GLogS: Interactive Graph Pattern Matching Query At Large Scale

Longbin Lai[1], *Yufan Yang[2]*, *Zhibin Wang[3]*, *Yuxuan Liu[2]*, *Haotian Ma[2]*, *Sijie Shen[1]*, *Bingqing Lyu[1]*, *Xiaoli Zhou[1]*, *Wenyuan Yu[1]*, *Zhengping Qian[1]*, *Chen Tian[3]*, *Sheng Zhong[3]*, *Yeh-Ching Chung[2]* and *Jingren Zhou[1]*

[1]Alibaba Group, China
[2]The Chinese University of Hong Kong, Shenzhen
[3]Nanjing University

Abstract

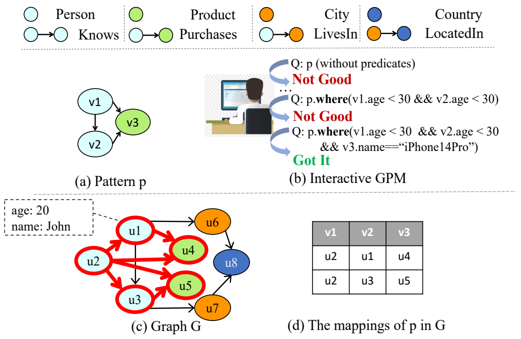
Interactive GPM (iGPM) is becoming increasingly important, where a series of graph pattern matching (GPM) queries are created and submitted in an interactive manner based on the insights provided by the prior queries. To solve the iGPM problem, three key considerations must be taken into account: performance, usability and scalability, namely if results can be returned in a timely manner, if queries can be written in a declarative way without the need of imperative fine-tune, and if it can work on large graphs. In this paper, we propose the GLogS system that allows users to interactively submit queries using a declarative language. The system will compile and compute optimal execution plans for the queries, and execute them on an existing distributed dataflow engine. In the evaluation, we compare GLogS with the alternatives systems Neo4j and TigerGraph. GLogS outperforms Neo4j by 51× on a single machine due to better execution plans. Additionally, GLogS can scale to processing large graphs with distributed capability. While compared to TigerGraph, GLogS is superior in usability, featuring an optimizer that can automatically compute optimal execution plans, eliminating the need of manual query tuning as required in TigerGraph.

1 Introduction

Graph pattern matching (GPM) aims to compute the mappings in a data graph that match a given small pattern graph, and it plays an important role in a variety of applications covering bioinformatics [3, 24, 37], chemistry [17, 23], social/web network analysis [20, 26], and recently in enhancing the expressive power of graph neural network [15, 53, 55].

Increasingly, interactive GPM (iGPM) is becoming critical for data scientists to mine relationships, identify frauds or detect intrusions from a variety of large graphs in real life. In these scenarios, data scientists create and

[*]Equal Contribution.


Person
Knows
Product
Purchases
City
LivesIn
Country
LocatedIn
v1
v2
v3
(a) Pattern p
Q: p (without predicates)
Not Good
...
Q: p.**where**(v1.age < 30 && v2.age < 30)
Not Good
Q: p.**where**(v1.age < 30 && v2.age < 30 && v3.name==“iPhone14Pro”)
Got It
(b) Interactive GPM
age: 20
name: John
u1
u2
u3
u4
u5
u6
u7
u8
(c) Graph G
(d) The mappings of p in G

v1	v2	v3
u2	u1	u4
u2	u3	u5

Figure 1: Example of interactive graph pattern matching, where a user will interactively submit graph pattern matching queries (see Example 2.1) to explore the graph.

submit a series of GPM queries in an interactive manner based on the insights provided by the results of prior queries. For example, we demonstrate a simplified application scenario as follows.

Example 1.1. In Figure 1, a user is exploring recommendation rules in an e-commerce graph, which maintains relationships such as “Purchases” between persons and products. The user is specifically looking at a pattern (Figure 1(a)) of co-purchasing among pairs of people who are acquainted. If such co-purchasing occurs very frequently among these pairs in historical data, a recommendation rule may be created to recommend the product that has been purchased by one person to his/her friends who have not yet purchased it. The user tries different patterns and constraints (using predicates) through a series of interactive queries. In Figure 1(b), the user decides to create a rule suggesting that young people (“*age* < 30”) tend to co-purchasing the “iPhone 14 Pro”, rather than an arbitrary product.

In the above scenario of iGPM, it is necessary to consider the requirements of *performance*, *scalability* and *usability* simultaneously. Performance allows users to quickly obtain useful insights from the “trial-and-error” process. Usability enables users to easily present arbitrary GPM queries. Scalability is also crucial as it is now common to handle large-scale graphs. Due to the computation-intensiveness of GPM queries [14, 29], it is already difficult for a graph expert to tune the execution [13, 25, 54]. The problem becomes even more complex in iGPM, where users may not be experts and the queries can involve intricate patterns and optional predicates. Therefore, the following features are essential to meet the above requirements.

Declarative Language. A declarative query language can provide users convenience and flexibility to express complex GPM queries.

Automatic Optimization. Automatic optimization allows non-expert users to focus on exploring valuable patterns without having to worry about the challenging task of performance tuning.

Distributed Execution. With the graph partitioned across the cluster, distributed execution is expected to spread the workloads accordingly.

However, the systems that are potentially usable for iGPM, including Neo4j [33] and TigerGraph [18], all fall short in providing one or more above features (Section 2). In this paper, we propose the GLogS (name after GLogue, see Section 5.3) system to fill in the gap. Our goal is to give the users the convenience and flexibility of presenting GPM queries interactively, and to have the system deal with the complexities that arise from compilation, query optimization, and distributed execution. We mainly make the following technical contributions.

(1) A compilation stack that compiles declarative GPM queries into distributed programs. We adopt Gremlin's declarative `match` step, an industrial-strength query language, for expressing GPM queries. The `match` step, after compilation and optimization, will be transformed into a program that can be executed on a distributed dataflow engine.

(2) An optimizer that can automatically derive optimal execution plans for GPM queries. While analyzing the execution plans of Neo4j, we have identified two critical impact factors of deriving good execution plans for GPM: worst-case optimal execution plan [4] and high-order statistics [12]. We take into considerations both factors to design and implement the optimizer for GLogS.

(3) A system that allows users to interactively submit and efficiently execute GPM queries at large scale. We build the GLogS system upon the existing distributed dataflow engine GAIA [38] to leverage its optimization for graph queries. In the evaluation based on the LDBC benchmark, we compare GLogS with Neo4j and TigerGraph. GLogS outperforms Neo4j by 51$\times$ on a single machine due to better execution plans. Additionally, GLogS can scale to handle large graphs with distributed capability. While compared to TigerGraph, GLogS is superior in usability, featuring an optimizer that can automatically compute optimal execution plans, eliminating the need of manual query tuning as required in TigerGraph.

2 Background and Challenges

2.1 The Problem of iGPM

We adopt the property graph model [6] for usability. A property graph $G(V_G, E_G)$ is a directed labelled graph, as shown in Figure 1, in which each vertex $u \in V_G$ models an entity, and each edge $(u_s, u_t) \in E_G$ models the relationship from a source vertex u_s to a target vertex u_t. We call the edge (u_s, u_t) the adjacent edge of u_s and u_t, and u_s (resp. u_t) is an in neighbor (resp. out neighbor) of u_t (resp. u_s). A vertex u (an edge is analogously defined) is assigned a globally unique identifier (Id) and a label (Label) to indicate is type. Moreover, it can carry a collection of key-value pairs as the properties. We use $u.\mathsf{key}$ to denote accessing u's property of given key.

A pattern $p(V_p, E_p)$ is a small *connected* graph. Given a pattern p and graph G, the graph pattern matching (GPM) problem aims to compute all mappings $Q_G(p)$ of the pattern in the graph G, where each mapping $f \in Q_G(p)$ matches the pattern vertices[1] to a set of non-duplicate graph vertices one by one, so that if there is a pattern edge between two pattern vertices, there must be a graph edge between the two matched graph vertices. For a pattern vertex v, we use $f(v) = u$ to obtain the matched graph vertex u. The number of mappings is called the frequency of the pattern in the graph, denoted as $\mathscr{F}_G(p)$. When the context is clear, the subscript of G in above notations may be omitted (i.e. $Q(p)$). Predicates can be specified, while we mostly omit predicates to focus on the pattern in the paper. Details of how we handle predicates are in our open-source page [46].

Example 2.1. In Figure 1, there are two mappings of the triangle pattern p in the graph as shown, and thus the frequency of p is 2. Specifically, a mapping f matches v_1 to u_2 and v_2 to u_1, namely $f(v_1) = u_2$ and $f(v_2) = u_1$. Obviously, there is a graph edge (u_2, u_1) corresponding to the pattern edge (v_1, v_2). The predicate "$v_1.age < 30$ && $v_2.age < 30$" in Figure 1(b), constrains that the vertices matching v_1 and v_2 must have "age" smaller than 30.

We target the iGPM scenario to process GPM queries on large-scale property graphs in an interactive context.

2.2 Solving iGPM using Existing Systems

Graph databases [2, 8, 18, 27, 33] allow users to interactively query the graph using declarative query languages,

[1]The details of matching edges are not discussed as they are similar to matching vertices.

and thus have the most potential to be deployed for iGPM. However, they often lack support for automatic optimization or distributed execution, and thus cannot meet performance, usability and scalability at the same time. We discuss Neo4j and TigerGraph as representatives. Other related systems are surveyed in Section 8. Neo4j [33] is one of the most popular graph databases, but is limited by its single-machine design and insufficient optimizer, leading to poor scalability and performance as reported in previous studies [38, 44, 48]. TigerGraph [18], on the other hand, is a distributed system that can scale well but lacks an automatic optimizer, requiring users to manually tune the plan for good performance, which significantly limits its usability. In addition, users need to pre-install the queries before they can be executed on TigerGraph. The pre-installation involves native code generation and compilation, which can take 1 to 3 minutes per query in our evaluation and may not be tolerable in the interactive context.

2.3 Challenges of Solving iGPM

It's challenging to develop an iGPM system with performance, usability and scalability. Here, we discuss the issues arise from query compilation and optimization.

Compilation. Compiling a declarative query language itself is non-trivial, and the interactive context introduces extra difficulties. Due to the timeliness of iGPM, it's infeasible to generate native codes from the queries and process a time-consuming online compilation like TigerGraph [18]. Preparing store procedurals for a set of queries offline is not possible, as the useful patterns remain unknown in advance.

