GES: High-Performance Graph Processing Engine and Service in Huawei (Technical Report)

Sen Gao*
Huawei & SJTU, China
gaosen13@huawei.com

Jianwen Zhao*
Huawei, China
zhaojianwen3@huawei.com

Hao Zhang†
Huawei, China
zhanghao687@huawei.com

Shixuan Sun†
Shanghai Jiao Tong University, China
sunshixuan@sjtu.edu.cn

Chen Liang
Huawei, China
liangchen28@huawei.com

Gongye Chen
Huawei, China
chengongye@huawei.com

Wenliang Zhang
Huawei, China
zhangwenliang14@huawei.com

Bo Ren
Huawei, China
jerous.ren@huawei.com

Chao Liu
Huawei, China
liuchao298@huawei.com

Chenyi Zhang
Huawei, China
zhangchenyi2@huawei.com

Quan Chen
Shanghai Jiao Tong University, China
chen-quan@sjtu.edu.cn

Chao Li
Shanghai Jiao Tong University, China
lichao@cs.sjtu.edu.cn

Jingwen Leng
Shanghai Jiao Tong University, China
leng-jw@sjtu.edu.cn

Minyi Guo
Shanghai Jiao Tong University, China
guo-my@cs.sjtu.edu.cn

Abstract

This paper presents the Graph Engine Service (GES), an advanced graph database management system characterized by its composable architecture and highly factorized query executor. This design enables GES to manage a substantial volume of concurrent queries with remarkable efficiency. GES currently holds the top position in the authoritative `LDBC-SNB-DECLARATIVE` benchmark rankings, showcasing its superior performance – exceeding the throughput of the second-place system by over three orders of magnitude, which provides its customers with exceptional cost-effectiveness and extraordinary user experiences.

CCS Concepts

• **Information systems** → **Database management system engines**; **Database query processing**.

Keywords

Graph Database, Factorization, Composable Architecture

*Both authors contributed equally to this research.
†Corresponding authors.

SIGMOD-Companion '25, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3724439

ACM Reference Format:
Sen Gao, Jianwen Zhao, Hao Zhang, Shixuan Sun, Chen Liang, Gongye Chen, Wenliang Zhang, Bo Ren, Chao Liu, Chenyi Zhang, Quan Chen, Chao Li, Jingwen Leng, and Minyi Guo. 2025. GES: High-Performance Graph Processing Engine and Service in Huawei (Technical Report). In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3722212.3724439

1 Introduction

Graph data is increasingly ubiquitous in Huawei's products, providing an intuitive and powerful means of modeling complex relationships and interconnections across diverse entities. A multitude of graph database management systems (GDBMS) [1, 12, 14, 17, 21, 27, 39, 41] have emerged as essential tools for processing such data. Their ability to efficiently traverse relationships and perform deep analytics on interconnected information makes them vital for a wide range of applications, including social networks, recommendation engines, fraud detection, and supply chain management [5, 40, 45, 46]. These application scenarios require handling diverse graph query workloads, such as transactional, analytical, and business intelligence tasks, which place high demands on system performance. Key requirements include high throughput, robust concurrency, data consistency, and low latency to ensure optimal performance and reliability.

Existing graph databases, such as Neo4j [27], AgensGraph [1], TigerGraph [39], TuGraph [41], and GraphDB [14], use relational query engines that handle and map graph-specific query language to relational operators, such as vertex expansion to join and property access to projection [17]. Their executors process graph data in a relational manner, with operators digesting inputs and generating results as sets of tuples, also referred to as a flat representation. As

demonstrated in our experiments, such relational tuples can consume several gigabytes of RAM for a single query. The extremely high memory overhead makes it hard for intermediate results to reside in the Last Level Cache (LLC). Additionally, the data movement when piping these tuples between operators incurs inefficiencies.

Factorization [28] is an effective method to address the volume issue of intermediate tuples and is widely used in RDBMS. It involves decomposing complex data structures into simpler, more normalized forms, representing them in a factorized format to reduce redundancy and ensure data integrity. Few graph database products apply factorization in their query engines, with Kùzu [17] being the only one to explore its usage, primarily focusing on optimizing the performance of many-to-many (m-n) joins by modifying the hash join operator.

However, existing graph databases that utilize factorization have several limitations. First, complex query workloads in graph databases are typically evaluated by cooperating different operators, not just joins. Current systems handle other operators on a case-by-case basis, which introduces additional overhead [15, 17, 18]. Second, during the evaluation process, flat and factorized data are frequently mixed and transformed, leading to significant overhead in processing time and computational resources, making them less suitable for large-scale or real-time applications. Third, in an execution plan, operators can be piped in various orders. Optimizing factorization manually for each operator order and combination incurs significant complexity in programming, engineering, and testing.

To address these challenges, we have developed an efficient and generic factorized query engine in Huawei Graph Engine Service (GES). The key technical approach is to design a practical factorized representation based on factorization theory [31] by developing efficient in-memory tree-like structures that reduce redundancy in intermediate results. The core components of our data structures are designed to be compact and succinct, maximizing cache efficiency even for billion-level graphs in more complex queries. Based on this new representation, the query engine can natively support the execution of most frequently used operators in a factorized manner without introducing significant additional overhead. The new GES engine is also designed to be compatible with existing query optimization techniques, such as operator fusion, which is effective in shortening execution plan pipelines and further reducing the cost of generating and passing intermediate results.

With the factorized query engine, GES achieves substantial performance improvements, enabling fast query execution, extremely high throughput, and low latency for complex query processing tasks. Extensive experiments have been conducted to demonstrate the effectiveness and efficiency of factorized GES, solidifying its leading position among popular graph database products. Additionally, the significance of the performance gains has been validated by the authoritative `LDBC-SNB-DECLARATIVE` benchmark [1], where GES outperformed competitors by over three orders of magnitude.

2 Background

In this section, we begin by describing GES's composable architecture. Next, we examine the various graph query workloads that GES is designed to handle.

[1] https://ldbcouncil.org/benchmarks/snb-interactive

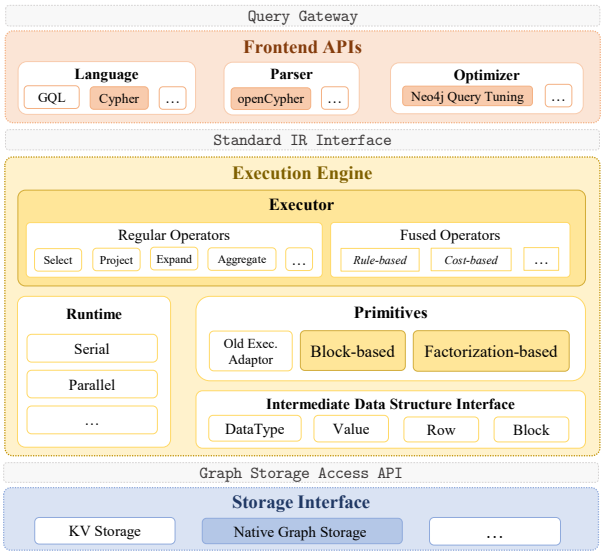


Figure 1: Composable GES architecture

2.1 System Overview

Architecture. The architecture of GES, as shown in Figure 1, consists of three disaggregated layers: a frontend layer, an execution engine layer, and a graph storage layer. This architecture follows a composable design [32]. Each layer of GES accommodates multiple components, and each component comprises multiple modules. Modules within the same component share some commonality but are functionally diverse. To accommodate the increasing diversity of workloads, GES can be configured as a specific graph data management system by selecting modules from different layers and registering them during development to achieve reusability and flexibility. Currently, GES has been extensively developed in Huawei Cloud to facilitate the querying and analysis of graph-structured data based on various relationships.

Graph Model. GES adopts the Label Property Graph (LPG) model [34]. LPG is a graph data structure that extends the basic graph model by allowing nodes (vertices) and edges (relationships) to be labeled and annotated with properties. In this model, nodes and edges can each have one or more labels, which provide a way to categorize them, and properties, which are key-value pairs that store additional data.

Execution Engine. Upon receiving the intermediate representation (IR) of a query from the frontend layer, the GES execution engine generates a corresponding physical execution plan based on the configured modules. The execution engine consists of four major components. The *Primitives* specify the data representation during query execution and are currently configured to prefer factorization through APIs in the *Intermediate Data Structure Interface*. In cases where factorization does not provide a performance benefit due to complex processing logic, the executor seamlessly switches to block-based execution, continuing the processing until completion. The *Runtime* manages the query workload parallelism, including sequential, inter-query parallel, and intra-query parallel execution, and also specifies the amount of computational resources invested.

The *Executor* handles individual operator evaluation based on the determined primitives and data structures. Through a unified storage access interface, the *Executor* runs operators against graph storage backends, which have diverse storage formats and remain transparent to the execution engine.

Our focus in this work is on the query executor. The other modules utilized in this system are highlighted in Figure 1.

2.2 Graph Query Workloads

General Graph Workload. The graph community classifies query workloads into three primary categories [21]: Online Analytical Processing (OLAP), Online Serving Processing (OLSP), and Online Transaction Processing (OLTP). OLAP workloads are typically complex, involving large-scale graph traversal for risk management and pattern detection, often accessing a significant portion of data with a high amplification rate and requiring parallelism and scalability to reduce latency and errors. OLSP workloads focus on real-time data fetching and processing, particularly for applications like recommendation systems, involving frequent updates from user actions. These workloads handle massive read and write query throughput and require high elasticity to manage peak load bursts and dynamic property changes. OLTP workloads, on the other hand, are transactional in nature, ensuring data consistency and atomicity, especially for updates across multiple vertices and edges, while maintaining high throughput with lightweight read and write operations.

LDBC SNB Interactive Workload. The LDBC Social Network Benchmark (SNB) Interactive workload simulates a real-world graph processing scenario involving complex read queries that access the neighborhood of a given node, alongside update operations that continuously insert new data into the graph. Specifically, the workloads in LDBC SNB interactive v1 consist of 29 different queries, which can be categorized into three types: 14 interactive complex read queries (`IC`), 7 interactive short read queries (`IS`), and 8 update queries (`IU`). `IC` queries involve complex interactions with the graph, such as exploring relationships between entities, aggregating data, or finding specific patterns in the network. `IS` queries focus on short and quick reads that retrieve specific pieces of information, such as user profiles, friends, or posts. `IU` queries simulate updates to the social network graph, such as adding new users, relationships, or posting content. Different queries are mixed based on their frequencies, which are determined by the workloads. Detailed descriptions of the workloads can be found in [20].

