our approach, denoted as "binary (RLE)", further optimizes this by utilizing RLE.

As shown in Figure 9a, our RLE-based method substantially outperforms the baselines, requiring on average only 2.5% and 8.4% of the storage space compared to the "string" and "binary (plain)" methods, respectively. We exclude dictionary encoding of Parquet despite its potential storage gains over the "string" baseline, because it can slow down decoding by up to 10×.

Performance of simple condition filtering. Recognizing that filtering based on simple conditions represents the cornerstone operation in graph query languages, we prioritize evaluating this operation. For each graph, we perform experiments where we consider each label individually as the target label for filtering, and



(a) Storage consumption for labels.

(b) Simple condition filtering.

(c) Complex condition filtering.

(d) Scaling the number of labels, tested on BL (left) and MA (right).

Figure 9: Micro-benchmark of label filtering.

determine the vertices with that label. For accuracy, each experiment is repeated 100 times and the total execution time is reported.

Figure 9b illustrates the results, demonstrating that GraphAr's method significantly improves the performance of label filtering based on simple conditions. The most straightforward approach, "string", which involves decoding the string of labels and conducting matching for each vertex, is the slowest. The "binary (plain)" method separates labels into individual columns and utilizes a binary representation, while the 'binary (RLE)" method further optimizes the encoding by using RLE. However, both of these methods still require evaluating each vertex. In contrast, our method of "binary (RLE) + interval", simply selects all satisfied intervals.

For each graph, we report the middle value of the execution time among filtering each label as the height of the bar, with the error bar representing the range of execution time. Our method may have a large range on some graphs (AX, MA) due to the varying encoding efficiency (i.e., the number of intervals generated) for different columns. However, since the number of intervals is not larger than the number of vertices in any case, our method consistently outperforms the baselines. On average, it achieves a speedup of 14.8× over the "string" method, 8.9× over the "binary (plain)" method, and 7.4× over the "binary (RLE)" method.

Performance of complex condition filtering. We also assess the performance of label filtering based on complex conditions. For graphs obtained from Neo4j, we first refer to the provided documentation to identify a filtering operation that involves two labels. If not provided, we create a condition by combining two related labels using either the logical *AND* operator (if there are vertices satisfying the condition) or *OR* (otherwise) to reflect real-world semantics. For other graphs, we combine the first two labels by *OR* as the filtering condition. Figure 9c presents the performance of different methods, measured as the total execution time of 100 runs. The results demonstrate that GraphAr performs the best for all test cases. Further analysis reveals that this improvement is attributed to the binary representation (as seen in the comparison between "binary (plain)" and "string"), and utilization of RLE (as seen in the comparison between "binary (RLE)" and "binary (plain)"). The merge-based decoding method yields the largest gain, where "binary (RLE) + merge" outperforms "binary (RLE)" by up to 60.5×.

Scale up the number of labels. Figure 9d illustrates the average execution time of filtering conditions with varying numbers of labels, focusing on BL and MA, which are selected from different datasets (Neo4j and OGB) and have a relatively large number of labels. We test the filtering with i labels by combining the first i labels through *OR* as the condition. As shown in the figure, GraphAr consistently outperforms others on BL. While on MA, it performs best when the number of involved labels is no more than 40. As the number of labels continuously increases, it performs worse than the baseline "binary (RLE)", which is due to the number of merged intervals also increasing. In the worst case, the UDF is called for each vertex, means any two consecutive vertices have different labels. Considering the overhead of merging intervals, our method may perform worse than directly evaluating each vertex. Fortunately, our investigation of real-world workloads reveals that the number of filtered labels in user-written queries is often limited, e.g., in the Neo4j documentation examples [20], the filtering involves at most 5 labels. This suggests that our method is highly promising.