Optimization. The automatic optimization of GPM lies at the core of an iGPM system, but it's difficult to design such an optimizer for real-life queries. To see this, in Figure 2, we've demonstrated the execution plans Plan_G and Plan_N of a benchmark query derived by the optimizers of GLogS and Neo4j, respectively. For now, one only needs to know that a better execution plan typically, if not definitely, produces less intermediate results, which are a collective of the mappings of all intermediate patterns that must be computed during the execution. We mark in Figure 2 the corresponding intermediate pattern frequencies in the benchmark graph G_1 (Table 1). Obviously, Plan_N produces much larger intermediate results than Plan_G. We execute the two plans in our system, and Plan_N not only runs orders of magnitude slower, but also consumes significantly larger memory, than Plan_G. This demonstrates that Neo4j's optimizer is insufficient to handle such complex GPM queries.

There are two main reasons for this. Firstly, the execution plan given by Neo4j cannot guarantee *worst-case optimality*. Secondly, it uses only *low-order statistics* to

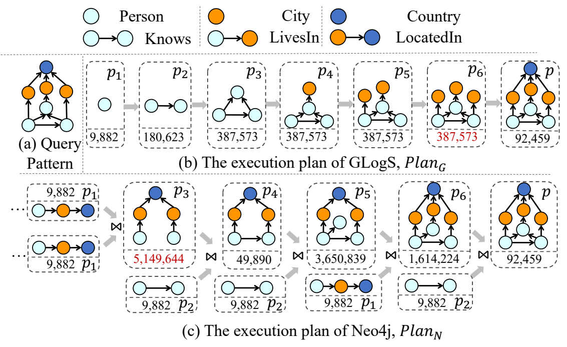


Figure 2: Execution plans of GPM. Certain trivial steps in the plans are not shown for clarity.

estimate the cost of a plan, which can cause the resulting plan to have small estimated cost even though it actually costs heavily. We elaborate in details.

Worst-case optimality. An execution plan for computing a pattern p is worst-case optimal, if the frequency of any *intermediate pattern* in the plan does not exceed $\mathscr{F}(p)$ in the worst case. Here, an intermediate pattern in an execution plan refers to a subgraph (or sub-pattern, interchangeably) of the queried pattern whose mappings must be computed during the execution. For instance, the patterns p_1-p_6 in Plan_G (all plans in this section are referred to Figure 2) are intermediate patterns of p. The process of solving GPM typically involves operations of binary join and vertex expansion. Briefly, binary join involves performing a hash join on the mappings of two input patterns in order to produce the results of the output pattern. For example, in Plan_N, p_2 is joined with p_3 to produce p_4. Execution plans that rely *solely* on binary joins, such as Plan_N, are called binary-join plans. However, these plans may not guarantee worst-case optimality [29]. Neo4j's plan actually falls into this category, which is a dominant factor of its poor performance [48].

Alternatively, Ammar et al.[4] have looked into Ngo's algorithm[35] for GPM optimization. The key operation to this algorithm is vertex expansion, which involves expanding a base pattern by adding one more vertex to it. In Plan_G, for instance, p_2 is expanded to p_3 in this way. Begin with a base pattern that is a vertex, the algorithm processes vertex expansions iteratively until the desired pattern is obtained. According to Ngo's algorithm, we can obtain an execution plan that is worst-case optimal, such as Plan_G for GPM.

Recent research [1, 32, 54] has shown that the best possible execution plans for GPM must incorporate both binary joins and vertex expansions. Consequently, our optimizer must be capable of handling this hybrid strategies. It's crucial to note that to achieve worst-case optimality, such hybrid plans must carefully consider the use of binary joins, as will be explained in Section 5.4.

High-order statistics. An optimal execution plan for a GPM query is the plan that has the smallest cost. The pattern frequencies are essential for evaluating the cost of an execution plan. In large-scale graphs, it is more feasible to estimate these frequencies, rather than trying to exactly compute them. Neo4j uses Low-order statistics such as the number of vertices and edges (of each type) to estimate pattern frequencies by assuming independent existence of graph edges [34]. However, this assumption is too idealistic and can lead to inaccurate cost estimation and poor execution plans in practice. To address this issue, we follow Mhedhbi et al. [32] to exploit the *high-order statistics* [12] of the graph.

Definition 2.1. The high-order statistics of a graph refer to the frequencies of a series of small patterns (also known as motifs [3]), from the smallest single-vertex patterns to the largest patterns that are *complete* graphs of k vertices. Here, k is called the high-order *level*, and must be at least 3 to avoid degrading to low-order.

Mhedhbi et al. [32] and us have both demonstrated the effectiveness of the high-order statistics. However, their computation is at least as costly as the widely recognized computation-intensive workload of graph pattern mining [45]. To reduce the computation cost, Mhedhbi et al. [32] have proposed using a sampling technique that matches a randomly selected portion of data vertices and edges at runtime. Nevertheless, the sampling technique is difficult to apply to large-scale graphs that are partitioned in the cluster.

3 System Overview

We've built the GLogS system for iGPM, as shown in Figure 3, to address the challenges in Section 2.3. The system allows users to interactively submit their GPM queries using Gremlin's declarative `match()` step [7]. An example of the Gremlin code for the triangle pattern is presented. Because GPM is computationally intractable [28], users can optionally specify a timeout to prevent the query from running for an unreasonable amount of time. It is worth noting that relational operations, including projection, ordering and grouping, may also be applied in iGPM. However, for queries that involve these operations, GLogS processes them only after the execution of GPM. Therefore, their computation complexity is dominated by that of GPM, and will not be further discussed in this paper.

Frontend Module. The frontend machine runs the processors of a *pattern parser*, a *plan optimizer* and a GLogue *manager* for parsing and optimizing the GPM queries. The pattern parser parses a Gremlin query into a *language-agnostic* structure called PatternDesc. The

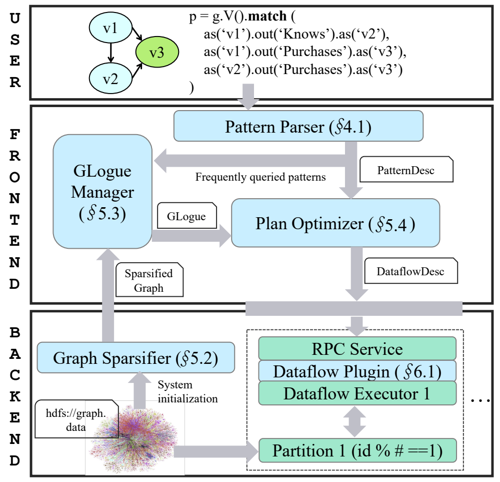

```
p = g.V().match (
    as('v1').out('Knows').as('v2'),
    as('v1').out('Purchases').as('v3'),
    as('v2').out('Purchases').as('v3')
)
```

Figure 3: System overview. Blue components are the focus of this paper, with the paper's sections indicated.

key objective of designing PatternDesc is to decouple the query language and the optimizer. This enables easy integration with other query languages such as Neo4j's Cypher [16]. For given PatternDesc, the plan optimizer aims to produce an execution plan that has the smallest-possible cost based on a generic cost model. The cost model considers the high-order statistics of the graph, which are maintained in a novel graph-based structure called GLogue. The GLogue manager handles the construction of the GLogue by computing the frequencies on the sparsified graph for patterns up to k vertices (i.e. with high-order level k) when the system is initiated. It also buffers the frequently queried patterns whose frequencies are missing from the GLogue, and launches a procedural periodically to append these frequencies to the GLogue.

Backend Module. The backend module consists of a *distributed dataflow engine*, a *graph store* and a *graph sparsifier*, spreading across a cluster of n computing nodes. We have built the GLogS system on an existing dataflow engine, which organizes n executors, each corresponding to a computing node in the cluster. A declarative DataflowDesc is constructed from the execution plan produced by the plan optimizer, which embeds the computing instructions of GPM in a dataflow that is a directed acyclic diagram (DAG). The DataflowDesc is then distributed to all executors via RPC services to launch the computation in parallel. To make the DataflowDesc executable, a library of dataflow plugin is implemented that contains the generated code and a job assembler to assemble the distributed program. The dataflow plugin is required to co-compile offline with the underlying dataflow engine, which bypasses a costly online native-code compilation as TigerGraph [18].

The graph store manages the partitioned graph data in the cluster. As it is not the main focus of this paper, for simplicity, we adopt in-memory and immutable graph store, where the graph data is partitioned using a hash partition strategy, namely the vertex u will be placed on the partition of "u.Id % # partitions", together with all its properties and adjacent (both in and out) edges. Such a simple yet widely used partition strategy[4, 29, 38] may lead to load skew, which can potentially impact query performance. Nonetheless, a well-optimized execution plan is still the key to the efficiency and scalability of GPM. Therefore, rather than exploring alternative partition strategies [52], we employ the simple strategy and focus on query optimization in this work. The i[-th] partition of the graph is co-located with the i[-th] executor of the dataflow engine. Moreover, The raw graph data are pre-partitioned, encoded and stored in a distributed file system such as *HDFS*. Each dataflow executor loads its partition into the main memory while starting up the system. During system initialization, a graph sparsifier will be simultaneously launched as loading graph data, which is responsible for sparsifying the large-scale graph into a small graph that can fit into the main memory of the frontend machine. The sparsified graph will be serialized to a persistent store to prevent the need for re-sparsification when the system is restarted.

4 Compiling Declarative GPM Queries

We demonstrate in Figure 4 the process of compiling the declarative Gremlin's `match()` step for a GPM query into distributed dataflow program.

4.1 The Pattern Parser

As shown in Figure 4(a), in Gremlin's `match()` step, a pattern is described as a collection of clauses in the form of "`as().[in|out]().as()`", in which the start and end `as()` steps identify the two vertices with tags that are unique in the pattern, and the `in()` or `out()` step in between expresses the edge that connects the two vertices. In this simplest[2] form of a `match()`, each clause expresses an edge in the pattern. Given a Gremlin's `match()` step, the pattern parser utilizes the ANTLR tool [21] (officially provided by Gremlin) to parse a query into an Abstract Syntax Tree, from which a `PatternDesc` is built, as illustrated in Figure 4(b).

We first define two *computing primitives* called `GetV` and `GetE` for describing GPM queries. A `GetV` primitive is a 4-tuple (eTag, tag, label, [Source|Target]) that encodes the semantics of matching the vertex with "tag" as the source or target vertex of an "eTag" edge. A `GetE`(vTag, tag, label, [In|Out]) encodes matching the edge with "tag" as the in or out edge of a "vTag" vertex.

A sentence that is an ordered sequence of `GetV` and `GetE` is then used to encode the semantics of a clause in `match()`, and a `PatternDesc` is composed of a collection of sentences. The semantics are self-explanatory, and we just discuss some special use cases in the first sentence of Figure 4(b). Observe that the first `GetV` has the "eTag" field unspecified (`NA`), which means that the vertex may not be bound to any prior edge and should match all vertices in the graph. In the `GetE`, the "tag" field is specified as an empty `String`. This tells the runtime that the matched edge (also applied for `GetV`) should not be kept in the results, which is useful when only a part of the matched instances are needed in practice. Following the `GetE`, a `GetV` has an empty "eTag", which means the vertex must be obtained directly from this "previous" edge.