LDBC workloads are designed to comprehensively cover a wide range of graph processing tasks, including OLAP, OLTP, and OLSP. Additionally, our practical use of these workloads shows that they align closely with GES application scenarios that require querying rich relationship data, such as social relationship analysis, recommendations, information communication, and anti-fraud. Therefore, we test our system using these workloads to identify opportunities for optimization and conduct fair comparisons with other systems.

Typically, a complete run of the benchmark is conducted on two machines, where the system under test (SUT) is developed on one instance, and the driver implementing LDBC workloads runs on the other. The driver initializes queries and fires them to the SUT for evaluation. Upon receiving the query results from the SUT, the driver logs the results based on multiple metrics. Once the benchmark is complete, the driver audits the correctness and latency of the queries to ensure the benchmark is valid. It then computes a score based on the throughput of the SUT and generates a full report. An important parameter in the benchmark is the scale factor (SF), which refers to the size of the graph data in gigabytes and determines the amount of data used to run the benchmark.

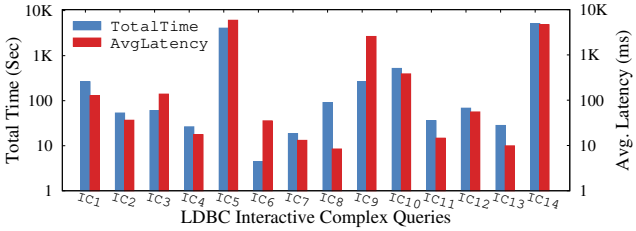
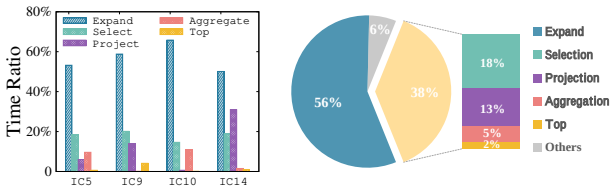


Figure 2: Benchmark results under LDBC SNB Interactive workload on the SF100 Graph



(a) Top operators by runtime (b) Runtime distribution across operators

Figure 3: Operator-level analysis of long-running queries

3 LDBC Workload Execution Analysis

Among the three types of LDBC workloads, we focus on analyzing and optimizing the performance of the GES execution engine on `IC` queries, as they typically incur higher computational and memory costs during evaluation.

3.1 Execution Analysis

The benchmark results of a single run of GES under LDBC SNB workloads on SF100 are depicted in Figure 2. Here, we focus solely on the query execution stage on a single core and omit the effects of network latency, query planning, and parsing. We also exclude `IS` and `IU` queries, as the costs associated with them are negligible. As shown in the figure, the total and average running times of different queries vary significantly. Several queries take hundreds of times longer to execute than others. In the entire benchmark test, these long-running queries contribute to prolonged resource monopolization, complicating task scheduling. As a result, system performance deteriorates as numerous short queries are blocked and delayed.

Operator-Level Analysis. Note that the GES execution engine evaluates a query based on its physical plan, which is provided by the optimizer. The evaluation process passes through multiple operators, with each operator consuming inputs from its child operator and sending the generated output to its parent operator. To analyze the evaluation cost of these long-running queries in greater detail, we break down the runtime by operators and present the results in Figure 3. Figure 3(b) highlights the most costly operators for each query. It is evident that the Expand operator dominates the runtime, accounting for nearly half of the total execution time.

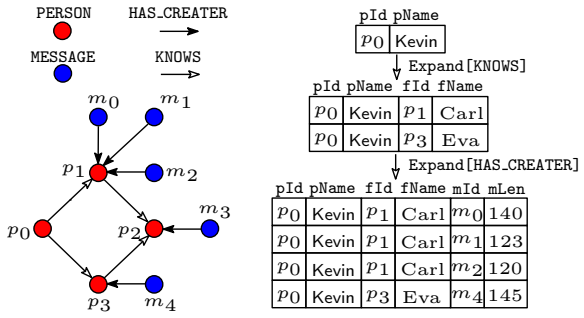


Figure 4: A data graph and its two-hop expansion in a flat representation for a given vertex p_0

Expand typically processes a set of vertices, accesses the edges of a specific relation for each vertex, and generates outputs consisting of all neighboring vertices. Other operators, such as Select and Project, also contribute a significant portion of the runtime.

To understand why the Expand operator is so costly, we conducted profiling on its implementation. The results reveal that a significant amount of time is spent on a loop that materializes a flat table of tuples, where each row consists of a combination of the source vertex and one of its neighbors. This inefficiency arises because, in most graph database engines, each operator processes data in a flat format, where the intermediate results (both inputs and outputs) are fully materialized as multiple tuples. An example illustrating this process is shown in Figure 4. In this example, for a given source vertex p_0 and its property, the tuple is replicated many times during the two-hop expansion, resulting in a large intermediate result. Consequently, the vast amount of data is passed through and processed by operators, leading to significant increases in memory consumption. Our experimental results further reveal that the peak size of intermediate results can reach several gigabytes, making them difficult to fit in the last-level cache (LLC).

Relational database management systems (RDBMS) have adopted factorization techniques to optimize the size of intermediate data. In graph databases, as far as we know, only Kùzu [15, 17] has made attempts to reduce this size by representing intermediate results in a factorized format, supported by theoretical foundations [31]. In these works, significant effort has been put into optimizing the performance of the join operator (which functions similarly to the Expand operator), while considering the execution of other operators on a case-by-case basis and demonstrating that additional overhead may arise during their processing. As shown in Figure 3(b), other operators also consume a large portion of the total runtime and should be handled efficiently. Moreover, the flat and factorized representations in existing systems are currently mixed and interconverted frequently throughout the evaluation, incurring additional computational overhead and reducing the overall performance gains achieved through factorized representations.

3.2 Executor Design Goals

To address these issues, we set out to design a highly factorized query executor capable of handling complex, real-world graph workloads efficiently. We summarize the design goals as follows:

- **Generalizability.** To optimize the size of intermediate results, the factorized representation should be adopted in GES's execution engine. Additionally, GES's executor should natively support the execution of most operators in a factorized manner.
- **Cache-efficiency.** The factorized representation designed in our engine should be concise and compact to significantly reduce the size of intermediate results. The construction of the representation and the processing of its data should be performed within the LLC to fully utilize data locality.
- **Integrability.** Many optimizations have been proposed for graph database query processors. GES's query executor should aim to accommodate these techniques and integrate them seamlessly and effectively.
- **Robustness.** For complex operators, such as blocking operators that require global information across all tuples, the executor should efficiently *de-factor* the representation into the standard flat format when necessary.

4 Query Executor

We next present GES's query executor – a highly factorized processor which executes the most (if not all) parts of a physical plan in a factorized manner (i.e., organizing the intermediate results produced by each operator in a compact, factorized data structure while preserving the factorization relationship). When encountering complex processing logic where factorization offers no benefit, the processor seamlessly reverts to block-based execution and continues processing until completion. Several optimization techniques have been developed and applied to address efficiency issues. We begin with a brief introduction to factorization theory, then illustrate the data structures adopted in our design, and subsequently discuss the end-to-end query execution process.

4.1 Background: Factorization

Conceptually, factorization is a way of representing a relation symbolically as relational algebra expressions consisting of *Union* (∪), *Cartesian Product* (×), and *Singletons*[2] [28]. It employs the *Distributivity of Cartesian product over Union* and factors out useless data values (and hence the name *"Factorization"*) to achieve succinctness.

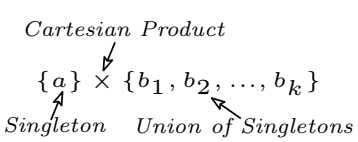
Factorized representations utilize a form known as the *f-tree* [28, 31], which exhibits several advantageous properties:

- **Redundancy elimination**. Data redundancies inherent in traditional tabular representations can be eliminated through a lossless yet more succinct factorized representation. Figure 5 shows an example where a relation represented in tabular form necessitates the repeated storage of attribute a by k times. In contrast, the factorized representation eradicates data redundancy, as each attribute value is stored only once.
- **Direct computation**. Some processing logic can be directly performed over the factorized representation without *"de-factoring"*. For instance, suppose the *f-tree* shown in Figure 5(c) is adopted to *"encode"* the relation in Figure 5(a), and we want to filter out some tuples based on the value of attribute b. Without *"decoding"*, we can directly scan the leaves (which store b) of the *f-tree* and mark the nodes that passes the filtering condition as valid. The subsequent computation can continue with the same *f-tree*.

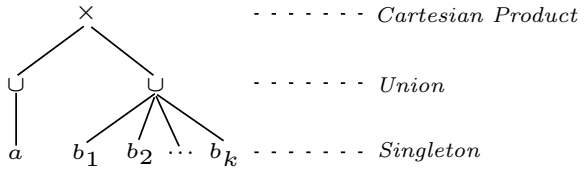
[2] A singleton is a unary relation with only one tuple.

Attri. a	Attri. b
a	b_1
a	b_2
...	...
a	b_k

(a) tabular representation



(b) factorized symbolic representation



(c) *f-tree* for *f-representation* [28]

Figure 5: Factorization eliminates data redundancies by its compact representation

- **Constant-delay enumeration** (Theorem 4.11. of [31]). The enumeration of tuples in a factorized representation incurs a time delay proportional to the size of each tuple, while remaining independent of the overall number of tuples.

In other words, the factorization technique significantly enhances query processing. Firstly, factorized intermediate results can be exponentially smaller than their flat representations [31], thereby reducing memory overhead and minimizing data movement between operators. Secondly, computations can be performed directly on the factorized results, resulting in fewer predicate and expression evaluations. Lastly, all tuples can be reproduced from the factorized representation with the same time complexity as generating the entire table.