6.4 Storage Media

We assess the efficiency of GraphAr across various storage media: local in-memory *tmpfs*, *ESSD* (an Alibaba Cloud virtualized elastic block device), and S3-like Object Store Service (*OSS*). The graph used is SF100, with specifically focus on the *comment* vertex type and *comment_hasTag_tag* edge type. Table 2 encapsulates the efficiency of GraphAr across different storage media. These results demonstrates that GraphAr is not only efficient but also robust, delivering consistently high performance, with speedups of 88× to 154× for neighbor retrieval and 2.7× to 11× for label filtering.

Table 2: Performance comparison across storage media.

Storage	Neighbor Retrieval (s)		Label Filtering (s)	
	Plain	GraphAr	String	GraphAr
tmpfs	6.446	0.053	3.984	1.489
ESSD	16.41	0.106	19.06	1.746
OSS	189.4	2.145	252.8	26.22

6.5 End-to-end Graph Query Workloads

To demonstrate the practicality of GraphAr in real-world scenarios, we conduct a performance evaluation using end-to-end workloads from the LDBC SNB benchmark [37, 67]. Although the benchmark specifies vertex/edge types, it does not explicitly define the labels. However, we are able to identify certain vertex types that are *static* (e.g., *tagclass* and *place*), which have a fixed and very small vertex set size that does not scale with the graph size. On the other hand, vertex types like *comment* and *person* are considered *dynamic*. Based on this observation, we can treat information related to *static* types as labels for *dynamic* types in GraphAr, for example, all tag classes of a comment are attached as labels for the corresponding *comment* vertex. Similar strategies are also adopted by graph databases [44].

For evaluation, graphs at different scales (listed in Table 1) are generated using the LDBC SNB data generator. These graphs are then converted into GraphAr format, with the vertex labels attached as described above. Upon investigating the benchmark, including 7 short and 14 complex interactive queries, as well as 20 business intelligence queries, we find that neighbor retrieval is frequently encountered, involved in approximately 90% of the queries. Considering the aforementioned label organization, label filtering is also common, involved in approximately 50% queries.

Query implementations. The evaluation focuses on three representative queries, with the required parameters set according to the reference implementations [17, 18]. **IS-3** (*interactive-short-3*) aims to find all the friends of a given person and return their information. It exemplifies the common pattern of querying neighboring vertices and retrieving associated properties. **IC-8** (*interactive-complex-8*) is more complex as it involves traversing multiple hops from the starting vertex. Lastly, the **BI-2** (*business-intelligence-2*) query involves finding and counting the messages associated with tags within a specific tag class, thus requiring vertex filtering by labels.

We develop hand-written implementations for each query based on GraphAr, which utilize the data organization and specifically prioritize two essential operations: neighbor retrieval and label filtering. Our implementation adheres to the official reference implementations [17, 18] to ensure equivalence to the original queries.

We then implement these queries in Acero [2], which is a powerful C++ library integrated into Apache Arrow for analyzing large streams of data. It offers a comprehensive set of operators such as *scan*, *filter*, *project*, *aggregate*, and *join*, among others. Moreover, Acero supports taking Parquet as the data source and enables the pushdown of predicates, making it a strong baseline for comparison with GraphAr. Despite our best efforts to optimize it, we do not perform data re-organization or utilize GraphAr's encoding/decoding optimizations for this implementation based on Acero.

We also include two additional baselines: Apache Pinot [8], a real-time OLAP datastore used by LinkedIn for processing and querying large social networks, and Neo4j [19], a main graph database utilizing the Cypher query language. While both are widely-used, they are not natively designed for data lakes and require an Extract-Transform-Load (ETL) process for integration.

Table 3: Query execution times (in seconds), with the format of *Pinot (P), Neo4j (N), Acero (A), GraphAr (G)*. "OM" denotes failed execution due to out-of-memory errors.