Observe that we include the label information in `GetV` that is not actually given in the Gremlin query. In GLogS, while loading the graph data, we can meanwhile extract the *meta connections* that maintain the possible types of source and target vertices of each edge type. For example, a Purchases edge can only connect a Person to a Product. Such meta connections not only help us validate user queries, but also reduce the number of patterns stored in the GLogue (Section 5.3).

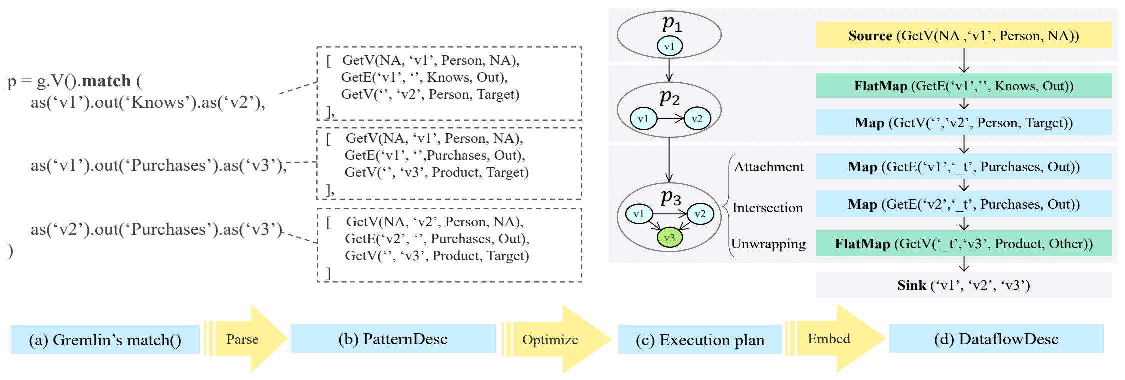
4.2 The Dataflow Embedding

In the plan optimizer (Section 5.4), an optimized execution plan will be computed for the `PatternDesc`, which will be further embedded into a `DataflowDesc` that describes how to compute the pattern in a dataflow engine.

In a dataflow engine, a dataflow is a directed acyclic graph (DAG) that abstracts the computation, in which a vertex stands for an operator that defines the computing logic, and an edge between two operators o_1 and o_2 represents the data channel such that the output of o_1 is the input of o_2. In the task of GPM, the input and output data of each operator in a dataflow are mappings of the patterns. We introduce five operators in this paper:

- `Source(udf)`: A `Source` operator specifies the input data of the dataflow program, which are a collection of vertices in the graph.
- `Sink(udf)`: The `Sink` operator (only one allowed) writes the results to the output channel (e.g. an RPC port) that users can access.
- `Map(udf)`: For each input item, a `Map` operator computes *exactly* one data item using the given user-defined function (udf).
- `FlatMap(udf)`: For each input item, a `FlatMap` operator can produce *arbitrary* (none, single or multiple) number of data items via the udf.
- `Join(key1, key2, udf)`: A `Join` operator consumes two input data, extracts the corresponding

[2]The other more complex forms only bring in engineering details.



```
p = g.V().match (
    as('v1').out('Knows').as('v2'),

    as('v1').out('Purchases').as('v3'),

    as('v2').out('Purchases').as('v3')
)
```

```
[   GetV(NA, 'v1', Person, NA),
    GetE('v1', '', Knows, Out),
    GetV('', 'v2', Person, Target)
],
[   GetV(NA, 'v1', Person, NA),
    GetE('v1', '',Purchases, Out),
    GetV('', 'v3', Product, Target)
],
[   GetV(NA, 'v2', Person, NA),
    GetE('v2', '', Purchases, Out),
    GetV('', 'v3', Product, Target)
]
```

```
Source (GetV(NA ,'v1', Person, NA))
FlatMap (GetE('v1','', Knows, Out))
Map (GetV('','v2', Person, Target))
Map (GetE('v1','_t', Purchases, Out))
Map (GetE('v2','_t', Purchases, Out))
FlatMap (GetV('_t','v3', Product, Other))
Sink ('v1', 'v2', 'v3')
```

Figure 4: The process of compiling a Gremlin's `match` step into a `DataflowDesc`.

keys, and conducts a join via the `joinFunc` on the two input data.

In a nutshell, a `DataflowDesc` for GPM is a dataflow that embeds the GPM primitives of such as `GetV` and `GetE` in the above operators as the udfs. Figure 4(d) illustrates a `DataflowDesc` for computing the triangle pattern. We look into the first three operators for now, which describe the computation of the mappings for v_1 and v_2. First, v_1 is matched in a `Source` operator that consumes all Person vertices from the graph. Then for each vertex that matches v_1, a `FlatMap` operator is assigned to traverse its out edges, which is reasonable as a vertex typically has more than one adjacent edges in the graph. The last `Map` operator extracts the target vertex from each edge to match v_2. More details of how we compute the given `DataflowDesc` will be discussed in the next section.

5 Automatic Optimization

This section covers the automatic optimization of GPM queries, which is a collaboration of the system components of graph sparsifer, GLogue manager, and plan optimizer. For a graph vertex u, we denote $\mathsf{Nbr}[elabel](u)$ as the neighbors[3] of u in the graph G subject to the edge label constraint. Given two graphs G_1 and G_2, G_2 is a subgraph of G_1, denoted as $G_2 \subseteq G_1$, if $V_{G_2} \subseteq V_{G_1}$ and $E_{G_2} \subseteq E_{G_1}$. Furthermore, G_2 is an induced subgraph of G_1, if it contains *all* edges in G_1 among V_{G_2}. For two sets S_1 and S_2, we denote $S_1 \setminus S_2$ as the set of elements in S_1 but not in S_2.

5.1 Execution Plan and Cost Model

We first introduce the execution plan for GPM and define its cost, which allows us to search for the optimal execution plan as the one with the smallest cost.

[3]Note that the edges can be in out, in or even both directions, but we omit the direction in the notation for simplicity.

Execution Plan. To allow the optimizer to derive hybrid execution plans as mentioned in Section 2.3, we consider two basic operations: the *binary join* and *vertex expansion* that are critical to fulfil the binary joins and worst-case optimal joins, respectively. A binary join, denoted as $\mathsf{Join}(\{p_{s_1}, p_{s_2}\} \to p_t)$, conducts hash join operation for $Q(p_{s_1})$ and $Q(p_{s_2})$ on the join key of $V_{p_{s_1}} \cap V_{p_{s_2}}$ to compute the results of $Q(p_t)$. The operation of vertex expansion needs further explanation.

Definition 5.1. Consider two patterns, p_s and p_t with $V_{p_t} \setminus V_{p_s} = \{v\}$, and $E_{p_t} \setminus E_{p_s} = \{e_1 = (v_1, v), e_2 = (v_2, v), \dots, e_k = (v_k, v)\}$ without loss of generality. A vertex-expansion operation, denoted as $\mathsf{Expand}(p_s \to p_t)$, extends each mapping f of p_s by one more graph vertex corresponding to v. The newly matched graph vertex must be in the common neighbors of all $f(v_i)$ for $1 \le i \le k$, namely $\bigcap_{i=1}^{k} \mathsf{Nbr}[e_i.\mathsf{Label}](f(v_i))$.

Example 5.1. In Figure 4(c), it's clear that $V_{p_3} \setminus V_{p_2} = \{v_3\}$. This allows us to perform a vertex expansion $\mathsf{Expand}(p_2 \to p_3)$. We use the graph in Figure 1 to illustrate the process. For a given mapping $f = \{u_2, u_1\}$ of p_2, we can expand it to the mapping $\{u_2, u_1, u_4\}$ for p_3. Here, v_3 is matched to $\{u_4\}$, which is obtained by intersecting $\mathsf{Nbr}[\mathsf{Purchases}](u_2)$ and $\mathsf{Nbr}[\mathsf{Purchases}](u_1)$. We can similarly perform this process for the other mappings $\{u_1, u_3\}$ and $\{u_2, u_3\}$.

Given a queried pattern p, we denote an execution plan for computing p as $\mathsf{Plan}(p) = (\Phi = \{p_1, p_2, \dots, p_n = p\}, \Gamma = [\tau_1, \tau_2, \dots, \tau_m])$, where Φ represents a set of intermediate patterns and Γ is an ordered sequence of operations that can be either binary join or vertex expansion. For example, we have the execution plan in Figure 4(c) as $(\Phi = \{p_1, p_2, p_3\}, \Gamma = [\mathsf{Expand}(p_1 \to p_2), \mathsf{Expand}(p_2 \to p_3)])$.

Cost Model. With the execution plan $\mathsf{Plan}(p) = (\Phi, \Gamma)$, we propose the cost model as

$$\mathsf{Cost}(\mathsf{Plan}(p)) = \sum_{p' \in \Phi} \mathscr{F}(p') + \sum_{\tau \in \Gamma} \mathsf{Cost}(\tau). \quad (1)$$

The first part refers to the cost of accessing the intermediate results from the memory, which can be considered as the commmunication cost, namely, the cost of accessing remote memory. This is because the intermediate results are a collection of the output from all executors in the cluster, and the cost of accessing remote data is much greater than that of accessing local data. The second part stands for the cost of the operations, also known as the computation cost.

As any join algorithm must go through the data of both participants, the cost of a binary join is computed as

$$\mathsf{Cost}(\mathsf{Join}(\{p_{s_1}, p_{s_2}\} \to p_t)) = \alpha_j(\mathscr{F}(p_{s_1}) + \mathscr{F}(p_{s_2})), \tag{2}$$

where α_j is a normalized factor. We do not consider the joined results in Equation 2 because it must have been considered as the communication cost in Equation 1.

Consider a vertex expansion $\mathsf{Expand}(p_s \to p_t)$ in Definition 5.1, and let f be one mapping of p_s. The cost of the vertex expansion of f is dominated by intersecting the neighbors of $f(v_i)$ for $1 \leq i \leq k$, which has the complexity of $\sum_{i=1}^{k} |\mathsf{Nbr}[e_i.\mathsf{Label}](f(v_i))|$. Regarding e_i, let $\sigma_{e_i}(f) = |\mathsf{Nbr}[e_i.\mathsf{Label}](f(v_i))|$ be the vertex-expansion factor of the mapping f, and $\overline{\sigma_{e_i}}$ the average factor of all mappings. We have the cost of vertex expansion as

$$\begin{aligned}\mathsf{Cost}(\mathsf{Expand}(p_s \to p_t)) &= \alpha_{ve} \sum_{f \in Q(p_s)} \sum_{i=1}^{k} \sigma_{e_i}(f) \\ &= \alpha_{ve} \mathscr{F}(p_s) \sum_{i=1}^{k} \overline{\sigma_{e_i}},\end{aligned} \tag{3}$$

where α_{ve} is a normalized factor that, along with α_j in Equation 2, aligns the differences in computation cost of vertex expansion and binary join, as well as the communication cost and the computation cost.