4.2 Data Structures

Guided by factorization theory, we devise efficient in-memory data structures to accommodate intermediate results and facilitate graph query computations. The core component of our data structures is a cache-friendly, column-oriented structure termed "Factorized Block" (*f-Block*, denoted by $\mathcal{F}_\mathcal{B}$), which stores the *Union* of tuples over its own schema. A relation can be decomposed into the *Cartesian Product* of several *f-Blocks*. To maintain this *"product"* relationship, we construct a tree (*f-Tree*, denoted by $\mathcal{F}_\mathcal{T}$), where each node $v \in \mathcal{F}_\mathcal{T}$ manages an *f-Block*, and each edge $(u, v) \in \mathcal{F}_\mathcal{T}$ indicates the *Cartesian Product* relationship between the *f-Blocks* contained in u and v respectively. The details of the aforementioned data structures are illustrated below.

f-Block ($\mathcal{F}_\mathcal{B}$). An $\mathcal{F}_\mathcal{B}$ is defined as a set of columns. It also posses a schema, denoted as $\mathcal{S}(\mathcal{F}_\mathcal{B})$, which is the set of attributes depicted by each column. The following explains some basic properties:

- **Column-oriented storage**: each column stores a set of *Singletons* (with the same data type) in a consecutive chunk of memory.
- **Cardinality restriction**: each column has the same cardinality, denoted as $N_{\mathcal{F}_\mathcal{B}}$.
- **Relation representation**: given an index $i \in \left[1, N_{\mathcal{F}_\mathcal{B}}\right]$, we can generate a tuple $t = \mathcal{F}_\mathcal{B}{}^{[i]}$ by collecting all the values stored in the i-th position of each column. Then for an index range $[i, j]$,

$$\mathcal{F}_\mathcal{B}{}^{[i,j]} = \mathcal{F}_\mathcal{B}{}^{[i]} \cup \mathcal{F}_\mathcal{B}{}^{[i+1]} \ldots \cup \mathcal{F}_\mathcal{B}{}^{[j]}$$

Specially, $\mathcal{F}_\mathcal{B}{}^{\left[1,N_{\mathcal{F}_\mathcal{B}}\right]}$ represents all the tuples (over the schema $\mathcal{S}(\mathcal{F}_\mathcal{B})$) encoded by $\mathcal{F}_\mathcal{B}$.

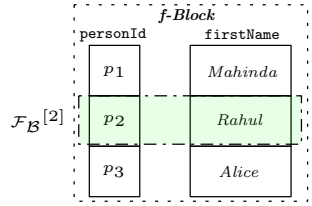


Figure 6: An example of *f-Block*

Example 4.1. Figure 6 illustrates an *f-Block* comprising two columns. $\mathcal{S}(\mathcal{F}_\mathcal{B})$ = {`personId, firstName`}, $N_{\mathcal{F}_\mathcal{B}} = 3$, and $\mathcal{F}_\mathcal{B}{}^{[2]}$ exemplifies a tuple $\{p_2, \textit{Rahul}\}$. □

f-Tree ($\mathcal{F}_\mathcal{T}$). The *f-tree* form, as introduced in [31], holds theoretical significance but lacks practical applicability due to efficiency concerns. To remedy this, we modify it to a tree that can be effectively deployed practically. An $\mathcal{F}_\mathcal{T}$ is a rooted tree where each node $u \in \mathcal{F}_\mathcal{T}$ contains:

- an *f-Block*, denoted as $\overset{u}{\mathcal{F}_\mathcal{B}}$.
- a selection vector S where

$$\text{for any } i \in \left[1, N_{\overset{u}{\mathcal{F}_\mathcal{B}}}\right], S[i] = \begin{cases} 1 & \text{if } \overset{u}{\mathcal{F}_\mathcal{B}}{}^{[i]} \text{ is valid} \\ 0 & \text{otherwise} \end{cases}$$

- a set of child pointers where each of them points to a specific child v of u. The directed edge (u, v) is associated with an index vector $I_{(u,v)}$. Each element $I_{(u,v)}[i] = [j, k]$ represents an index range, which indicates that $\overset{u}{\mathcal{F}_\mathcal{B}}{}^{[i]}$ adheres to the *Cartesian Product* rule with $\overset{v}{\mathcal{F}_\mathcal{B}}{}^{[j,k]}$. Phrased differently, $\overset{u}{\mathcal{F}_\mathcal{B}}{}^{[i]} \times \overset{v}{\mathcal{F}_\mathcal{B}}{}^{I_{(u,v)}[i]}$ defines a set of tuples over the schema $\mathcal{S}\left(\overset{u}{\mathcal{F}_\mathcal{B}}\right) \cup \mathcal{S}\left(\overset{v}{\mathcal{F}_\mathcal{B}}\right)$, where $i \in \left[1, N_{\overset{u}{\mathcal{F}_\mathcal{B}}}\right]$, $j \le k$ and $[j, k] \in \left[1, N_{\overset{v}{\mathcal{F}_\mathcal{B}}}\right]$.

Note that the last bullet applies only if u is an internal node of $\mathcal{F}_\mathcal{T}$.

Given a node $u \in \mathcal{F}_\mathcal{T}$ and let C_u be the children of u. Fix an $i \in \left[1, N_{\overset{u}{\mathcal{F}_\mathcal{B}}}\right]$, we use $\mathcal{R}_u^i$ to symbolize the relation *"induced"* by u and i in the sub-tree of $\mathcal{F}_\mathcal{T}$ rooted at u (denoted as $\mathcal{F}_{\mathcal{T}u}$), where

$$\mathcal{R}_u^i = \begin{cases} \overset{u}{\mathcal{F}_\mathcal{B}}{}^{[i]} & \text{if } u \text{ is a leaf node} \\ \overset{u}{\mathcal{F}_\mathcal{B}}{}^{[i]} \times \left(\bigtimes_{v \in C_u} \bigcup_{j \in I_{(u,v)}[i]} \mathcal{R}_v^j \right) & \text{otherwise} \end{cases} \quad (1)$$

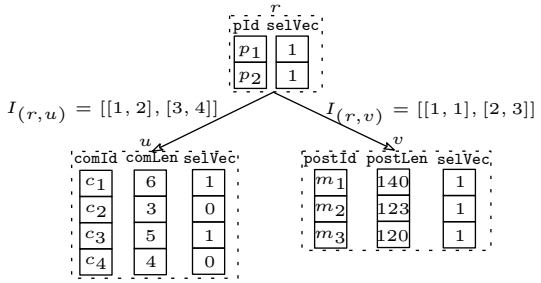


Figure 7: An example of *f-Tree*

Denote by r the root of a given $\mathcal{F}_\mathcal{T}$, then the relation $\mathcal{R}_{\mathcal{F}_\mathcal{T}}$ factorized by $\mathcal{F}_\mathcal{T}$ can be calculated as follows:

$$\mathcal{R}_{\mathcal{F}_\mathcal{T}} = \bigcup_{i \in \left[1, N^{\ r}_{\mathcal{F}_\mathcal{B}}\right]} \mathcal{R}^i_r \tag{2}$$

Example 4.2. Figure 7 demonstrates an $\mathcal{F}_\mathcal{T}$ rooted at r. By equation (1), $\mathcal{R}^1_u = \{c_1, 6\}, \mathcal{R}^3_u = \{c_3, 5\}, \mathcal{R}^1_v = \{m_1, 140\}, \mathcal{R}^2_v = \{m_2, 123\}$ and $\mathcal{R}^3_v = \{m_3, 120\}$ since u and v are both leaf nodes. $\mathcal{R}^2_u$ and $\mathcal{R}^4_u$ are invalid as indicated by the selection vector of u. Additionally,

$$\begin{aligned}\mathcal{R}^1_r &= \{p_1\} \times \mathcal{R}^1_u \times \mathcal{R}^1_v = \{\{p_1, c_1, 6, m_1, 140\}\} \\ \mathcal{R}^2_r &= \{p_2\} \times \mathcal{R}^3_u \times \left(\mathcal{R}^2_v \cup \mathcal{R}^3_v\right) \\ &= \{\{p_2, c_3, 5, m_2, 123\}, \{p_2, c_3, 5, m_3, 120\}\}\end{aligned}$$

Thus by equation (2), we can show that:

$$\begin{aligned}\mathcal{R}_{\mathcal{F}_\mathcal{T}} &= \mathcal{R}^1_r \cup \mathcal{R}^2_r \\ &= \{\{p_1, c_1, 6, m_1, 140\}, \{p_2, c_3, 5, m_2, 123\}, \{p_2, c_3, 5, m_3, 120\}\}\end{aligned}$$

namely, $\mathcal{R}_{\mathcal{F}_\mathcal{T}}$ encodes 3 valid tuples. □

Denote by $\mathcal{S}\left(\mathcal{R}_{\mathcal{F}_\mathcal{T}}\right)$ the schema of $\mathcal{R}_{\mathcal{F}_\mathcal{T}}$. It is important to note the following *disjoint schema partition* property:

- $\bigcup_{u \in \mathcal{F}_\mathcal{T}} \mathcal{S}\left(\mathcal{F}^u_\mathcal{B}\right) = \mathcal{S}\left(\mathcal{R}_{\mathcal{F}_\mathcal{T}}\right)$.
- $\mathcal{S}\left(\mathcal{F}^u_\mathcal{B}\right) \cap \mathcal{S}\left(\mathcal{F}^v_\mathcal{B}\right) = \emptyset, \forall u, v \in \mathcal{F}_\mathcal{T}$.

In other words, the proposed *f-Tree* decomposes the schema of a relation into disjoint subsets and utilizes its nodes to *cover* each subset. The internal structures (e.g., *f-Block*) and the metadata (e.g., the index vector) stored therein collectively manage the relational algebra relationships (i.e., *Union* and *Cartesian Product* of Singletons), thereby constituting the entire *factorization* logic.

Example 4.3. For the $\mathcal{F}_\mathcal{T}$ shown in Figure 7, $\mathcal{S}\left(\mathcal{F}^r_\mathcal{B}\right) = \{\texttt{pId}\}$, $\mathcal{S}\left(\mathcal{F}^u_\mathcal{B}\right) = \{\texttt{comId, comLen}\}$ and $\mathcal{S}\left(\mathcal{F}^v_\mathcal{B}\right) = \{\texttt{postId, postLen}\}$. Clearly, they are disjoint and $\mathcal{S}\left(\mathcal{R}(\mathcal{F}_\mathcal{T})\right) = \mathcal{S}\left(\mathcal{F}^r_\mathcal{B}\right) \cup \mathcal{S}\left(\mathcal{F}^u_\mathcal{B}\right) \cup \mathcal{S}\left(\mathcal{F}^v_\mathcal{B}\right) = \{\texttt{pId, comId, comLen, postId, postLen}\}$. □

The following lemma manifests an important property of an $\mathcal{F}_\mathcal{T}$.

Lemma 4.4. *The tuples factorized by a given $\mathcal{F}_\mathcal{T}$ can be enumerated in $O\left(\left|\mathcal{S}\left(\mathcal{R}_{\mathcal{F}_\mathcal{T}}\right)\right|\right)$ delay and space.*

The above lemma implies that the proposed *f-Tree* structure retains the *constant-delay enumeration* property with respect to data complexity (where the schema size will be regarded as a constant), which is crucial for efficiency in case we need to materialize all the tuples in $\mathcal{R}_{\mathcal{F}_\mathcal{T}}$. The proof is provided in Appendix A.