	SF30				SF100				SF300			
	P	N	A	G	P	N	A	G	P	N	A	G
ETL	6024	390	—	—	17726	2094	—	—	OM	9122	—	—
IS-3	1.00	0.30	0.16	**0.01**	6.59	2.09	0.48	**0.01**	OM	4.12	1.39	**0.03**
IC-8	1.35	**0.37**	72.2	3.36	8.43	**1.26**	246	6.56	OM	**2.98**	894	23.3
BI-2	125	45.0	67.7	**4.30**	3884	1101	232	**16.3**	OM	6636	756	**50.0**

Performance comparison. Table 3 presents a comparison of end-to-end performance, clearly demonstrating that the implementation based on GraphAr significantly outperforms Acero, achieving an average speedup of 29.5×. A closer analysis of the results reveals that the performance gains stem from the following factors: 1) data layout design and encoding/decoding optimizations we proposed, to enable efficient neighbor retrieval (IS-3, IC-8, BI-2) and label filtering (BI-2), as demonstrated in micro-benchmarks; 2) bitmap generation during the two critical operations, which can be utilized in subsequent selection steps (IS-3, IC-8, BI-2).

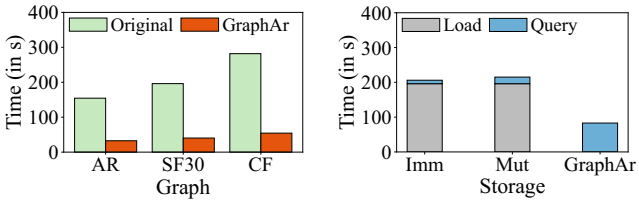
As for Pinot and Neo4j, their end-to-end performance is often dominated by extensive ETL processes, in the context of data lakes, as the results show. GraphAr performs best on IS-3, which is a single-hop query, and BI-2, where GraphAr utilizes label filtering for the early elimination of irrelevant data. While on IC-8, GraphAr is outperformed by Neo4j due to the query involving traversing multiple hops, which results in a significant volume of data loading for both Acero and GraphAr. Nevertheless, GraphAr not only offers efficient query performance but also eliminates the ETL overhead, potentially avoids out-of-memory errors that may occur. Thus, GraphAr provides a more practical solution for data lake scenarios.

6.6 Integration with Graph Processing Systems

One of the advantages of GraphAr is its compatibility with existing graph processing systems. To demonstrate this, we have integrated GraphAr into GraphScope [38], a popular distributed system designed to meet a diverse range of graph computing needs. GraphAr is utilized as the archive format for persistent storage. It also serves as an accessible storage backend for executing infrequent queries in an out-of-core manner.

Serve as the archive format. We first compare the performance of building graphs in GraphScope using 8 nodes, from external storages in GraphAr format against the baseline, where the datasets are in CSV format, sourced directly from the data providers. The findings illustrated in Figure 10a indicate that GraphAr significantly outperforms the baseline, achieving an average speedup of 4.9×. This improvement can be attributed to GraphAr's efficient encoding strategies that reduce the data volume to be loaded, as well as its optimized data organization and layout, which facilitate a faster in-memory graph construction within GraphScope.

Serve as a storage backend. Leveraging the capabilities for graph-related querying, the graph query engine within GraphScope can execute queries directly on the GraphAr data in an out-of-core manner. We evaluate the performance of GraphScope with GraphAr as the storage backend, and compare the average querying time



(a) Graph loading time. **(b) BI queries execution time.**

Figure 10: GraphScope's performance w/ and w/o GraphAr.

of BI queries on SF30 with two baseline scenarios wherein GraphScope relies on its native in-memory storage options, specifically the immutable ("Imm") and mutable ("Mut") variants. Figure 10b demonstrates that although the querying time with GraphAr exceeds that of the in-memory storages, attributable to intrinsic I/O overhead, it significantly surpasses the process of loading and then executing the query, by 2.4× and 2.5×, respectively. This indicates that GraphAr is a viable option for executing infrequent queries.