5.2 Graph Sparsifier

While it is necessary to compute $\mathscr{F}(p)$ of any pattern p for Equation 1, it's cost-prohibitive to do so directly. The sampling technique proposed in [32] cannot be applied to large-scale graphs (see Section 2.3). Therefore, we explore the technique of graph sparsification [40, 49]. Specifically, during system initialization, the graph sparsifier will conduct sparsification on each partition of the graph to randomly preserve a subset of edges, and aggregate them at the frontend machine to form the sparsified graph G^*. It is obviously more feasible to compute the pattern frequencies on G^* than on the original graph G. Thus, we use $\mathscr{F}_{G^*}(p)$ (with normalization) as an estimation of $\mathscr{F}_G(p)$ for cost evaluation.

However, it's non-trivial to sparsify real-life graphs that can contain many different types of edges. A naive uniform sparsification [40, 49] adopts a uniform *sparsification ratio* (the probability of keeping an edge) for all edges during sparsification. Although such a naive approach can obtain unbiased estimation of $\mathscr{F}_G(p)$, but it still works poorly in our evaluation (Section 7). The main reason is that different types of edges can appear in rather skewed frequencies in real-life graphs. A less frequent type of edge, such as the $\mathsf{LocatedIn}$ edge in Figure 1 that appears in thousands compared to the $\mathsf{Purchases}$ edge in billions, is more likely to get eliminated during sparsification, causing the estimation to have large variance.

We also notice that there are sparsification [11, 42] and coarsening [31] algorithms based on spectral graph theory, aiming to offer a superior approximation via biased sampling. However, these algorithms emphasize preserving global statistics such as edge cut, rather than counting subgraphs that are local information. As a result, they may not be suitable for our task.

Regarding our task, we adopt the stratified sparsification [19] that treats each type of edges as an independent stratum, and assign each stratum an individual sparsification ratio. The stratified sparsification provides the flexibility in choosing the sparsification ratio, and we propose an optimization problem that aims to minimize the estimation variance through tuning the ratio. Before proposing our optimization problem, we first introduce the norm factors and discuss its unbiasedness. Let the sparsification ratios be $\Omega = \{\rho_1, \rho_2, \ldots, \rho_l\}$, where ρ_i denotes the ratio for the stratum of edges with label i without loss of generality. The following lemma holds.

Lemma 5.1. *Let* $\widetilde{\mathscr{F}(p)} = \prod_{e \in E_p} \frac{1}{\rho_{e.\mathsf{Label}}} \mathscr{F}_{G^*}(p)$*. We have* $\mathbb{E}[\widetilde{\mathscr{F}(p)}] = \mathscr{F}_G(p)$*, where* $\mathbb{E}[X]$ *denotes the expected value of a random variable* X*.*

Proof is in Section A.1.1. In the following, when we write $\mathscr{F}(p)$, we by default mean $\widetilde{\mathscr{F}(p)}$ if not otherwise specified.

The next question is how to specify the sparsification ratios. Given that the sparsified graph must reside in the frontend machine, we model an optimization problem subject to the memory constraint M, that minimizes the variance of the frequency estimation regarding a forged pattern p^* formed by all types of edges in the graph, as:

$$\begin{aligned}&\arg\min_{\Omega} \mathrm{Var}[\widetilde{\mathscr{F}(p^*)}] \\ &\text{s.t.} \sum_{i=1}^{l} s_i \rho_i \leq M,\end{aligned} \tag{4}$$

where s_i denotes the frequency of the edges that have label i. The optimization problem achieves its optimum under the following condition:

$$\rho_i = \min(1, \frac{M}{l} \times \frac{1}{s_i}),$$

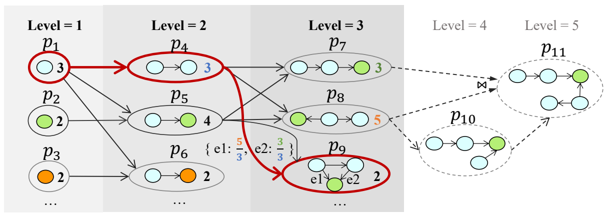


Figure 5: A fragment of GLogue. The value aside the pattern indicates its frequency in the graph in Figure 1(c). Note that the patterns in "Level=4/5" do not actually present in the GLogue.

Detailed derivation can be found in Section A.1.2. Interestingly, if $s_i \ll M$, ρ_i must be set to 1, which means that a very infrequent type of edge, such as the above LocatedIn edges, must not be ruled out during sparsification.

5.3 GLogue Manager

Our system aims to automatically derive the optimal execution plan for any arbitrary pattern. To do so, we follow the methodology presented in [32] to calculate the high-order statistics (Definition 2.1) of the graph. However, the approach in [32], which employs a table-based *catalogue* for retaining the high-order statistics, is not only expensive to construct but also challenging to apply when there are numerous complex pattern relationships in real-world graphs that need to be recorded, thereby making plan searching a difficult task. Instead, we have recognized the intrinsic suitability of graph structure for retaining complex relationships of this sort, and thus, we have proposed GLogue as a graph-based catalogue.

The GLogue is a hierarchical property graph as shown in Figure 5, in which each vertex is a pattern p at level $|V_p|$ with its frequency $\mathscr{F}(p)$ as the property. To ease lookup, the pattern will be encoded as a `String` using the technique of canonical labelling [10]. We say that the GLogue has k levels, if it maintains the high-order statistics up to level k. There're two types of edges in GLogue. The first type connects p_s and p_t in GLogue, if p_s can be expanded to p_t via vertex expansion. Regarding $\mathsf{Expand}(p_s \to p_t)$, the edge (p_s, p_t) records the vertex-expansion factors $\overline{\sigma_e}$ for all $e \in E_{p_t} \setminus E_{p_s}$. The second type corresponds to a binary join $\mathsf{Join}(\{p_{s_1}, p_{s_2}\} \to p_t)$, which introduces one edge from p_{s_1} to p_t with $(p_{s_2}, \mathscr{F}(p_{s_2}))$ as the property.

We deploy the GLogue manager in GLogS for the construction and maintenance of GLogue. When the system is initiated, with a threshold of level k, GLogue will be constructed from scratch to include all valid patterns with up to k vertices that satisfy the meta connections (see Section 4). While processing queries, the GLogue manager will buffer the incoming patterns (patterns only without predicates) from the users, and will launch a procedure to update the GLogue from the buffered patterns periodically.

Algorithm 1: The Plan Optimizer.

```
1  Function PlanOptimizer (GLogue, PatternDesc)
2      Construct a pattern $p$ from the PatternDesc;
3      Let $QSet$ organize all induced subgraphs of $p$ by level;
4      Initialize a PlanMap to record $\{p : (plan, cost)\}$ with
         patterns in level 1 and 2 pre-computed;
5      for $3 \le level \le |V_p|$ do
6          for $p \in QSet[level]$ do
7              searchPlan $(p, \text{PlanMap}, \text{GLogue})$;
8      return PlanMap.get$(p)$;
9  Function searchPlan (p, PlanMap, GLogue)
10     Initialize Plan$(p)$ and Cost(Plan$(p)$) $\leftarrow \infty$;
11     for edge $= (p_{s_1}, p) \in$ GLogue.getEdges$(p)$ do
12         $(plan1, cost1) \leftarrow$ PlanMap.get$(p_{s_1})$;
13         if edge is a vertex extension then
14             Compute a new $plan'$ by merging $plan1$ and
                 Expand$(p_{s_1} \to p)$;
15         else if edge$\{(p_{s_2}, \mathscr{F}(p_{s_2}))\}$ is binary join then
16             $(plan2, cost2) \leftarrow$ PlanMap.get$(p_{s_2})$;
17             Compute a new $plan'$ by merging $plan1$, $plan2$
                 and Join$(\{p_{s_1}, p_{s_2}\} \to p)$;
18         Compute a new $cost'$ of $plan'$ by Equation 1;
19         if $cost' <$ Cost(Plan$(p)$) then
20             Update Plan$(p)$ as $plan'$ and the cost as $cost'$;
21     PlanMap.insert$(p, (\text{Plan}(p), \text{Cost}(\text{Plan}(p))))$;
```

5.4 Plan optimizer

Another benefit of the graph-based GLogue is that the searching of an optimal plan can be reduced to a variant of shortest path problem: the optimal plan of p is a shortest "path" that has the smallest cost regarding Equation 1, from the base pattern (a single vertex) to p. An example is highlighted in Figure 5. We first assume that the queried pattern and all its sub-patterns are present in the GLogue. The process is shown in Algorithm 1.

The optimizer first builds the pattern from the `PatternDesc` that is compiled from a Gremlin query (line 2), and then organizes all *induced* subgraphs of the queried pattern by levels (line 3). Note that the use of induced subgraphs is key to ensuring worst-case optimality of the computed plan [29]. The `searchPlan` function is then launched for each pattern (line 7). We now consider processing a pattern p in the `searchPlan` function. Before searching for the plan for p, the optimal plans for all its subgraphs in the lower level must have already been computed in the `PlanMap` (line 12,16). We use the graph interface of `getEdges` in line 11 to obtain all sub-patterns in the lower level that connect to the current pattern p. Depending on whether the edge stands for a vertex expansion or binary join, the new plan will be accordingly computed in line 14 and 17. As long as the new cost (line 18) is smaller than a previous value, the plan and its cost will be updated. The optimal plan in the GLogue can be cached to avoid re-computation. For example, in Figure 5, the cached optimal plan for computing p_9 is highlighted in red, which is the worst-case optimal plan in Figure 4(c).

Handling Pattern Miss. We discuss how to process the queried pattern p when it has not yet been recorded in the GLogue. First of all, given two *induced* subgraphs p_1, p_2 of p with $E_p = E_{p_1} \cup E_{p_2}$, by assuming the independent presence of p_1 and p_2 in the graph, we can compute the frequency of p as follows

$$\mathscr{F}(p) = \text{Avg}_{p_1,p_2} \frac{\mathscr{F}(p_1) \times \mathscr{F}(p_2)}{\mathscr{F}(p_1 \cap p_2)}, \tag{5}$$

where $p_1 \cup p_2$ denotes a pattern formed by the common parts of p_1 and p_2. Equation 5 can be recursively called in case p_1 and p_2 are not present. Then in line 11, instead of calling getEdges of the GLogue, we simply enumerate all p_s that can potentially expand to p, either via vertex expansion or binary join. The remaining process naturally follows Algorithm 1. In Figure 5, p_{11} is a pattern missing from the GLogue, which can either be expanded from p_{10}, or joined from p_7 and p_8. Thus, p_7, p_8 and p_{10} can all be p_s in line 11.