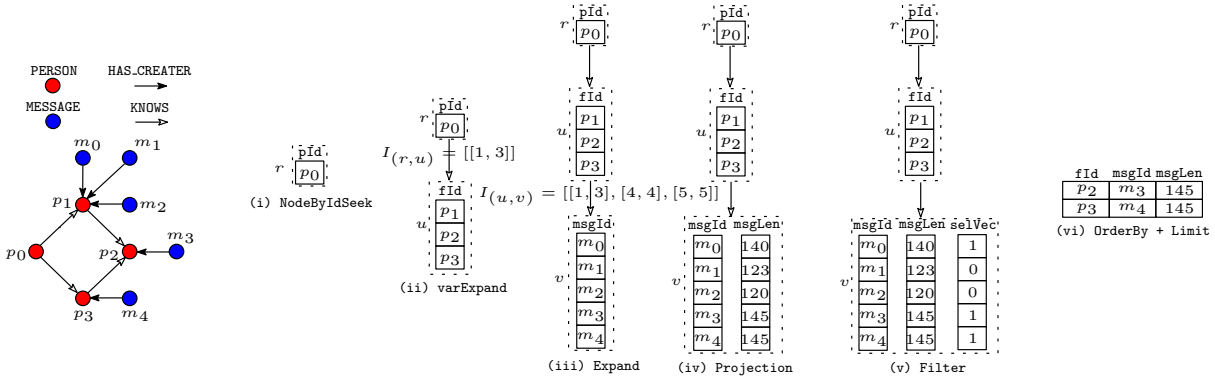
Flat-Block. The intricate computational logic, such as *group-by* operations involving multiple attributes that span different nodes of a *f-Tree* ($\mathcal{F}_\mathcal{T}$), diminishes the efficiency of factorization. Consequently, the ultimate solution involves *de-factoring* the $\mathcal{F}_\mathcal{T}$ and storing the enumerated tuples into a *flat* data structure – specifically, the flat-block. A flat-block is essentially a fundamental, row-oriented data structure widely adopted in a typical database executor where each row corresponds to a distinct tuple.

4.3 Query Processing

We are now ready to discuss the processing of a graph query by our highly factorized executor. Our discussion begins with an overview of the frequently used operators in a typical graph database. It then continues with a detailed presentation of the factorized execution process, using an end-to-end example. Finally, we examine the primary optimization technique, known as operator fusion, which effectively reduces the overhead introduced by the *de-factoring* process when the ultimate solution has to be adopted.

Operators. A typical interactive graph query starts with a given vertex s of the data graph G and performs a multi-hop traversal to obtains a set of neighbors within certain distance from s. Meanwhile, properties of neighbors are obtained for further processing (e.g., filtering, statistics calculation). The final results presented to a user are often sorted and limited to a certain cardinality. This process involves the following frequently used operators.

- **`Expand`**. Similar to the `Join` operator in relational databases, the `Expand` operator is the most data-intensive. It extends a set of starting vertices to their (multi-hop) neighbors, potentially generating an exponentially large number of intermediate results (i.e., a substantial volume of *"start-vertex, neighbor"* pairs). To eliminate duplicates, source vertices can be stored in an *f-Block*, while their neighbors are recorded in another *f-Block*. These two factorized blocks are interconnected through the nodes of the corresponding *f-Tree*, with neighbor relationships specified by the index vector associated with each *f-Tree* edge. Essentially, each execution of the `Expand` operation adds *f-Tree* nodes to a factorization tree ($\mathcal{F}_\mathcal{T}$), progressively expanding it into a larger structure.
- **`Projection, Filter`**. Due to the columnar storage structure of the *f-Block*, it is straightforward to append new columns (e.g., for storing vertex or edge properties) or retrieve specific columns from an $\mathcal{F}_\mathcal{B}$. Consequently, `Projection` operations can be efficiently managed. Regarding `Filter` operations, the *disjoint schema partition* property of an $\mathcal{F}_\mathcal{T}$ allows for the rapid identification of the *f-Tree* nodes encompassing the filtering attributes. Subsequently, the corresponding selection vector can be updated by evaluating the filtering expressions.



(a) a data graph G (b) growth of the *f-tree*

Figure 8: An example of factorized query processing

- **`Order-By, Aggregation`**. The simplest case of `Order-By` (or `Group-By`) involves attributes that are confined to the same *f-Tree* node. In this scenario, we traverse and identify the corresponding *f-Tree* node, subsequently appending a special column to the $\mathcal{F}_{\mathcal{B}}$ to indicate the orders (or groups) of each tuple. Enumeration thus follows by scrutinizing this special column. However, when `Order-By` (or `Group-By`) attributes span multiple *f-Tree* nodes, the process becomes significantly more complex, rendering factorized structures less advantageous. In such cases, the ultimate solution is to *de-factor* the $\mathcal{F}_{\mathcal{T}}$ into a flat-block and the subsequent downstream tasks (e.g., aggregation) will resort to the traditional block-based execution.

Factorized Execution. We present an end-to-end example to elucidate the factorized execution process. Consider the following `Cypher` query, which shares a query pattern similar to `IC9` as depicted in Figure 10(b), executed over the data graph in Figure 8(a):

```
MATCH (p:PERSON)-[:KNOWS*1..2]->(f) WHERE id(p) = p0
WITH f
MATCH (f:PERSON)<-[:HAS_CREATOR]-(msg)
WHERE msg.len > 125
RETURN id(f), id(msg), msg.len
ORDER BY msg.len DESC, id(f) ASC
LIMIT 2
```

The query will be passed to the GES's parser and optimizer, ultimately generating a physical plan which will be handed over to the executor. One possible plan and its execution process for the aforementioned query is demonstrated in Figure 8(b) and illustrated as follows: a `NodeByIdSeek` operator first locates the vertex with the identifier p_0 in G and stores it in the root r of an $\mathcal{F}_{\mathcal{T}}$ (Figure 8(b)(i)). Subsequently, by following `"KNOWS"` edges, a 2-hop expansion is performed to find all friends of p_0 within a distance of 2 (in terms of the number of edges). The friends' IDs are stored in another *f-Block* contained within an *f-Tree* node u, whose parent is r, with the *Cartesian Product* relationship between p_0 being specified by the index vector $I_{(r,u)}$ (Figure 8(b)(ii)). Similarly, another `Expand` operator identifies messages created by those friends, extends the $\mathcal{F}_{\mathcal{T}}$ by adding a child node v to u, and stores the messages' IDs in $\mathcal{F}_{\mathcal{B}}^{v}$.

The index vector $I(u,v)$ is attached to the *f-Tree* edge (u,v) for factorization purposes (Figure 8(b)(iii)). Subsequently, a `Projection` operator appends a new column to $\mathcal{F}_{\mathcal{B}}^{v}$ to accommodate the message length (Figure 8(b)(iv)), and a `Filter` operator updates the selection vector of v by evaluating the filtering condition `"msg.len > 125"` (Figure 8(b)(v)). Observing that the attributes involved in the `Order-By` operation span two different nodes of $\mathcal{F}_{\mathcal{T}}$, our query executor thus automatically *de-factors* $\mathcal{F}_{\mathcal{T}}$ by enumerating the valid tuples with a constant delay while maintaining and outputting the final top-2 results in the form of *flat-block* (Figure 8(b)(vi)).

Applicability and Trade-offs. As mentioned before, complicated computations diminish the efficiency of factorized data structures. For instance, handling cycles or loops in a query requires performing joins between different *f-Blocks*. In such cases, all tuples have to be materialized, causing GES's executor to revert to the traditional flat-block-based execution. Although Lemma 4.4 theoretically guarantees a fast *de-factoring* process, practical implementations still incur non-negligible overhead due to *f-Tree* traversals, memory copies and data movements.

Operator Fusion. To maximally reduce overheads and exploit performance advantages of factorization, we use *operator fusion* – an optimization technique used in database systems to improve query performance by merging multiple operators into a single, more efficient operator [24]. This process minimizes the overhead associated with materializing intermediate results and passing tuples between operators. By fusing operators together, the system can execute complex operations more efficiently, reducing the number of intermediate steps and memory accesses, leading to better CPU and cache utilization.

It has been observed that certain computation patterns frequently occur in interactive graph queries. For instance, `Aggregation` followed by `Projection` and `Top-k` is commonly seen not only in LDBC benchmarks (e.g., `IC5` and `IC6`) but also in queries from GES's customers. However, complex aggregations are operations that our factorized executor is not well-suited for, and therefore must be processed by first *de-factoring* the *f-tree*. This process may introduce overhead and counteract the benefits of factorization. This is precisely the scenario where operator fusion can have a

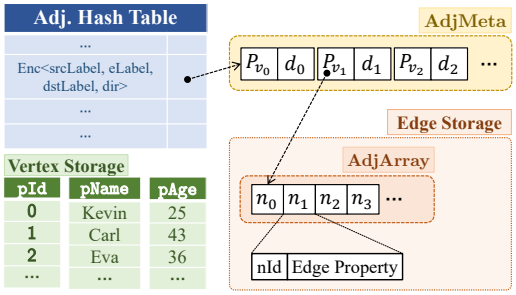


Figure 9: The memory layout of GES storage

significant impact. GES's optimizer effectively identifies the aforementioned pattern based on predefined heuristic rules and fuses them into a single operator. In addition to the *AggregateProjectTop* fusion, other commonly used fusion rules can also be efficiently implemented using our *f-Tree*. For example, the *VertexExpand* fusion combines the `NodeSeek` and `Expand` operators to directly retrieve the set of neighbors of a given vertex without enumerating its adjacent edges. The *FilterPushDown* fusion pushes the predicates of the `Filter` operator down to the appropriate pattern matching, reducing intermediate results. For instance in Figure 8(b), the `Filter` in step (v) can be moved behind step (ii) and fused with the `Expand` operator in step (iii), avoiding the listing of unused neighbors m_2 and m_3 along with their properties. Many other operator fusion rules can also be applied to the factorized representation we propose, without introducing additional overhead.