Application scenarios. This integration demonstrates the potential benefits of GraphAr in improving the efficiency of graph processing systems. In summary, its application scenarios include: 1) *Data loading:* GraphAr can significantly reduce graph loading times, making it an ideal choice for external storage formats; 2) *Out-of-core queries:* GraphAr can serve as a storage backend for executing graph queries in an out-of-core manner. It is particularly beneficial for infrequent queries that access only a portion of the graph data, eliminating the need for full in-memory graph representation. It also enables querying graphs that exceed the capacity of available memory. In other scenarios, such as: 1) real-time queries requiring low latency and frequent execution, or 2) graph analytics algorithms involving iterative computations across the entire graph (e.g., PageRank), GraphAr might not be the optimal direct storage solution and in-memory storage options are more suitable.

Summary. In summary, GraphAr has been demonstrated to be a highly effective storage scheme for LPGs in data lakes:

- **Storage efficiency:** GraphAr remarkably reduces storage requirements, using only 29.2% of the storage compared to baseline methods for storing topology, and as low as 2.5% for label storage on average.
- **Query performance:** GraphAr significantly outpaces the baselines in retrieval time, achieving an average speedup of 4452× for neighbor retrieval, and an average speedup of 14.8× for simple label filtering, as observed in micro-benchmarks on the *ESSD* storage.
- **Storage media:** Evaluations indicate seamless compatibility across various storage layers like *tmpfs*, *ESSD*, and *oss*, all achieving high speedup of 88× to 154× for neighbor retrieval and 2.7× to 11× for label filtering.
- **Real-world relevance:** In end-to-end workloads using the LDBC SNB benchmark, GraphAr shows an average speedup of 29.5× over the Acero baseline, substantiating its practical utility in real-world scenarios.
- **Compatibility:** GraphAr is seamlessly integrated into a widely-used graph processing system GraphScope, enhancing its graph loading efficiency by 4.9×, and accelerating its infrequent query execution with a speedup of 2.4×.

Collectively, these results validate GraphAr as a robust, storage-efficient, and high-performance solution for both academic research and industrial applications.

7 RELATED WORK

File formats in data lakes. The data lake ecosystem encompasses various common file formats, including CSV, JSON, Protocol Buffers, HDF5 [24], AVRO [4], ORC [6], and Parquet [7]. While these formats support various optimizations that benefit both tables and graphs, they fall short in comprehensively representing LPG semantics and supporting graph-specific operations.

Data management in data lakes. The popularity of data lakes has led to efforts aimed at enhancing their architecture and data management [22, 30, 56, 57, 64]. As for LPG management in data lakes, LinkedIn uses Apache Gobblin [5] for data ingestion and employs Apache Spark [75] and Apache Pinot [8] for processing large graph datasets representing the social network. Graph-specific querying frameworks like Neo4j [19] and Apache TinkerPop [9] are also widely-used, and they integrate with data lakes via ETL processes. These endeavors primarily focus on managing existing data within data lakes and are distinct from GraphAr. GraphAr, on the other hand, can be considered as a new storage format with unique features tailored for LPGs. It can be leveraged by these works to further extend the utility and capabilities of data lakes.

Graph file formats. Certain formats are designed for graph [13, 29, 32, 45] and RDF (Resource Description Framework) data [23, 53]. However, their primary focus is to describe or exchange data in a standardized manner, e.g., utilizing XML, and are not optimized for storage and retrieval purposes. The lack of encoding, compression and push-down optimizations can lead to far inferior performance, making them less suitable for managing LPGs in data lakes.

Graph-related databases. Some databases [16, 19, 36, 39] are designed to store and manage graph data. There are also efforts focus on optimizing graph-related queries [54, 58, 62, 76]. While they offer various graph-related features, they primarily focus on in-memory mutable data management, operating at a higher level compared to GraphAr. GraphAr, with its format compatible with the LPG model, can be utilized as an archival format for graph databases.

Operation pushdown. Some previous works [10, 52, 70–73] aim to develop high-performance operators on storage formats of either column-oriented or row-oriented. These works and GraphAr share the same goal of improving pushdown operators and leveraging CPU instructions like SIMD and BMI. However, these works mainly focus on operations related to relational data, such as scan, select, and filter based on properties. In contrast, GraphAr specifically focuses on two graph-specific operations.