5.5 Dataflow Embedding

Given the optimal execution plan $\mathsf{Plan}(p)$, one last step is to embed the execution plan into a `DataflowDesc`. Following the operations of $\mathsf{Plan}(p)$, there must be some base patterns (single vertex) that are not target patterns in any operation. We encode these base patterns as `GetV` and then embed them into `Source` operators. For $\mathsf{Join}(\{p_{s_1}, p_{s_2}\} \rightarrow p_t)$, a `Join` operator is installed, which are connected by the operators that computes p_{s_1} and p_{s_2}, and the keys of the `Join` operator are set to the vertex tags of $V_{p_{s_1}} \cap V_{p_{s_2}}$.

There are two cases for hanlding a vertex expansion $\mathsf{Expand}(p_s \rightarrow p_t)$. If p_t has only one more edge than p_s, the vertex expansion will be transformed into a pair of `FlatMap(GetE)` and `Map(GetV)`, which has been discussed in Section 4.2. Otherwise, suppose p_t has $k > 1$ more edges than p_s. The execution will be decomposed into three phases, namely *attachment*, *intersection* and *unwrapping*, as shown in Figure 4(c). In the attachment phase, a `Map(GetE)` operator is installed, which tells the runtime to attach the adjacent edges of the given vertex as a *set*. The intersection phase handles $k-1$ consecutive intersection operations, while each intersection is achieved by a `Map(GetE)` that instructs computing the common edges between the existing set and the current adjacent edges. The last unwrapping phase uses a `FlatMap(GetV)` to unwrap the neighbors into discrete elements.

```
impl MapFunction for GetE { vtag,tag,label,dir } {
  fn map(&self, mut datum: GRecord) -> GRecord {
    let v = datum.get(self.vtag)?;
    // edge tag already present, do intersection
    if let Some(set) = dataum.get_mut(self.tag) {
        set.intersect(to_set(
          G.get_edges(  // G is a graph handle
            v.get_id(), self.label, self.dir))
        );
    } else { // not present, do attachment
        dataum.insert(self.tag, to_set(
          G.get_edges(
            v.get_id(), self.label, self.dir));
        );
    }
    return datum;
  }
}
```

Figure 6: Code generation of `Map(GetE)`.

Example 5.2. In Figure 4(c), let's consider a mapping f that matches (v_1, v_2) before entering the process of vertex expansion. The attachment phase first maps f into $(f \mid set := \mathsf{Nbr}(f(v_1)))$ by directly attaching the adjacent edges (we reuse the notation of neighbors) as a set. In the phase of intersection, the set is updated by intersecting with the current neighbors of $f(v_2)$, as $(f \mid set := set \cap \mathsf{Nbr}(f(v_2)))$. Finally, the *set* is unwrapped into discrete vertices to match v_3.

6 System Implementation

We have implemented the frontend components including pattern parser, plan optimizer and GLogue manager in Java, to easily connect with Tinkerpop's Java runtime. The backend components are implemented in Rust to be compatible with the underlying dataflow engine, GAIA [38]. The GAIA engine runs n executors in the cluster, and each executor further forks working threads for parallel processing. The frontend and backend components of the system are bridged via the RPC services.

We implement `GRecord` to record a mapping of a pattern, which is essentially a `Map` with the key as pattern's tag (Section 4), and the value that is an `Object` to either encode a vertex, an edge, or a set of vertices/edges. Consequently, a collection of `GRecords` serve as the input and output data of all operators of the GAIA engine. Initially, the executors will load corresponding vertices as `GRecords` according to the graph partition. In the following computation, we can use the `Repartition` primitive of GAIA to reshuffle the data as needed. For example, a vertex v will be loaded by the executor numbered as "v.Id % #partitions". Moreover, in order to get adjacent edges from the matched vertices tagged as v, we can `Repartition` the `GRecords` according to the field of v.

6.1 The Dataflow Plugin

The dataflow plugin is key to making the `DataflowDesc` executable (Section 4) on the dataflow engine, which is a native library consisted of the generated code for operators in `DataflowDesc` and a job assembler for assembling the GAIA job.

We perform code generation for all possible operators in `DataflowDesc`, and co-compile the generated code with GAIA. In Figure 6, we show the generated code of `Map(GetE)` that fulfils the phases of attachment and intersection for vertex expansion (Section 5.4). Note that in the `Map` operator for attachment in Figure 4(c), the system assigns a "_t" (as temporary) tag in the `GetE`, which will cause the adjacent edges maintaining as a set in the "_t" field of a `GRecord`. In the `Map` operator for intersection, as the "_t" field must present, it does an intersection between the existing set and the current adjacent edges.

The job assembler, after receiving the `DataflowDesc`, attempts to assemble the GAIA job. Basically, it will install the corresponding GAIA operator for each operator in the `DataflowDesc`. For example, a `Map(GetE)` will be installed as a `Map` operator with the generated code in Figure 6. The job assembler is also responsible for installing the `Repartition` primitive in case that data shuffling is needed.

7 Evaluation

7.1 Setup

Datasets. We base the evaluation on Linked Data Benchmark Council (LDBC)'s social network benchmark [30], which is the only publicly available resource that provides the scale we target in this work. As shown in Table 1, 5 datasets are generated using LDBC data generator, where G_{sf} denotes the graph generated with scale factor sf. The largest graph G_{1000} consumes around 2TB on disk and roughly 6TB aggregated memory in the cluster. The default sparsifying rate $\gamma = 100\frac{|E_{G^*}|}{|E_G|}\%$ is given for each graph. With γ, we set the memory constraint $M = \gamma|E_G|$ (Section 5.2) for graph sparsification. We by default construct the GLogue with level = 3 using the corresponding sparsified graph.

Table 1: The LDBC datasets.

Graph	$\|V\|$	$\|E\|$	**Size**	γ
G_1	3M	17M	1.5GB	100%
G_{30}	89M	541M	40GB	1%
G_{100}	283M	1,754M	156GB	0.1%
G_{300}	817M	5,269M	597GB	0.1%
G_{1000}	2,687M	17,789M	1,960GB	0.03%

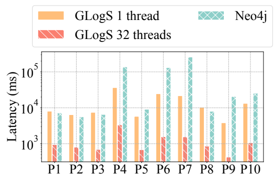
Queries. On the basis of the business intelligence (BI) workloads from LDBC benchmark, we've manually constructed 10 queries, denoted as p_1 to p_{10}, for evaluation. Details of the construction of these queries and their execution plans are in Section A.2. These queries have sufficient variance, ranging from simple triangle patterns to complex patterns like p_9 that contains 7 vertices and 9 edges. We will specify predicates corresponding to the BI workloads for p_4 to p_{10} on G_{100}, G_{300} and G_{1000}, to prevent the tests from running unnecessarily long. The LDBC benchmark driver has been modified to run each queries 5 times from a set of randomly selected parameters. Average query latency is reported.

Systems. We compare GLogS with Neo4j [33] and TigerGraph [18], two potential systems for iGPM (Section 2). We directly use the execution plans of Neo4j. By default, GLogS runs the optimal execution plan of a query p, which are derived from the GLogue (on each sparsified graph) specifically constructed from p. As TigerGraph does not have an optimizer, on the one hand, we used the queries generated by its graph studio [47], on the other hand, we manually wrote the queries according to the optimal plans of GLogS. Note that we did not include the results of our base system, GAIA [38], as it could not terminate in reasonable time in most of our large-scale tests. On the one hand, GAIA has optimized graph workloads by utilizing a breadth-first/depth-first hybrid scheduling and memory-bounded execution model, enabling it to handle considerable amounts of data without overflowing the memory. In fact, our GLogS has benefited from GAIA in handling large-scale data (Section 7.3). On the other hand, GAIA lacks proficiency in executing GPM queries efficiently, primarily due to the fact that GAIA must comply with Gremlin's imperative traversal which conforms to a sub-optimal EdgeJoin execution plan [29] that sequentially joins edges.

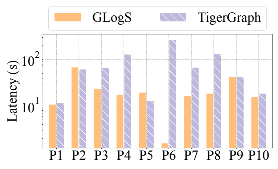
We use the default system configurations of Neo4j and TigerGraph. For GLogS, we have measured the differences in the operations of vertex expansion and binary join, as well as the communication and computation cost, which allows us to set $\alpha_j = 60$ and $\alpha_{ve} = 1$ in Equations 2 and 3, respectively. We deploy a cluster for the evaluation that contains one frontend server and up to 16 backend servers. Each server configures two 24-core Intel(R) Xeon(R) Platinum 8163 CPUs at 2.50GHz and a 512GB RAM. The servers are connected to an EDR 25 Gbps InfiniBand network, which can scale deterministically and achieve full bisection bandwidth. If not mentioned, we will use 32 threads on each server for GLogS (some threads are reserved for communication and system calls) as suggested by GAIA authors [38]. TigerGraph will use all threads as recommended.

7.2 Compare with Alternative Systems

We first compare GLogS with Neo4j on a single machine, using the smallest graph G_1 to allow Neo4j processing all queries in a reasonable time. The query latencies of both systems are shown in Figure 7a. GLogS with a single thread still performs better than Neo4j for most queries, with 4.4× speedup. After using 32 threads, GLogS out-

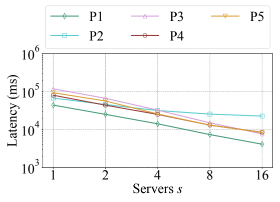


(a) GLogS v.s. Neo4j on G_1

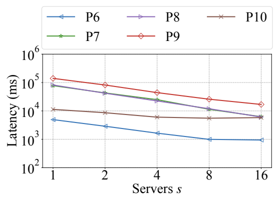


(b) GLogS v.s. TigerGraph on G_{300}

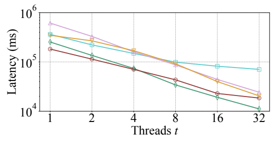
Figure 7: Compare with alternative systems.



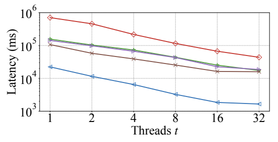
(a) Scale-Out: Group1



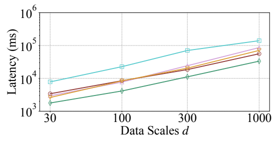
(b) Scale-Out: Group2



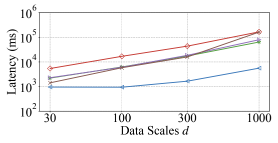
(c) Scale-Up: Group1



(d) Scale-Up: Group2



(e) Data-Scale: Group1



(f) Data-Scale: Group2

Figure 8: Scalability Experiments

performs Neo4j by an average of 51× for all queries.

The tests against TigerGraph are performed on G_{300} using 16 machines. While benefiting from the optimal execution plans of GLogS, TigerGraph still performs 59% slower on average. Note that the results obtained from TigerGraph's graph studio are not reported as they exceed the time limit (one hour) in all tests. Additionally, TigerGraph requires the installation of queries via native-code compilation before they can be run, with installation times ranging from 1 to 3 minutes. In contrast, GLogS does not have this overhead because of the design of dataflow plugin (see Section 6.1). The time required to compile and optimize all queries in GLogS is less than 1 millisecond, which is insignificant compared to the query execution time. This demonstrates GLogS's advantage in usability for iGPM.