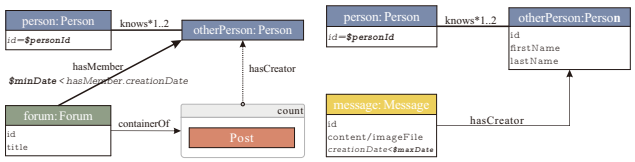
5 Implementation Details

Graph Storage. Our system stores the entire graph using an adjacency list representation implemented as an array *adjMeta* of arrays (*adjArray*) as shown in Figure 9, where *adjArray* stores neighbors of each vertex and *adjMeta* of size $|V|$ indexes the metadata of *adjArray*, including the RAM address and length. The *adjMeta* can be refered in a hash table where the key is encoded as a tuple $\langle srcLabel, edgeLabel, dstLabel, direction \rangle$. Since the size of the hash table is quite small in the real-world applications and *adjArray* can be directly accessed by *adjMeta*, therefore, the cost to identify an adjacency array is minor. It is known that adjacency array is not that flexible for the update of graph topology, here we adopt a widely-used technique, marking for deletion and allocating larger space once insertions take all slots in adjacency arrays, to address this issue. For vertex properties, we organize them in a columnar table, with each row corresponding to a vertex and each column representing a property. Our graph storage shows practically efficiency based on the truth that read queries are dominant in the real-world applications.

Concurrency Control. To coordinate query execution and versioning, the system employs a version manager initialized to zero. Because write queries update the graph with known write sets in advance, we coordinate them using the classical Multi-Version Two-Phase Locking (MV2PL) protocol to ensure serializability. MV2PL allows for non-blocking reads. To support this concurrency control method, we maintain coarse-grained versions at the vertex level rather than at the edge level. Specifically, a write query creates a new snapshot for the vertices it modifies using a copy-on-write strategy, while read queries construct a graph snapshot by combining the snapshots of these vertices. Additionally, our system employs a memory pool to facilitate the copy-on-write strategy, reducing the overhead caused by frequent memory allocation and deallocation by the operating system.

Table 1: Datasets and statistics

Scale Factors	#Persons	#Vertices	#Edges	Graph Size
SF1	11K	4.0M	23.0M	1 GiB
SF10	73K	36.4M	231.4M	10 GiB
SF30	182K	106.7M	701.4M	30 GiB
SF100	499K	337.4M	2.3B	100 GiB
SF300	1.25M	969.9M	6.7B	300 GiB



(a) IC5 (b) IC9

Figure 10: Visualization of sample interactive complex read queries. Keywords in bold italics represent input parameters.

Vectorization. Our factorized executor is well-suited for incorporating *vectorization* as a general optimization technique, primarily due to the column-oriented design of *f-Block*. During the generation of a physical plan, the *f-Tree* (and consequently, the *f-Blocks*) is pre-allocated in advance. This pre-allocated *f-Tree* is subsequently reused across different processing batches, thereby avoiding the overhead associated with dynamic memory allocation and deallocation. Additionally, we leverage SIMD instructions provided by modern CPU architectures to further accelerate query processing.

Pointer-based Join. In essence, pointer-based join is a technique used to efficiently process join queries by leveraging pointers or references to link related records [37], bypassing the need for full table scans or nested loops, which can be computationally expensive. This can lead to significant performance gains, especially for large datasets and complex queries. For graph queries, finding neighbors for a given set of vertices is an ubiquitous operation, commonly appearing in an `Expand` operation (i.e., the *Join*). This typically necessitates copying neighbor IDs into the data structure accommodating intermediate results (e.g., a column of an *f-Block*), which incurs additional overhead. Fortunately, for a given vertex, we can directly access its neighbors by scanning the corresponding *adjArray*, thanks to our graph storage design (see Figure 9). Therefore, in our practical implementation, instead of materializing neighbor IDs into a column, we can store only (1) the pointer to the memory address of *adjArray* and (2) the size of *adjArray* to avoid expensive data scan and copy. Those neighbor IDs are accessed or lazily copied via the stored pointer and size information only if we have to do so (e.g., during *de-factoring* of an *f-Tree*). This approach dramatically accelerates the join processing.

6 Evaluation

Experiment Setup. The experiments are conducted on two ac8.48-xlarge.4 instances in Huawei Cloud, where one runs the system

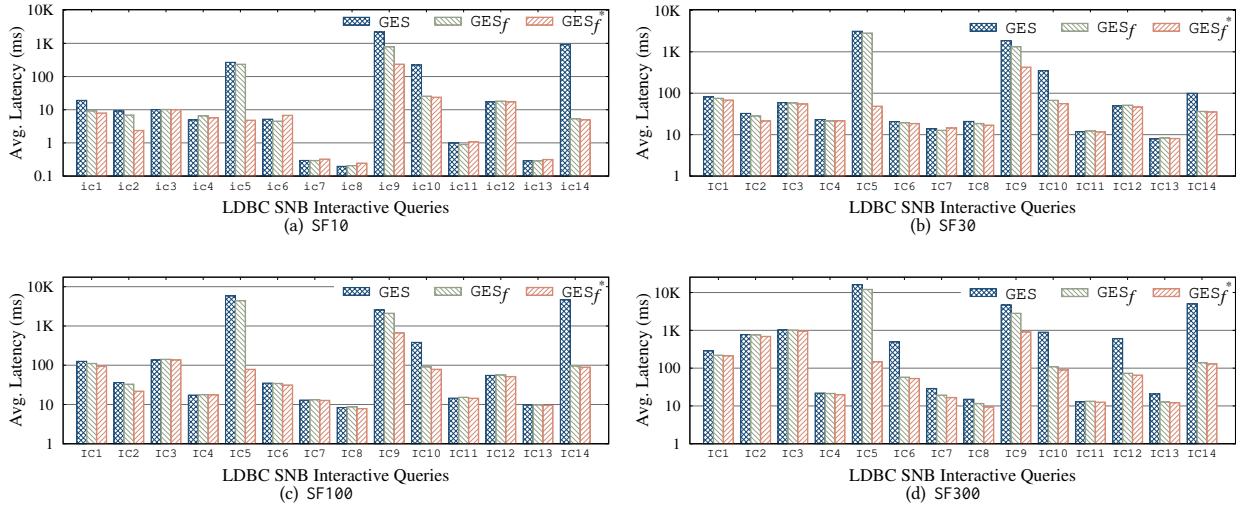


Figure 11: Average query latency with flat, factorized, and operator-fusion-based factorized query engines on four graphs

under test (SUT) and the other runs the driver, which fires queries against the SUT based on the workloads, collects and logs the returned results, and generates statistical reports. Each instance, running Ubuntu 20.04 LTS, is equipped with 96 AMD vCPUs at 2.4 GHz, 384 MB of LLC cache, and 768 GB of DDR5 RAM.

Graph Datasets and Workloads. All experiments are conducted following the LDBC SNB benchmark auditing policies, as the benchmark covers a comprehensive set of graph processing scenarios. The datasets used in the experiments are generated using the LDBC SNB Hadoop-based Datagen [19]. Table 1 presents the statistics for five graphs of varying scales, with the graph size calculated based on the storage required for both the topologies and the properties of the vertices and edges. The workloads, including the query generation method, the number of each type of query, and the frequency of query submissions, align with the benchmark. We ensure that all results are reported under the best Time Compression Ratio (TCR), which means that the number of delayed queries does not exceed 5% of the total. Two example queries and their corresponding tasks are illustrated in Figure 10. More details about the datasets, workloads, and auditing can be found in [20].

System Under Test (SUT). We compare our system with multiple leading competitors. For GES, we consider three variants: GES, GES$_f$, and GES$_f^*$. The versions GES and GES$_f$ represent intermediate results in flat and factorized forms, respectively. The version of GES with the best performance, denoted by GES$_f^*$, is obtained by applying operator fusion to GES$_f$. To evaluate competitors using the LDBC SNB interactive benchmark, each system must implement the LDBC driver interface. Although our goal was to include a diverse array of graph database systems in this comparison, several systems, such as Kùzu, lack the required implementation and are therefore excluded from this experiment. Nevertheless, the comparison includes six widely adopted systems: Neo4j v5.25.1 [27], PostgreSQL v17.1.0 [33], GraphDB v10.7.6 [14], TigerGraph v2.5.1 [39], TuGraph v1.4.4 [41], and AgensGraph v2.1.2 [1].

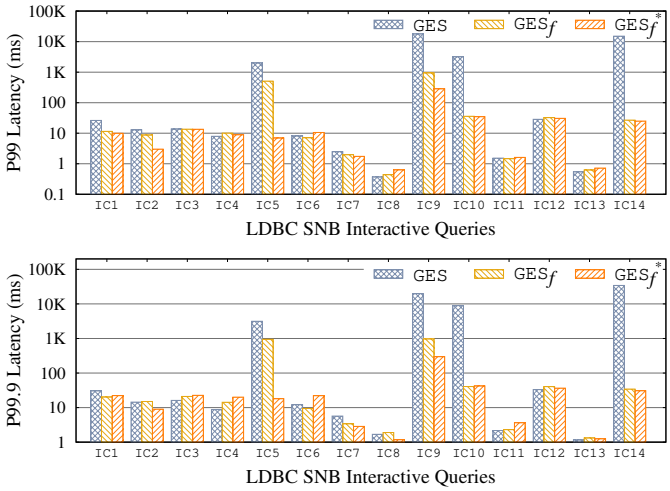


Figure 12: P99 and P99.9 tail latency on SF300

6.1 Ablation Study

To study the effects of factorization and optimizations, we first compare the performance of three variants of our executor: GES, GES$_f$, and GES$_f^*$.