8 CONCLUSION

In conclusion, this paper introduces GraphAr as an efficient and specialized storage scheme for graph data in data lakes. GraphAr focuses on preserving LPG semantics and supporting graph-specific operations, resulting in notable performance improvements in both storage and query efficiency over existing formats designed for relational tables. The evaluation results validate the effectiveness of GraphAr and highlight its potential as a crucial component in data lake architectures.

REFERENCES

[1] 2023. Information technology – Database languages – SQL. ISO/IEC 9075:2023.

[2] 2024. Acero: A C++ streaming execution engine. https://arrow.apache.org/docs/cpp/streaming_execution.html. Accessed on 2024-12-17.

[3] 2024. Apache Arrow. https://arrow.apache.org/. Accessed on 2024-12-17.

[4] 2024. Apache AVRO. https://avro.apache.org/. Accessed on 2024-12-17.

[5] 2024. Apache Gobblin. https://gobblin.apache.org/. Accessed on 2024-12-17.

[6] 2024. Apache ORC File Format. https://orc.apache.org/. Accessed on 2024-12-17.

[7] 2024. Apache Parquet File Format. https://parquet.apache.org/. Accessed on 2024-12-17.

[8] 2024. Apache Pinot. https://pinot.apache.org/. Accessed on 2024-12-17.

[9] 2024. Apache TinkerPop. https://tinkerpop.apache.org/. Accessed on 2024-12-17.

[10] 2024. AWS S3 Select. https://aws.amazon.com/cn/blogs/aws/s3-glacier-select/. Accessed on 2024-12-17.

[11] 2024. Cypher Query language. https://neo4j.com/developer/cypher/. Accessed on 2024-12-17.

[12] 2024. Export Neo4j Data to CSV. https://neo4j.com/labs/apoc/4.4/export/csv/. Accessed on 2024-12-17.

[13] 2024. GEXF File Format. https://gexf.net/. Accessed on 2024-12-17.

[14] 2024. Graph 500 Benchmarks. https://graph500.org/. Accessed on 2024-12-17.

[15] 2024. Intel(R) Threading Building Blocks. https://github.com/wjakob/tbb. Accessed on 2024-12-17.

[16] 2024. JanusGraph: an open-source, distributed graph database. https://janusgraph.org/. Accessed on 2024-12-17.

[17] 2024. LDBC SNB Business Intelligence (BI) workload implementations. https://github.com/ldbc/ldbc_snb_bi. Accessed on 2024-12-17.

[18] 2024. LDBC SNB Interactive workload implementations. https://github.com/ldbc/ldbc_snb_interactive_impls. Accessed on 2024-12-17.

[19] 2024. Neo4j Graph Database and Analytics. https://neo4j.com/. Accessed on 2024-12-17.

[20] 2024. Neo4j Graph Examples. https://github.com/neo4j-graph-examples. Accessed on 2024-12-17.

[21] 2024. Neo4j Spark Connector. https://neo4j.com/docs/spark/current/reading/. Accessed on 2024-12-17.

[22] 2024. PuppyGraph: A Cloud-Native Graph Data Lakehouse. https://puppygraph.com/. Accessed on 2024-12-17.

[23] 2024. RDF Formats. https://www.w3.org/TR/. Accessed on 2024-12-17.

[24] 2024. The HDF5 Library and File Format. https://www.hdfgroup.org/solutions/hdf5/. Accessed on 2024-12-17.

[25] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, et al. 2023. PG-Schema: Schemas for property graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[26] Dmitry Anikin, Oleg Borisenko, and Yaroslav Nedumov. 2019. Labeled Property Graphs: SQL or NoSQL?. In *2019 Ivannikov Memorial Workshop (IVMEM)*. 7–13. https://doi.org/10.1109/IVMEM.2019.00007

[27] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał undefinedwitakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[28] Nico Baken. 2020. Linked data for smart homes: Comparing RDF and labeled property graphs. In *LDAC2020—8th Linked Data in Architecture and Construction Workshop*. 23–36.