7.3 Scalability

It's important to test the scalability on GPM workload given its nature of irregularity [14, 29, 54]. The results are in Figure 8, where the queries are split into two groups based on their latency for clear illustration.

Scale-Out. We vary the server number as 1, 2, 4, 8, 16 and run all queries on G_{100}. Note that G_{100} is the largest graph that can reside in the main memory of a single server. The results are reported in Figure 8a and Figure 8b. Most queries scale well, with up to 15× (average 6×) performance gain from one machine to 16.

Scale-Up We use 16 servers and vary the number of working threads on each server from 1 to 32. The results on G_{300} are shown in Figure 8c and Figure 8d. We see an improvement in runtime of up to 23× (average of 10×) when increasing the number of threads from 1 to 32. A common trend in both scale-out and scale-up tests is that some queries, such as p_2 and p_{10}, scale less significantly when using more working threads. This phenomenon is not unique to our system [14, 38] and is mostly due to the sensitivity of GPM workloads to data skew [14]. It's a future work to further address this issue.

Data-Scale Using 16 servers, we run all queries on the graphs of G_{30}, G_{100}, G_{300} and G_{1000}. The results are reported in Figure 8e and Figure 8f. As the graphs become larger, most queries demonstrate an almost linear trend in performance degradation, except for p_6 and p_{10}. As for p_6, its execution time only tripled from G_{30} to G_{1000}, because it is a short-running query that visits a small part of the graph. However, for p_{10}, its performance degrades by 100× from G_{30} to G_{1000}. This is likely because the execution plan for p_{10} is the only plan that involves a join operator, which maintains a hashmap for the "build" component of the join. When processing a large volume of data, a hashmap lookup can become slower because many entries may have been mapped to the same bucket. Despite this, the plan with the join operator still performs much better than the one without it.

In summary, GLogS exhibits excellent scalability in the test, which we attribute to both the well-designed optimizer and GAIA's graph-specific optimizations.

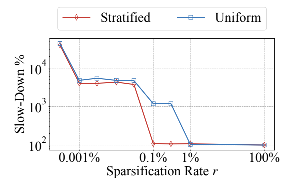
7.4 Plan Optimization

In this experiment, we will study the impact of high-order statistics and graph sparsification on plan optimization for all queries. While running a query, we obtain the time t using the optimal execution plan, and the time t' using the computed execution plan in a certain context. We report the slow-down rate as $100\frac{t'}{t}\%$. For convenience, we run all tests in a single server.

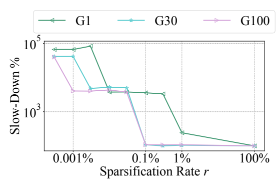
Table 2: The effectiveness of high-order statistics.

	level=2	level=3	level=4
Slow-down (%)	966	245	243
Generation Time(s)	6	55	1664
Memory Usage(GB)	2	3	105
# Patterns	34	248	4164

High-Order v.s. Low-Order. To study the effectiveness of high-order statistics, we construct the GLogue of level 2, 3 and 4 for G_1, and try to evaluate the average slow-down rate of all queries while using the execution plans



(a) Uniform v.s. Stratified



(b) Vary graphs.

Figure 9: Impact of sparsification on plan optimization.

computed from these GLogue. Here, we use the unsparsified G_1 to rule out potential impact from sparsification. The results are shown in Table 2. When increasing the level of GLogue from 2 (low-order statistics only) to 3, the performance of queries improves by $4\times$ on average. The shows that high-order statistics can contribute to deriving better execution plans for GPM queries. Moreover, the performance of queries remains almost unchanged when we increase GLogue's level from 3 to 4, but the time and memory consumption for constructing the 4-level GLogue increase significantly. Given that the GLogue will be further updated from frequently queried patterns, we suggest that the initial construction of 3-level GLogue is sufficient.

Uniform v.s. Stratified. This test verifies the advantage of the proposed stratified sparsification over the uniform alternative. As shown in Figure 9a, we apply both methods on G_{100}, and report the average slow-down rate for all queries using the execution plans computed from the sparsified graphs of various sparsifying rate ranging from 0.001% to 100%. Stratified sparsification performs much better than the uniform alternative, as it has resulted in a better execution plan at a lower sparsifying rate and, at the same sparsifying rate, it has achieved a lower slow-down. With statified sparsification, the graph can be sparsified $10\times$ more edges, on which the optimizer can still derive the optimal execution plans.

Vary Graphs. To verify whether we can use larger sparsifying rate on larger graph, we sparsify three graphs G_1, G_{30} and G_{100} (G_{300} and G_{1000} are too large to process in a single server) using different rates, and report the average slow-down of all queries from the resulting plans in Figure 9b. Clearly, larger graphs can be sparsified at a lower rate, while still rendering good execution plans. For example, the performance of queries on G_{100} and G_{30} only notably downgrades when $\gamma < 0.1\%$. In comparison, the downgrading point of G_1 are $\gamma < 1\%$. Furthermore, while sparsified to 0.001%, the resulting plans from G_{100} slow down by roughly $50\times$, but those from G_{30} and G_1 downgrade by over $500\times$.

8 Related Work

GPM Algorithms. Ullmann proposed the first backtracking algorithm [50] for GPM, based on which many optimizations have been proposed, such as tree indexing [41], symmetry breaking [25] and compression [13]. As it's hard to parallelize the backtracking algorithm, join-based algorithms, such as binary-join algorithms [28, 29, 43], have been developed in the distributed context. Aware that binary-join algorithms cannot guarantee worst-case optimality, [4] implemented the worst-case-optimal join algorithm [35] for solving GPM. A hybrid mechanism [1, 32, 54] has been further explored to combine the advantage of binary join and worst-case optimal join. The above algorithms all rely on low-order statistics to devise execution plans. In order to improve cost estimation, [32] proposed to leverage the high-order statistics of the graph to compute execution plans for GPM. These algorithmic approaches are lacking essential system components needed to solve iGPM.

Query Languages and Graph Databases. GPM lies at the core of the query languages of Gremlin [39], Cypher [22], G-Core [5], PGQL [51] and GSQL [18]. These languages have been widely adopted in graph databases and systems. Tinkerpop [7] uses the Gremlin language to express graph traversal and pattern matching. Neo4j [33] is one of the most popular graph databases that uses Cypher as the query language. Gremlin-enabled JanusGraph [27], Orient DB [36] and Neptune [9] store graph data in distribution, but they adopt a sequential computing engine and can still suffer from scalability issue [38]. Targeting large scale, GAIA [38] has been developed to compile Gremlin traversal queries into a distributed dataflow program. However, the imperative Gremlin traversal cannot guarantee worst-case optimality, and it requires users to manually tune the execution. TigerGraph [18] is a distributed graph database, that uses the GSQL query language. However, the lack of an automatic optimizer greatly limits its usability for iGPM.

9 Conclusion

We've presented the GLogS system in this paper to solve the iGPM, meeting the requirements of performance, usability and scalability. GLogS allows users to interactively submit declarative GPM queries. With the worst-case optimality and high-order statistics, we've implemented an optimizer in GLogS that can automatically derive optimal execution plans for arbitrary GPM queries. Furthermore, on top of an existing distributed dataflow engine, GLogS is capable of being deployed in a large cluster to handle large-scale real-life graphs.

Acknowledgments

We sincerely thank our shepherd Călin Iorgulescu and the anonymous reviewers for their insightful comments. This work was supported by Alibaba Innovative Research (AIR) Program.

References

[1] Aberger, C. R., Lamb, A., Tu, S., Nötzli, A., Olukotun, K., and Ré, C. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS) 42*, 4 (2017), 1–44.

[2] AgensGraph. `https://bitnine.net/`. [Online; accessed 20-October-2022].

[3] Alon, N., Dao, P., Hajirasouliha, I., Hormozdiari, F., and Sahinalp, S. C. Biomolecular network motif counting and discovery by color coding. *Bioinformatics 24*, 13 (2008), i241–i249.

[4] Ammar, K., McSherry, F., Salihoglu, S., and Joglekar, M. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proc. VLDB Endow. 11*, 6 (oct 2018), 691–704.

[5] Angles, R., Arenas, M., Barcelo, P., Boncz, P., Fletcher, G., Gutierrez, C., Lindaaker, T., Paradies, M., Plantikow, S., Sequeda, J., van Rest, O., and Voigt, H. G-core: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data* (New York, NY, USA, 2018), SIGMOD '18, Association for Computing Machinery, p. 1421–1432.

[6] Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., and Vrgoč, D. Foundations of modern query languages for graph databases. *ACM Comput. Surv. 50*, 5 (sep 2017).

[7] Apache TinkerPop. `http://tinkerpop.apache.org/`. [Online; accessed 20-October-2022].

[8] ArangoDB. `https://www.arangodb.com/`. [Online; accessed 20-October-2022].

[9] AWS Neptune. `https://aws.amazon.com/neptune/`. [Online; accessed 20-October-2022].

[10] Babai, L., and Luks, E. M. Canonical labeling of graphs. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1983), STOC '83, Association for Computing Machinery, p. 171–183.

[11] Batson, J., Spielman, D. A., Srivastava, N., and Teng, S.-H. Spectral sparsification of graphs: theory and algorithms. *Communications of the ACM 56*, 8 (2013), 87–94.

[12] Benson, A. R., Gleich, D. F., and Leskovec, J. Higher-order organization of complex networks. *Science 353* (2016), 163–166.

[13] Bi, F., Chang, L., Lin, X., Qin, L., and Zhang, W. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 1199–1214.

[14] Chen, X., et al. Efficient and scalable graph pattern mining on {GPUs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (2022), pp. 857–877.

[15] Chen, Z., Chen, L., Villar, S., and Bruna, J. Can graph neural networks count substructures? *Advances in neural information processing systems 33* (2020), 10383–10395.

[16] Cypher Query Language. `https://neo4j.com/developer/cypher/`. [Online; accessed 20-October-2022].

[17] Deshpande, M., Kuramochi, M., Wale, N., and Karypis, G. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Transactions on Knowledge and Data Engineering 33*, 8 (2005), 1036–1050.

[18] Deutsch, A., Xu, Y., Wu, M., and Lee, V. Tigergraph: A native mpp graph database. *arXiv preprint arXiv:1901.08248* (2019).

[19] Esfahani, M. S., and Dougherty, E. R. Effect of separate sampling on classification accuracy. *Bioinformatics 30*, 2 (2014), 242–250.

[20] Flake, G. W., Lawrence, S., Giles, C. L., and Coetzee, F. M. Self-organization and identification of web communities. *Computer 35*, 3 (2002), 66–70.

[21] for Language Recognition, A. T. `https://www.antlr.org/`. [Online; accessed 20-October-2022].

[22] Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., and Taylor, A. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 1433–1445.