Latency of Individual Queries. Figure 11 presents the average response time of each query on four graphs. As we can clearly see, GES$_f$, equipped with the factorized executor, outperforms the baseline GES for all queries across all datasets. The speedups for queries like IC10 and IC14 reach one to two orders of magnitude, even on the large-scale graph SF300. Moreover, the operator-fusion-based version GES$_f^*$ further reduces the runtime by several orders of magnitude for queries that are not efficiently handled by GES$_f$. For instance, on SF10, the optimization achieved through factorization alone may be less pronounced. This could be attributed to the simplicity of query patterns in short-running queries, which involve small, already compact flat blocks, resulting in limited benefits from

Table 2: The peak RAM usage of intermediate results and the reduction ratio (R.R.) of $\texttt{GES}^*_f$ compared to GES, with R.R. above 90% highlighted

Query	SF10				SF30				SF100				SF300			
	GES	$\texttt{GES}_f$	$\texttt{GES}^*_f$	R.R.	GES	$\texttt{GES}_f$	$\texttt{GES}^*_f$	R.R.	GES	$\texttt{GES}_f$	$\texttt{GES}^*_f$	R.R.	GES	$\texttt{GES}_f$	$\texttt{GES}^*_f$	R.R.
IC1	3.0 MB	1.1 MB	91.9 KB	**96.9%**	7.0 MB	2.7 MB	211.0 KB	**97.0%**	16.0 MB	5.9 MB	361.2 KB	**97.7%**	31.9 MB	12.1 MB	940.5 KB	**97.1%**
IC2	8.5 MB	936.3 KB	460.5 KB	**94.6%**	9.3 MB	1.2 MB	448.6 KB	**95.2%**	12.5 MB	1.7 MB	569.7 KB	**95.4%**	12.5 MB	2.1 MB	476.6 KB	**96.2%**
IC3	6.7 MB	6.7 MB	6.6 MB	0.7%	22.1 MB	22.1 MB	22.0 MB	0.3%	61.8 MB	61.7 MB	61.7 MB	0.1%	186.0 MB	186.0 MB	185.9 MB	0.1%
IC4	3.3 MB	1.9 MB	1.8 MB	46.5%	6.6 MB	3.9 MB	3.6 MB	45.5%	6.6 MB	3.8 MB	3.6 MB	45.7%	9.5 MB	5.5 MB	5.2 MB	45.2%
IC5	58.0 MB	15.1 MB	1.7 KB	**99.9%**	530.4 MB	183.3 MB	1.9 KB	**99.9%**	879.3 MB	286.8 MB	1.6 KB	**99.9%**	1.4 GB	435.2 MB	1.6 KB	**99.9%**
IC6	2.1 MB	2.0 MB	1.4 MB	33.4%	4.2 MB	4.1 MB	2.7 MB	36.1%	9.4 MB	9.3 MB	5.4 MB	43.0%	21.1 MB	21.0 MB	10.4 MB	50.7%
IC7	9.3 KB	8.0 KB	7.7 KB	16.7%	28.8 KB	20.4 KB	19.9 KB	30.8%	78.4B	67.2B	67.2B	14.3%	24.2 KB	17.0 KB	16.6 KB	31.3%
IC8	9.3 KB	8.3 KB	7.0 KB	25.3%	4.2 MB	735.3 KB	600.9 KB	85.6%	10.5 KB	7.7 KB	6.6 KB	37.6%	17.2 KB	9.4 KB	8.4 KB	50.9%
IC9	1.0 GB	106.3 MB	57.8 MB	**94.5%**	1.6 GB	170.9 MB	86.1 MB	**94.6%**	2.4 GB	254.9 MB	127.2 MB	**94.6%**	3.0 GB	346.9 MB	151.0 MB	**94.9%**
IC10	36.8 MB	36.8 MB	36.6 MB	0.6%	56.0 MB	56.0 MB	55.8 MB	0.4%	76.1 MB	76.2 MB	75.9 MB	0.2%	93.1 MB	93.3 MB	92.9 MB	0.2%
IC11	430.1 KB	418.6 KB	406.7 KB	5.4%	611.8 KB	600.3 KB	588.5 KB	3.8%	796.2 KB	784.7 KB	772.8 KB	2.9%	1.1 MB	1.1 MB	1.1 MB	2.0%
IC12	25.4 MB	25.4 MB	14.5 MB	43.1%	33.8 MB	33.8 MB	19.2 MB	43.2%	48.6 MB	48.6 MB	27.6 MB	43.2%	58.6 MB	58.6 MB	33.2 MB	43.3%
IC13	192.0B	192.0B	192.0B	0.0%	192.0B	192.0B	192.0B	0.0%	192.0B	192.0B	192.0B	0.0%	192.0B	192.0B	192.0B	0.0%
IC14	12.7 MB	110.1 KB	107.0 KB	**99.2%**	11.7 MB	10.3 KB	9.9 KB	**99.9%**	16.6 MB	955.4 KB	928.4 KB	**94.4%**	28.9 MB	798.6 KB	776.1 KB	**97.3%**

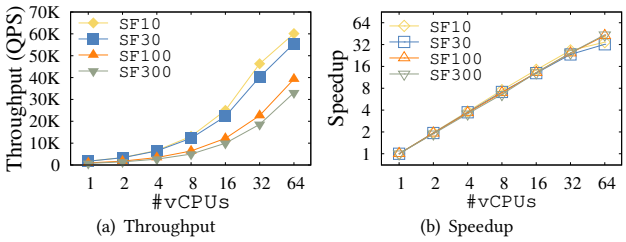
Table 3: Comparison of throughput and speedup across three GES variants

SFs	GES	$\texttt{GES}_f$		$\texttt{GES}^*_f$	
SF10	3,866	18,527	***4.8x***	65,718	***17.1x***
SF30	3,492	15,690	***4.5x***	59,376	***17.0x***
SF100	3,160	13,017	***4.1x***	51,986	***16.4x***
SF300	2,849	11,527	***4.1x***	45,513	***16.1x***

factorization. However, optimizations like operator fusion can be seamlessly integrated into our well-designed *f-tree*, allowing these workloads to be processed very efficiently as well. Additionally, the improvement on long-running workloads (e.g., IC5, IC9, IC10, and IC14) is more pronounced. In these cases, our compact *f-tree* plays a more significant role, as the larger query sizes and more complex patterns benefit from reduced computational complexity. As a result, the *f-tree* leads to more substantial performance gains, particularly in terms of execution time and resource utilization, compared to shorter-running queries. Another notable observation is the consistent and impressive speedup of $\texttt{GES}^*_f$ over GES across various graph sizes, achieving speedups of 22.1x, 48.4x, 78.2x, and 109.7x on four datasets for IC5.

Tail latency is depicted in Figure 12. The experimental results demonstrate that $\texttt{GES}_f$ and $\texttt{GES}^*_f$ generally achieve much lower tail latencies compared to the baseline GES across most workloads, particularly at the 99th and 99.9th percentiles. Notably, for high-latency queries such as IC5, IC9, IC10, and IC14, both $\texttt{GES}_f$ and $\texttt{GES}^*_f$ dramatically reduce latency—IC5 drops from over 2,000 ms with GES to under 20 ms with $\texttt{GES}^*_f$, and IC9 decreases from approximately 17,930 ms to 286 ms. This trend highlights the effectiveness of $\texttt{GES}_f$ and $\texttt{GES}^*_f$ in mitigating extreme latency spikes. However, some variations exist; for instance, in IC4 and IC6, $\texttt{GES}_f$ and $\texttt{GES}^*_f$ do not consistently outperform GES, and the reason is similar to the explanation provided above.

Memory Footprint. Table 2 shows the memory usage of intermediate data for each query. Note that the RAM usage for traversal operators, such as ShortestPath in IC13, is not counted since these



(a) Throughput (b) Speedup

Figure 13: Scalability test with varying the number of threads configured for SUT

operators typically perform complex graph traversals by predefined subgraph patterns and are implemented as stored procedures, where intermediate data is hard to factorize. The experimental results show significant memory reductions in $\texttt{GES}^*_f$ compared to GES across various graph sizes, with reduction ratios (R.R.) frequently exceeding 90%, especially for more complex queries like IC1, IC2, IC5, IC9, and IC14, where memory savings are as high as 99.9%. Notably, for smaller graphs (SF10), queries IC1 and IC2 achieve a reduction ratio (R.R.) of over 96%, with IC1 reducing from 3.0 MB to 91.9 KB and IC2 from 8.5 MB to 460.5 KB. As graph size increases, the reduction remains impressive, particularly for larger graphs (SF300), with IC9 reducing from 3.0 GB to 151.0 MB. However, for queries like IC3 and IC10, the reduction is negligible. This is because these query patterns involve enumerating 4-cycles, and as discussed earlier, such cases require materializing all intermediate results, causing the executor to revert to flat execution. Overall, the results highlight the potential for significant performance gains in most scenarios using our optimized executor, especially when scaling to larger datasets.

Overall Throughput. We compare the overall performance in throughput of the three GES variants in Table 3 using the official LDBC SNB benchmark. The experimental results clearly indicate that both $\texttt{GES}_f$ and $\texttt{GES}^*_f$ substantially enhance overall throughput compared to GES across all graph scales evaluated. Specifically, $\texttt{GES}_f$ with the factorized query engine achieves an approximate 4-fold speedup, increasing throughput from 3,866.8 queries per

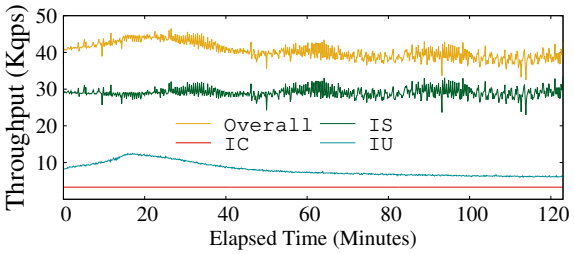


Figure 14: Throughput trace of GES^*_f across the full benchmark duration on SF300

second to 18,527.7 (SF10), and maintaining similar proportional improvements across larger scales. Remarkably, by fusing operators in the factorized engine, GES^*_f delivers an even more significant performance boost, attaining up to a 17-fold speedup by elevating throughput from 3,866.8 queries per second to 65,718.6 at SF10, and consistently around 16 to 17 times faster across larger scales. This consistent scalability in speedup underscores the robustness and efficiency of GES_f and GES^*_f in handling increasing workloads. The results demonstrate that adopting GES_f and especially GES^*_f can lead to substantial performance gains in throughput, making them highly effective choices for applications requiring high-performance graph processing.

6.2 Scalability and Stability

Scalability The results shown in Figure 13 highlight the impressive scalability and robust performance of GES^*_f across varying workloads and computational resources. As the number of virtual CPUs (vCPUs) increases from 1 to 64, throughput consistently rises on graphs with all scale factors. For instance, at SF10, throughput escalates from 1,740.57 queries per second with a single vCPU to 60,125.86 queries per second with 64 vCPUs, achieving a speedup of approximately 34.54x. Similar trends are observed across larger scale factors, where higher workloads benefit proportionally from increased vCPUs, with speedups reaching up to 43.68x at SF300. As observed, for smaller graphs such as SF10 and SF30, increasing the number of vCPUs results in only moderate throughput improvements—for example, scaling from 32 to 64 vCPUs. This limited gain is because the bottleneck shifts to other resources, such as networking and disk I/O bandwidth. Additionally, we notice that throughput continues to increase when bandwidth is enhanced, indicating that optimizing these resources can further improve performance for smaller graph workloads.