[29] Vladimir Batagelj and Andrej Mrvar. 2001. Pajek—analysis and visualization of large networks. In *International Symposium on Graph Drawing*. Springer, 477–478.

[30] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2326–2339. https://doi.org/10.1145/3514221.3526054

[31] P. Boldi and S. Vigna. 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th International Conference on World Wide Web* (New York, NY, USA) *(WWW '04)*. Association for Computing Machinery, New York, NY, USA, 595–602. https://doi.org/10.1145/988672.988752

[32] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, and M. Scott Marshall. 2002. GraphML Progress Report Structural Layer Proposal. In *Graph Drawing*, Petra Mutzel, Michael Jünger, and Sebastian Leipert (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 501–512.

[33] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[34] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[35] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2246–2258. https://doi.org/10.1145/3514221.3526057

[36] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2019. TigerGraph: A Native MPP Graph Database. *CoRR* abs/1901.08248 (2019). arXiv:1901.08248 http://arxiv.org/abs/1901.08248

[37] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2723372.2742786

[38] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, Youyang Yao, Qiang Yin, Wenyuan Yu, Jingren Zhou, Diwen Zhu, and Rong Zhu. 2021. GraphScope: a unified engine for big graph processing. *Proc. VLDB Endow.* 14, 12 (July 2021), 2879–2892. https://doi.org/10.14778/3476311.3476369

[39] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2022. Kùzu Database Management System Source Code. https://github.com/kuzudb/kuzu. Accessed on 2024-12-17.

[40] Avrilia Floratou, Umar Farooq Minhas, and Fatma Özcan. 2014. Sql-on-hadoop: Full circle back to shared-nothing database architectures. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1295–1306.

[41] Nadime Francis, Amelie Gheerbrant, Paolo Guagliardo, Libkin Leonid, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoč. 2023. A Researcher's Digest of GQL. In *26th International Conference on Database Theory (ICDT 2023)*, Vol. 255. Ioannina, Greece. https://doi.org/10.4230/LIPIcs.ICDT.2023.1

[42] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1433–1445. https://doi.org/10.1145/3183713.3190657

[43] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 599–613.

[44] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Columnar Storage and List-Based Processing for Graph Database Management Systems. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2491–2504. https://doi.org/10.14778/3476249.3476297

[45] Michael Himsolt. 2010. GML: A portable Graph File Format. https://api.semanticscholar.org/CorpusID:5806388. Accessed on 2024-12-17.

[46] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2021. Open Graph Benchmark: Datasets for Machine Learning on Graphs. (2021). arXiv:2005.00687 [cs.LG] https://arxiv.org/abs/2005.00687

[47] Todor Ivanov and Matteo Pergolesi. 2020. The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet. *Concurrency and Computation: Practice and Experience* 32, 5 (2020), e5523.

[48] Pwint Phyu Khine and Zhao Shun Wang. 2018. Data lake: a new ideology in big data era. In *ITM web of conferences*, Vol. 17. EDP Sciences, 03025.

[49] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on World Wide Web* (Rio de Janeiro, Brazil) *(WWW '13 Companion)*. Association for Computing Machinery, New York, NY, USA, 1343–1350. https://doi.org/10.1145/2487788.2488173

[50] D. Lemire and L. Boytsov. 2013. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (may 2013), 1–29. https://doi.org/10.1002/spe.2203

[51] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. Accessed: 2024-12-17.

[52] Yinan Li, Jianan Lu, and Badrish Chandramouli. 2023. Selection Pushdown in Column Stores using Bit Manipulation Instructions. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[53] Miguel A. Martínez-Prieto, Mario Arias, and Javier Fernández. 2012. Exchange and Consumption of Huge RDF Data, Vol. 7295. 437–452. https://doi.org/10.1007/978-3-642-30284-8_36

[54] Amine Mhedhbi and Semih Salihoğlu. 2022. Modern techniques for querying graph-structured relations: foundations, system implementations, and open challenges. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3762–3765.