[23] Gaüzère, B., Brun, L., and Villemin, D. Graph kernels in chemoinformatics. In *Quantitative Graph TheoryMathematical Foundations and Applications*, M. Dehmer and F. Emmert-Streib, Eds. CRC Press, 2015, pp. 425–470.

[24] Grochow, J. A., and Kellis, M. Network motif discovery using subgraph enumeration and symmetry-breaking. In *Research in Computational Molecular Biology* (Berlin, Heidelberg, 2007), T. Speed and H. Huang, Eds., Springer Berlin Heidelberg, pp. 92–106.

[25] Han, W.-S., Lee, J., and Lee, J.-H. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, Association for Computing Machinery, p. 337–348.

[26] Hu, Y., Ji, S., Jin, Y., Feng, L., Stanley, H. E., and Havlin, S. Local structure can identify and quantify influential global spreaders in large scale social networks. *Proceedings of the National Academy of Sciences 115*, 29 (2018), 7468–7472.

[27] JanusGrap. `https://janusgraph.org/`. [Online; accessed 20-October-2022].

[28] Lai, L., Qin, L., Lin, X., and Chang, L. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment 8*, 10 (2015), 974–985.

[29] Lai, L., Qing, Z., Yang, Z., Jin, X., Lai, Z., Wang, R., Hao, K., Lin, X., Qin, L., Zhang, W., et al. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment 12*, 10 (2019), 1099–1112.

[30] LDBC Social Network Benchmark. `https://ldbcouncil.org/benchmarks/snb/`. [Online; accessed 20-October-2022].

[31] Loukas, A., and Vandergheynst, P. Spectrally approximating large graphs with smaller graphs. In *International Conference on Machine Learning* (2018), PMLR, pp. 3237–3246.

[32] Mhedhbi, A., and Salihoglu, S. Optimizing subgraph queries by combining binary and worst-case optimal joins. *arXiv preprint arXiv:1903.02076* (2019).

[33] Miller, J. J. Graph database applications and concepts with neo4j. In *Proceedings of the southern association for information systems conference, Atlanta, GA, USA* (2013), vol. 2324.

[34] Neo4j Execution Plan. `https://neo4j.com/docs/cypher-manual/current/execution-plans/`. [Online; accessed 20-October-2022].

[35] Ngo, H. Q., Porat, E., Ré, C., and Rudra, A. Worst-case optimal join algorithms. *Journal of the ACM (JACM) 65*, 3 (2018), 1–40.

[36] Orient DB. `hhttps://orientdb.org/`. [Online; accessed 20-October-2022].

[37] Pržulj, N., Corneil, D. G., and Jurisica, I. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics 22*, 8 (2006), 974–980.

[38] Qian, Z., Min, C., Lai, L., Fang, Y., Li, G., Yao, Y., Lyu, B., Zhou, X., Chen, Z., and Zhou, J. GAIA: A system for interactive analysis on distributed graphs using a High-Level language. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (Apr. 2021), USENIX Association, pp. 321–335.

[39] Rodriguez, M. A. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages* (2015), pp. 1–10.

[40] Sanei-Mehri, S.-V., Sariyuce, A. E., and Tirthapura, S. Butterfly counting in bipartite networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2018), pp. 2150–2159.

[41] Shang, H., Zhang, Y., Lin, X., and Yu, J. X. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow. 1*, 1 (aug 2008), 364–375.

[42] Spielman, D. A., and Teng, S.-H. Spectral sparsification of graphs. *SIAM Journal on Computing 40*, 4 (2011), 981–1025.

[43] Steinbrunn, M., Moerkotte, G., and Kemper, A. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal 6*, 3 (1997), 191–208.

[44] Sun, S., Sun, X., Che, Y., Luo, Q., and He, B. Rapidmatch: a holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment 14*, 2 (2020), 176–188.

[45] Teixeira, C. H., Fonseca, A. J., Serafini, M., Siganos, G., Zaki, M. J., and Aboulnaga, A. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 425–440.

[46] The Open Source Page of GLogS. `https://github.com/MeloYang05/GLogS-Artifact`. [Online; accessed 7-June-2023].

[47] TigerGraph GraphStudio. `https://docs.tigergraph.com/gui/current/graphstudio/build-graph-patterns/visual-query-builder-overview`. [Online; accessed 20-October-2022].

[48] Truth Behind Neo4j's "Trillion" Relationship Graph. `https://www.tigergraph.co.jp/blog/truth-behind-neo4js-trillion-relationship-graph/`. [Online; accessed 20-October-2022].

[49] Tsourakakis, C. E., Drineas, P., Michelakis, E., Koutis, I., and Faloutsos, C. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Network Analysis and Mining 1*, 2 (2011), 75–81.

[50] Ullmann, J. R. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM) 23*, 1 (1976), 31–42.

[51] van Rest, O., Hong, S., Kim, J., Meng, X., and Chafi, H. Pgql: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems* (New York, NY, USA, 2016), GRADES '16, Association for Computing Machinery.

[52] Verma, S., Leslie, L. M., Shin, Y., and Gupta, I. An experimental comparison of partitioning strategies in distributed graph processing. *Proc. VLDB Endow. 10*, 5 (jan 2017), 493–504.

[53] Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *International Conference on Learning Representations* (2019).

[54] Yang, Z., Lai, L., Lin, X., Hao, K., and Zhang, W. Huge: An efficient and scalable subgraph enumeration system. In *Proceedings of the 2021 International Conference on Management of Data* (2021), pp. 2049–2062.

[55] You, J., Gomes-Selman, J. M., Ying, R., and Leskovec, J. Identity-aware graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2021), vol. 35, pp. 10737–10745.

A Appendix

A.1 Graph Sparsification

In the paper, we adopt the stratified sparsification [19] that treats each type of edges as an independent stratum, and assign each stratum an individual sparsification ratio. For clarity, we use v (or e) and u (or ε) to denote a vertex (or edge) in the pattern and graph, respectively. Moreover, given an edge e (or ε), we denote its label as $L(e)$. Let $\{1,2,\ldots,l\} \subset \mathbb{N}^+$ be the domain of edge labels without loss of generality, and $\{s_1,s_2,\ldots,s_l\}$ be the frequencies of the edges with the given label in the graph G. Then we define the sparsification ratios as $\Omega = \{\rho_1,\rho_2,\ldots,\rho_l\}$, where ρ_i denotes the ratio for the stratum of edges with label i.

For two random variables X, X', we denote $\mathbb{E}[X], \mathrm{Var}[X]$ as the expected value and variance of X, and $\mathrm{Cov}[X,X']$ the covariance of X and X'. Given that we only eliminate edges in the sparsification process, we assume that the vertex set retains after sparsification, namely $V_G = V_{G^*}$.

A.1.1 The Proof of Lemma 6.1

Proof. Let $E_G(u_1, u_2) = 1$ indicate the existence of edge (u_1, u_2) in the graph G and 0 otherwise. We formulate that

$$\mathscr{F}_{G^*}(p) = \underbrace{\sum_{(u_1,u_2,\ldots,u_{|V_p|}) \in V_{G^*}^{|V_p|}}}_{\text{For each possible subgraph,}} \underbrace{\prod_{\substack{e \in E_p,\\ e=(v_j,v_k)}} E_{G^*}(u_j, u_k)}_{\text{verify the existence}}, \tag{6}$$

and obtain

$$\begin{aligned} &\mathbb{E}[\mathscr{F}_{G^*}(p)] \\ &= \sum_{(u_1,u_2,\ldots,u_{|V_p|}) \in V_{G^*}^{|V_p|}} \prod_{\substack{e \in E_p,\\ e=(v_j,v_k)}} \rho_{L(e)} \times E_G(u_j, u_k) \\ &= \prod_{e \in E_p} \rho_{L(e)} \sum_{(u_1,u_2,\ldots,u_{|V_p|}) \in V_{G^*}^{|V_p|}} \prod_{\substack{e \in E_p,\\ e=(v_j,v_k)}} E_G(u_j, u_k), \end{aligned}$$

Since $V_G = V_{G^*}$, we have

$$\mathbb{E}[\mathscr{F}_{G^*}(p)] = \prod_{e \in E_p} \rho_{L(e)} \mathscr{F}_G(p).$$

Thus, the lemma holds.

A.1.2 The Optimization Problem for the Stratified Sparsification

Let M denote a memory constraint to ensure that the sparsified graph can reside in the frontend machine. We first consider a specific pattern p, and later generalize to an arbitrary pattern. We formulate the stratified sparsification as an optimization problem as:

$$\begin{aligned} &\arg\min_{\Omega} \mathrm{Var}[\widetilde{\mathscr{F}(p)}] \\ &\text{s.t.} \sum_{i=1}^{l} s_i \rho_i \le M. \end{aligned} \tag{7}$$

Given an ordered set of vertices $S = (u_1, u_2, \ldots, u_{|V_p|}) \in V_G^{|V_p|}$ from the graph G, we denote $\mathbb{1}(f_G(p)|S)$ to indicate whether there is a subgraph with S in G that can match the pattern p. If $\mathbb{1}(f_G(p)|S) = 1$, in other words, S must be mapping of p in G, we directly use $f_G(p)|S$ to represent the matched subgraph regarding S. Note that the notations will be applied to both the `eMap` and G^* in the following. For example, given a vertex set S (same vertex set in both `eMap` and G^*), if $\mathbb{1}(f_G(p)|S) = 1$ but $\mathbb{1}(f_{G^*}(p)|S) = 0$, we know that some edge in the matched subgraph has been eliminated during sparsification. With the indicator, we have

$$\mathscr{F}_{G^*}(p) = \sum_{S \in V_{\mathtt{eMap}}^{|V_p|}} \mathbb{1}(f_{G^*}(p)|S).$$

Therefore,

$$\begin{aligned} &\mathrm{Var}[\mathscr{F}_{G^*}(p)] \\ &= \sum_{S_1, S_2 \in V_G^{|V_p|}} \mathrm{Cov}\left[\mathbb{1}(f_{G^*}(p)|S_1), \mathbb{1}(f_{G^*}(p)|S_2)\right] \\ &= \sum_{S_1, S_2 \in V_G^{|V_p|}} (\mathbb{E}[\mathbb{1}(f_{G^*}(p)|S_1)\mathbb{1}(f_{G^*}(p)|S_2] \\ &\quad - \mathbb{E}[\mathbb{1}(f_{G^*}(p)|S_1)] \times \mathbb{E}[\mathbb{1}(f_{G^*}(p)|S_2)]. \end{aligned} \tag{8}$$

We observe that the covariance in Equation 8 must be zero in either of the following case.