Stability We plot the throughput trace of GES^*_f on SF300 after a complete run of the LDBC SNB benchmark in Figure 14. This figure reports the IC, IS, IU, and overall throughput over the entire 2-hour period. Note that the query frequency of each type is determined and controlled by the benchmark driver to simulate real production workloads. The results demonstrate that the throughput for every type of query remains highly stable over time, despite minor short-term fluctuations. This indicates that GES^*_f maintains consistent performance under sustained workloads, ensuring reliable and predictable processing.

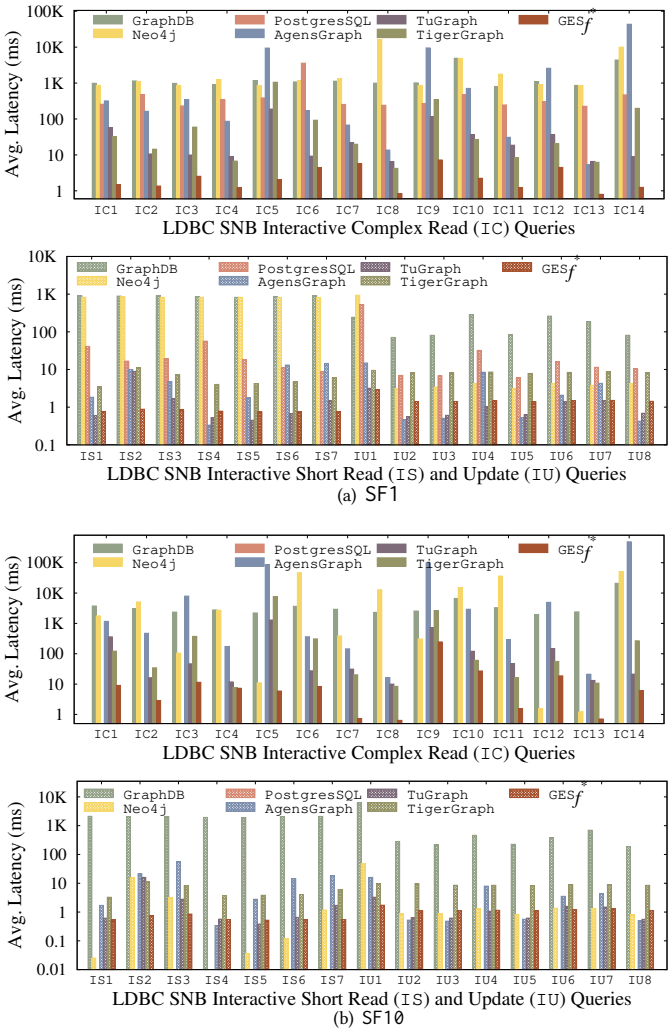


Figure 15: Performance of graph database systems on individual LDBC SNB workloads

Table 4: Throughput evaluation based on LDBC SNB Benchmark. *OOD*: out of disk space

SFs	Neo4j	Postgre	GraphDB	AgensGraph	TigerGraph	TuGraph	GES^*_f
SF1	852	433	32.9	525	10,012	29,405	113,687
SF10	62	*OOD*	14	52	3,739	9,952	65,718

6.3 Performance Evaluation with Existing Graph Database Systems

In this subsection, we compare the performance of GES with that of popular graph database systems.

Average Latency for IC, IS and IU. In Figure 15 we compare the average latency for each query on two small graphs SF1 and SF10 since the other systems performs badly on larger datasets as we will show later. PostgreSQL runs out of 5TB disk space (*OOD*) on SF10 and larger graphs where the results are not presented since it may flush the intermediate results to disk. The other results are consistent with another benchmarking study [42]. The experimental results demonstrate that GES^*_f significantly outperforms other

graph database systems across various workload types. On SF1, $\texttt{GES}_f^*$ consistently achieves the lowest average latency for complex read (IC) queries, often reducing execution time by up to three orders of magnitude compared to GraphDB, Neo4j, and TigerGraph. For instance, the latency on IC1 drops from 987 ms in GraphDB to just 1.5 ms in $\texttt{GES}_f^*$, and IC14 shows a reduction from 43,341 ms in AgensGraph to 1.25 ms in $\texttt{GES}_f^*$. Similarly, in short read (IS) and update (IU) queries on SF1, $\texttt{GES}_f^*$ maintains superior performance, with latencies rarely exceeding 2 ms compared to much higher values in other systems. On the larger SF10 graph, $\texttt{GES}_f^*$ continues to excel where other systems exhibit significantly higher latencies.

Throughput under LDBC Benchmark. We use the LDBC SNB benchmark to ensure a fair performance comparison with other systems. The throughput results shown in Table 4 clearly showcases GES's superior performance compared to existing graph database systems across two datasets. At SF1, GES achieves an impressive throughput of 113,687.1 queries per second, significantly outperforming the next best system, TuGraph, which records 29,405.4. This exceptional performance is maintained on the larger SF10, where GES delivers a robust throughput. In contrast, other systems such as Neo4j (62.3 q/s) and GraphDB (14.3 q/s) exhibit markedly lower performance. Additionally, TigerGraph and TuGraph show considerable drops in throughput from SF1 to SF10, with their performance consistently falling well below that of GES.

7 Related Work

Factorized Databases. Factorized databases aim to optimize query processing and data storage by representing data and query results in a compact, factorized form. This approach leverages redundancies and shared substructures within data to reduce the size of query outputs and improve computational efficiency. Bakibayev et al.[7] introduced foundational techniques for factorized representations of query results, demonstrating significant reductions in data size without loss of information. Olteanu and Schleich[29] further explored size bounds for these representations, providing theoretical guarantees and practical algorithms for their construction. Factorization has also been applied to probabilistic databases [30], enhancing the scalability of uncertain data management. Recent advancements include the development of factorized join algorithms that improve the performance of multi-way joins by avoiding redundant computations [36]. Additionally, factorized representations have been integrated into machine learning workflows to efficiently handle large datasets [35], enabling faster model training and inference. The integration of factorized databases with big data technologies has opened new avenues for scalable data analytics [26]. An vision of Kùzu [17] that has been proposed to use factorization in graph databases under m-n joins. Among them, GES is the first graph database that support evaluating most operators in a factorized manner.

Graph Databases. Graph databases are specialized systems designed to store, manage, and query graph-structured data, which is prevalent in domains like social networks, bioinformatics, and knowledge graphs. Systems such as Neo4j [27], Amazon Neptune [3], and JanusGraph [16] provide native graph storage and query capabilities, supporting languages like Cypher, Gremlin, and SPARQL. Research has focused on optimizing graph query processing [4], indexing [47], and distributed graph processing frameworks [23], enabling efficient analysis of large-scale graphs. Recent studies have explored challenges in graph data management, such as dynamic graph updates [13] and privacy-preserving graph analytics [43]. Graph embedding techniques [9] have gained significant attention, facilitating machine learning tasks on graph data by transforming nodes and edges into vector spaces. The emergence of graph neural networks [44] has further advanced the capability to perform deep learning on graph-structured data, enabling applications in recommendation systems, fraud detection, and more. Most application scenarios demand high-performance query processing, a capability that GES effectively provides.

Graph Database Workload and Benchmarks. Benchmarking is crucial for evaluating the performance and capabilities of graph database management systems (GDBMS). Unlike traditional relational databases, graph databases workload could be diverse [21]. Several benchmarks [2, 6, 8, 10, 25, 38] have been developed to address the unique challenges. Linked Data Benchmark Council (LDBC) Social Network Benchmark (SNB) [11] is an industry-neutral benchmark designed to facilitate the quantitative comparison of different graph data management solutions. It simulates user activity in a social network, providing a realistic and complex dataset for testing. Waterloo SPARQL Diversity Test Suite (WatDiv) [2] aims to evaluate the diversity of SPARQL queries over RDF data. It allows users to define custom schemas and generates datasets accordingly. While WatDiv provides valuable insights into system performance across different query types, it is limited to RDF data and SPARQL queries. The Large-Scale Subgraph Query Benchmark (LSQB) [25] focuses on testing GDBMS performance on global, unseeded subgraph matching queries. LSQB is designed to stress the query processing engine, particularly multi-way joins, by executing complex pattern matching queries without starting from specific nodes. While LSQB addresses an important class of queries, it lacks the breadth of workloads and choke-point analysis provided by LDBC SNB. Other benchmarks, such as the Train Benchmark (TB) [38] and the Graph Micro Benchmark [22], address specific use cases like validation scenarios and transactional operations, respectively. These graph database benchmarks are limited in scope, focusing on particular query types, data models, or use cases, whereas the LDBC SNB stands out for its comprehensive design.

8 Conclusions

The composable architecture of the GES query engine, combined with a factorized representation and operator-fusion optimizations, addresses the performance limitations of existing graph databases. By leveraging compact, tree-like in-memory structures, GES reduces memory overhead and enhances cache efficiency, allowing for efficient handling of billion-level graphs. The factorized execution and operator fusion further minimize data movement and processing latency, resulting in high throughput and low latency. Benchmark results, including those from the LDBC-SNB-DECLARATIVE , confirm that GES outperforms leading graph databases, establishing it as a powerful solution for large-scale, real-time graph processing.

References

[1] AgensGraph. 2025. AgensGraph Database. https://bitnine.net/agensgraph/. Accessed: 2025-04-13.

[2] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified Stress Testing of RDF Data Management Systems. In *ISWC (1) (Lecture Notes in Computer Science, Vol. 8796)*. Springer, 197–212.

[3] Amazon Web Services. 2025. Amazon Neptune. https://aws.amazon.com/neptune/. Accessed: 2025-04-13.

[4] Renzo Angles and Claudio Gutiérrez. 2011. An Introduction to Graph Data Management. *Information Systems* 36, 4 (2011), 223–246.

[5] Saeid Azadifar, Mehrdad Rostami, Kamal Berahmand, Parham Moradi, and Mourad Oussalah. 2022. Graph-based relevancy-redundancy gene selection method for cancer diagnosis. *Comput. Biol. Medicine* 147 (2022), 105766.

[6] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. 2017. gMark: Schema-Driven Generation of Graphs and Queries. In *ICDE*. IEEE Computer Society, 63–64.

[7] Nazim Bakibayev, Dan Olteanu, and Ján Závodný. 2012. Factorized Representations of Query Results: Size Bounds and Readability. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 325–336.

[8] Angela Bonifati, George H. L. Fletcher, Jan Hidders, and Alexandru Iosup. 2018. A Survey of Benchmarks for Graph-Processing Systems. In *Graph Data Management*. Springer, 163–186.