[55] Natalia Miloslavskaya and Alexander Tolstoy. 2016. Big data, fast data and data lake concepts. *Procedia Computer Science* 88 (2016), 300–305.

[56] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. 2019. Data Lake Management: Challenges and Opportunities. *Proc. VLDB Endow.* 12, 12 (aug 2019), 1986–1989. https://doi.org/10.14778/3352063.3352116

[57] Max Neunhöffer. 2024. Fishing for graphs in a Hadoop data lake. https://www.oreilly.com/content/fishing-for-graphs-in-a-hadoop-data-lake/. Accessed on 2024-12-17.

[58] Dung Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q Ngo, Christopher Ré, and Atri Rudra. 2015. Join processing for graph patterns: An old dog with new tricks. In *Proceedings of the GRADES'15*. 1–8.

[59] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1493–1508. https://doi.org/10.1145/2723372.2747645

[60] Orestis Polychroniou and Kenneth A. Ross. 2019. Towards Practical Vectorized Analytical Query Engines. In *Proceedings of the 15th International Workshop on Data Management on New Hardware* (Amsterdam, Netherlands) *(DaMoN'19)*. Association for Computing Machinery, New York, NY, USA, Article 10, 7 pages. https://doi.org/10.1145/3329785.3329928

[61] Orestis Polychroniou and Kenneth A. Ross. 2020. VIP: A SIMD Vectorized Analytical Query Engine. *The VLDB Journal* 29, 6 (jul 2020), 1243–1261. https://doi.org/10.1007/s00778-020-00621-w

[62] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[63] Raghu Ramakrishnan, Baskar Sridharan, John R. Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, Neil Sharman, Zee Xu, Youssef Barakat, Chris Douglas, Richard Draves, Shrikant S. Naidu, Shankar Shastry, Atul Sikaria, Simon Sun, and Ramarathnam Venkatesan. 2017. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 51–63. https://doi.org/10.1145/3035918.3056100

[64] Franck Ravat and Yan Zhao. 2019. Metadata management for data lakes. In *New Trends in Databases and Information Systems: ADBIS 2019 Short Papers, Workshops BBIGAP, QAUCA, SemBDM, SIMPDA, M2P, MADEISD, and Doctoral Consortium, Bled, Slovenia, September 8–11, 2019, Proceedings 23*. Springer, 37–44.

[65] Marko A. Rodriguez. 2015. The Gremlin Graph Traversal Machine and Language. *CoRR* abs/1508.03843 (2015). arXiv:1508.03843 http://arxiv.org/abs/1508.03843

[66] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[67] Gábor Szárnyas, Jack Waudby, Benjamin A Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proceedings of the VLDB Endowment* 16, 4 (2022), 877–890.

[68] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. 2011. The Anatomy of the Facebook Social Graph. (2011). arXiv:1111.4503 [cs.SI] https://arxiv.org/abs/1111.4503

[69] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: A Property Graph Query Language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems* (Redwood Shores, California) *(GRADES '16)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2960414.2960421

[70] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. 2013. Vectorizing Database Column Scans with Complex Predicates. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS 2013), Riva del Garda, Trento, I, August 26, 2013*. 1–12.

[71] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-Scan: Ultra Fast in-Memory Table Scan Using on-Chip Vector Processing Units. *Proc. VLDB Endow.* 2, 1 (aug 2009), 385–394. https://doi.org/10.14778/1687627.1687671

[72] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2101–2113. https://doi.org/10.14778/3476249.3476265

[73] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS Using S3 Computation. In *2020 IEEE 36th International Conference on Data Engineering*. 1802–1805. https://doi.org/10.1109/ICDE48307.2020.00174

[74] Matei Zaharia, Ali Ghodsi 0002, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

[75] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[76] Kangfei Zhao and Jeffrey Xu Yu. 2017. All-in-One: Graph Processing in RDBMSs Revisited. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1165–1180. https://doi.org/10.1145/3035918.3035943

[77] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: a computation-centric distributed graph processing system. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 301–316.