- S_1 or S_2 cannot form a mapping of p in G, i.e., $\mathbb{1}(f_{\mathtt{eMap}}(p)|S_1) = 0$ or $\mathbb{1}(f_{\mathtt{eMap}}(p)|S_2) = 0$. In this case, the matched subgraph must not exist in G^*, leading to $\mathbb{E}[\mathbb{1}(f_{G^*}(p)|S_1)] = 0$ or $\mathbb{E}[\mathbb{1}(f_{G^*}(p)|S_2)] = 0$.
- $S_1 \cap S_2 = \emptyset$, namely the two sets are disjoint.

Therefore, we only need to study the two sets S_1 and S_2, such that $\mathbb{1}(f_{\mathtt{eMap}}(p)|S_1) = 1$ and $\mathbb{1}(f_{\mathtt{eMap}}(p)|S_2) = 1$, and $S_1 \cap S_2 \neq \emptyset$. In this case, $f_{\mathtt{eMap}}(p)|S_1$ and $f_{\mathtt{eMap}}(p)|S_2$ may share either none, one, or more than one common edges. The trivial case of sharing no edge results in zero variance. For other cases, we empirically studied their occurrences while matching all benchmark queries on G_1. We found that the case of sharing one single edge occurs much more frequently than that of sharing multiple edges. Let the common edge be ε, which must be matched by a pattern edge $e' \in E_p$. According to Equation 8, the covariance becomes

$$\begin{aligned} &\mathrm{Cov}\left[\mathbb{1}(f_{\mathtt{eMap}}(p)|S_1), \mathbb{1}(f_{\mathtt{eMap}}(p)|S_2)\right] \\ &= \left(\prod_{e \in E_p} \rho_{L(e)}\right)^2 * \left(\rho_{L(e')}^{-1} - 1\right). \end{aligned} \tag{9}$$

We then group these pairs by the labels of e', which eliminates e' in Equation 9 and transforms Equation 8 into

$$\begin{aligned} &\mathrm{Var}[\mathscr{F}_{G^*}(p)] \\ &\approx \left(\prod_{e \in E_p} \rho_{L(e)}\right)^2 * \sum_{e \in E_p} \left(\rho_{L(e)}^{-1} - 1\right) \lambda_{\mathtt{eMap}}(p|e), \end{aligned} \tag{10}$$

where $\lambda_{\mathtt{eMap}}(p|e)$ denotes the number of pairs of S_1 and S_2 in `eMap` whose common edge is matched by e in pattern p.

Figure 10 demonstrates a pair of S_1 and S_2 that matches a triangle pattern, as an example. In addition, matched subgraphs $f_{\mathtt{eMap}}(p|S_1)$ and $f_{\mathtt{eMap}}(p|S_2)$ share a common edge in the example. We observe that $f_{\mathtt{eMap}}(p|S_1)$ and $f_{\mathtt{eMap}}(p|S_2)$ together form a new pattern,

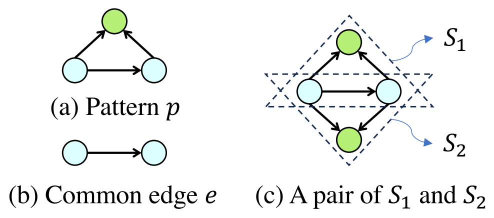


Figure 10: Example of a pair of S_1 and S_2 with shared edge e.

which is a mirror symmetric structure with respect to the common edge. In terms of an edge $e \in E_p$, we denote such a mirror pattern as p_e^+. It's clear that $\lambda_{\texttt{eMap}}(p|e)$ is equal to the frequency of pattern p_e^+ in eMap, namely $\lambda_{\texttt{eMap}}(p|e) = \mathscr{F}_G(p_e^+)$. Combining with the Equation 4 in the paper, we have

$$\lambda_{\texttt{eMap}}(p|e) = \mathscr{F}_G(p_e^+) \approx \frac{\mathscr{F}_G(p) \times \mathscr{F}_G(p)}{\mathscr{F}_G(e)} = \frac{\mathscr{F}_G(p)^2}{s_{L(e)}}, \tag{11}$$

Combining the definition of $\widetilde{\mathscr{F}(p)}$, Equation 10, and Equation 11, we obtain

$$\mathrm{Var}[\widetilde{\mathscr{F}(p)}] \approx \sum_{e \in E_p} \left(\rho_{L(e)}^{-1} - 1\right) \frac{\mathscr{F}_G(p)^2}{s_{L(e)}},$$

Note that $\mathscr{F}_G(p)$ can be treated as a constant value in the optimization problem. Consequently, we can rephrase the optimization problem in Equation 7 as

$$\begin{aligned} &\arg\min_{\Omega} \sum_{e \in E_p} \left(\rho_{L(e)} s_{L(e)}\right)^{-1}, \\ &\text{s.t.} \sum_{i=1}^{l} s_i \rho_i \leq M. \end{aligned} \tag{12}$$

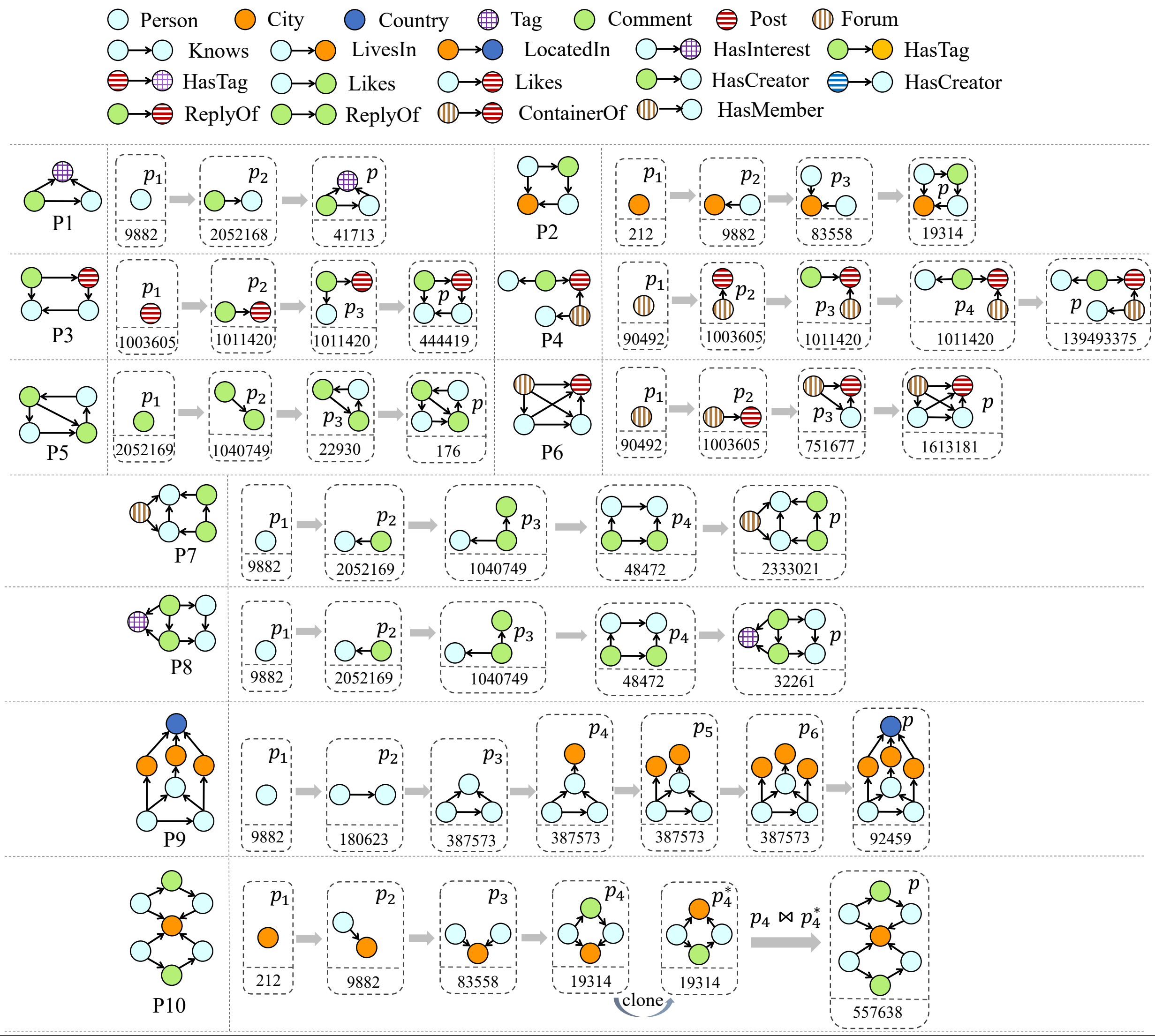
Till now, the optimization problem only considers a specific pattern p. For generalizing to an arbitrary pattern, we construct a pattern p that is formed by all types of edges in the graph, and in Equation 12, we enumerate all labels rather than those in the pattern p, which becomes

$$\begin{aligned} &\arg\min_{\Omega} \sum_{i=1}^{l} (s_i \rho_i)^{-1}, \\ &\text{s.t.} \sum_{i=1}^{l} s_i \rho_i \leq M. \end{aligned}$$

The minimal variance is achieved when $\rho_i = \frac{M}{l} \times \frac{1}{s_i}$. Since the sparsification ratio ρ_i has a upper bound 1, i.e., all edges with label i are preserved, we have $\rho_i = \min(1, \frac{M}{l} \times \frac{1}{s_i})$.

A.2 Queries and Execution Plans

We reported all queries used in the experiments, along with their execution plans generated by the optimizer of GLogS, in Figure 11. In the execution plans, we also marked the corresponding intermediate pattern frequencies in the benchmark graph G_1 for the evaluation of their performance. The table in Figure 11 explains how the queries are constructed. Specifically, their main structures of all queries are extracted from LDBC [30] Business Intelligence (BI) workloads, and then modified to cover more test scenarios. Overall, the queries contains Long-Chain, Triangle, Square, 4-Clique, House that are commonly used for benchmarking GPM queries [29, 32]. Their execution plans cover both Expand and Join operators.



Query	Source	Explanation
p_1	BI-8	p_1 is extracted from BI-8
p_2	BI-5	Wedge Person → Comment → Person is extracted from BI-5. Additionally, a City vertex and two LivesIn edges are added to form a Square
p_3	BI-15	p_3 is extracted from BI-15
p_4	BI-17	p_4 is extracted from BI-17
p_5	BI-17	p_5 is extracted from BI-17
p_6	BI-4, BI-15	Some subgraphs are extracted from BI-4 and BI-15 to form a 4-Clique
p_7	BI-19, BI-4	Its right square is extracted from BI-19. In addition, a Forum hat is extracted from BI-4 and added to the square to form a House
p_8	BI-19, BI-17	Its right square is extracted from BI-19. In addition, a Tag hat is extracted from BI-17 and added to the square to form a House
p_9	BI-11	p_9 is the main structure of BI-11
p_{10}	BI-5	p_{10} consists of two p_2 by joining on the City vertex. This is designed to verify whether GLogue can generate a plan with join

Figure 11: Queries and their executions plans generated by GLogS's Optimizer