[9] Hongyun Cai, Vincent W. Zheng, and Kevin Chen-Chuan Chang. 2018. A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications. *IEEE Transactions on Knowledge and Data Engineering* 30, 9 (2018), 1616–1637.

[10] Orri Erling, Sebastian Auer, Michael Martin, Olaf Noppens, Juan Sequeda Trujillo, Sören Stephens, and Gavin Williams. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 619–630.

[11] Orri Erling, Alex Averbuch, Josep Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD Conference*. ACM, 619–630.

[12] 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, Kai Zeng, Kun Zhao, Jingren Zhou, Diwen Zhu, and Rong Zhu. 2021. GraphScope: A Unified Engine For Big Graph Processing. *Proc. VLDB Endow.* 14, 12 (2021), 2879–2892.

[13] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental Graph Pattern Matching. *ACM Transactions on Database Systems* 38, 3 (2013), 18:1–18:47.

[14] GraphDB. 2025. GraphDB. https://www.ontotext.com/products/graphdb/. Accessed: 2025-04-13.

[15] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Columnar Storage and List-based Processing for Graph Database Management Systems. *Proc. VLDB Endow.* 14, 11 (2021), 2491–2504.

[16] JanusGraph Community. 2025. JanusGraph: Distributed Graph Database. https://janusgraph.org/. Accessed: 2025-04-13.

[17] Guodong Jin, Xiyang Feng, Ziyi Chen, Chang Liu, and Semih Salihoglu. 2023. KÙZU Graph Database Management System. In *CIDR*. www.cidrdb.org.

[18] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *SIGMOD Conference*. ACM, 1695–1698.

[19] LDBC. 2021. LDBC SNB Datagen Hadoop. https://github.com/ldbc/ldbc_snb_datagen_hadoop.

[20] LDBC. 2021. LDBC SNB Specification. https://ldbcouncil.org/ldbc_snb_docs/ldbc-snb-specification.pdf.

[21] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, Xudong Wang, Huiming Zhu, Xuwei Fu, Tingwei Wu, Hongfei Tan, Hengtian Ding, Mengjin Liu, Kangcheng Wang, Ting Ye, Lei Li, Xin Li, Yu Wang, Chenguang Zheng, Hao Yang, and James Cheng. 2022. ByteGraph: A High-Performance Distributed Graph Database in ByteDance. *Proc. VLDB Endow.* 15, 12 (2022), 3306–3318.

[22] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. 2018. Beyond Macrobenchmarks: Microbenchmark-based Graph Database Evaluation. *Proc. VLDB Endow.* 12, 4 (2018), 390–403.

[23] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 135–146.

[24] Prashanth Menon, Todd C Mowry, and Andrew Pavlo. 2017. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment* 11, 1 (2017), 1–13.

[25] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. 2021. LSQB: a large-scale subgraph query benchmark. In *GRADES-NDA@SIGMOD*. ACM, 8:1–8:11.

[26] Shunsuke Nakai and Hiroyuki Kitagawa. 2019. Factorized Databases Meet Big Data: A Survey. *Journal of Information Processing* 27 (2019), 516–526.

[27] Neo4j. 2021. Neo4j Graph Database Platform. https://neo4j.com/. Accessed: 2021-10-01.

[28] Dan Olteanu and Maximilian Schleich. 2016. Factorized databases. *ACM SIGMOD Record* 45, 2 (2016), 5–16.

[29] Dan Olteanu and Nayden Schleich. 2015. Factorized Representations of Query Results: Size Bounds and Readability. *ACM Transactions on Database Systems* 40, 1 (2015), 2:1–2:44.

[30] Dan Olteanu and Ján Závodný. 2012. Factorized Representations of Uncertain and Probabilistic Databases. In *Proceedings of the VLDB Endowment*, Vol. 5. VLDB Endowment, 1796–1807.

[31] Dan Olteanu and Jakub Závodnỳ. 2015. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)* 40, 1 (2015), 1–44.

[32] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satyanarayana R. Valluri, Mohamed Zaït, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *Proc. VLDB Endow.* 16, 10 (2023), 2679–2685.

[33] PostgreSQL. 2025. PostgreSQL Database. https://www.postgresql.org/. Accessed: 2025-04-13.

[34] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph Databases: New Opportunities for Connected Data* (2nd ed.). O'Reilly Media, Inc.

[35] Nayden Schleich, Guoliang Huo, and Dan Olteanu. 2019. A Layered Aggregate Engine for Analytics Workloads. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 1642–1659.

[36] Nayden Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning Linear Regression Models over Factorized Joins. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 3–18.

[37] Eugene J. Shekita and Michael J. Carey. 1990. A Performance Evaluation of Pointer-Based Joins. In *SIGMOD Conference*. ACM Press, 300–311.

[38] Gábor Szárnyas, Benedek Izsó, István Ráth, and Dániel Varró. 2018. The Train Benchmark: cross-technology performance evaluation of continuous model queries. *Softw. Syst. Model.* 17, 4 (2018), 1365–1393.

[39] TigerGraph. 2025. TigerGraph Database. https://www.tigergraph.com/. Accessed: 2025-04-13.

[40] Ke Tu, Wei Qu, Zhengwei Wu, Zhiqiang Zhang, Zhongyi Liu, Yiming Zhao, Le Wu, Jun Zhou, and Guannan Zhang. 2023. Disentangled Interest importance aware Knowledge Graph Neural Network for Fund Recommendation. In *CIKM*. ACM, 2482–2491.

[41] TuGraph. 2025. TuGraph Database. https://tugraph.tech/. Accessed: 2025-04-13.

[42] Ran Wang, Zhengyi Yang, Wenjie Zhang, and Xuemin Lin. 2020. An Empirical Study on Recent Graph Database Systems. In *KSEM (1) (Lecture Notes in Computer Science, Vol. 12274)*. Springer, 328–340.

[43] Xindong Wu, Xingquan Zhu, Gong-Qing Wu, and Wei Ding. 2014. Data Mining with Big Data. *IEEE Transactions on Knowledge and Data Engineering* 26, 1 (2014), 97–107.

[44] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S. Yu Philip. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2021), 4–24.

[45] Mohamad Zamini, Hassan Reza, and Minou Rabiei. 2022. A Review of Knowledge Graph Completion. *Inf.* 13, 8 (2022), 396.

[46] Zhilun Zhou, Yu Liu, Jingtao Ding, Depeng Jin, and Yong Li. 2023. Hierarchical Knowledge Graph Learning Enabled Socioeconomic Indicator Prediction in Location-Based Social Network. In *WWW*. ACM, 122–132.

[47] Lijun Zou, Lei Chen, and M. Tamer Özsu. 2013. Graph Pattern Matching: A Survey. *Frontiers of Computer Science* 7, 3 (2013), 264–277.

A MISSING PROOFS

A.1 Proof of Lemma 4.4

We first prove the following lemma, which is crucial for proving Lemma 4.4.

Lemma A.1. *Given an $\mathcal{F}_\mathcal{T}$ and its root r, for any index $i \in \left[1, N_{\overset{r}{\mathcal{F}_\mathcal{B}}}\right]$, the relation $\mathcal{R}_r^i$ induced by r and i can be enumerated in $O\left(\left|\mathcal{S}\left(\mathcal{R}_{\mathcal{F}_\mathcal{T}}\right)\right|\right)$ delay and space.*

Proof. We prove by induction on the number n of nodes contained in an $\mathcal{F}_\mathcal{T}$.

Base case: For $n = 1$, $\mathcal{F}_\mathcal{T}$ degenerates into an *f-Block*, hence $\mathcal{R}_r^i = \overset{r}{\mathcal{F}_\mathcal{B}}{}^{[i]}$, which can obviously be output in $O\left(\left|\mathcal{S}\left(\mathcal{R}_{\mathcal{F}_\mathcal{T}}\right)\right|\right)$ time and space.

Inductive step: Assume that for an $\mathcal{F}_\mathcal{T}$ with up to n nodes, Lemma A.1 holds. Now consider an $\mathcal{F}_\mathcal{T}$ with $n+1$ nodes, let C_r be the set of children of $\mathcal{F}_\mathcal{T}$'s root r, and for each $u \in C_r$, denote by $\mathcal{F}_{\mathcal{T}_u}$ the sub-tree of $\mathcal{F}_\mathcal{T}$ rooted at u. Clearly, $\mathcal{F}_{\mathcal{T}_u}$ contains at most n nodes. By the inductive hypothesis, $\forall j \in I_{(r,u)}$, tuples in $\mathcal{R}_u^j$ can be listed in $O\left(\left|\mathcal{S}\left(\mathcal{R}_{\mathcal{F}_{\mathcal{T}_u}}\right)\right|\right)$ delay and space. Consequently, $\bigcup_{j \in I_{(r,u)}} \mathcal{R}_u^j$ adheres to the same enumeration delay and space since accessing different j values takes only constant time and no extra space. Furthermore, tuples in $\bigtimes_{u \in C_r} \bigcup_{j \in I_{(r,u)}} \mathcal{R}_u^j$ can therefore be enumerated by a simple nested loop algorithm in $O\left(\left|\bigcup_{u \in C_r} \mathcal{S}\left(\mathcal{R}_{\mathcal{F}_{\mathcal{T}_u}}\right)\right|\right)$ delay and space. Then by equation (1), we can perform a cartesian product operation between $\overset{r}{\mathcal{F}_\mathcal{B}}{}^{[i]}$ and every tuple in $\bigtimes_{u \in C_r} \bigcup_{j \in I_{(r,u)}} \mathcal{R}_u^j$, which makes $\mathcal{R}_r^i$ be enumerated in $O\left(\left|\bigcup_{u \in C_r} \mathcal{S}\left(\mathcal{R}_{\mathcal{F}_{\mathcal{T}_u}}\right)\right| + \left|\mathcal{S}\left(\overset{r}{\mathcal{F}_\mathcal{B}}\right)\right|\right) = O\left(\left|\mathcal{S}\left(\mathcal{R}_{\mathcal{F}_\mathcal{T}}\right)\right|\right)$ delay and space, where the equality follows from the *disjoint schema partition* property of $\mathcal{F}_\mathcal{T}$. □

With the above lemma and equation (2), it is now straightforward to verify that the tuples in $\mathcal{R}_{\mathcal{F}_\mathcal{T}}$ can be enumerated with $O\left(\left|\mathcal{S}\left(\mathcal{R}_{\mathcal{F}_\mathcal{T}}\right)\right|\right)$ delay and space, thereby completing the proof of Lemma 4.4.