A Modular Graph-Native Query Optimization Framework

Bingqing Lyu
bingqing.lbq@alibaba-inc.com
Alibaba Group
Hangzhou, China

Xiaoli Zhou
yihe.zxl@alibaba-inc.com
Alibaba Group
Hangzhou, China

Longbin Lai
longbin.lailb@alibaba-inc.com
Alibaba Group
Hangzhou, China

Yufan Yang
xiaofan.yyf@alibaba-inc.com
Alibaba Group
Hangzhou, China

Yunkai Lou
louyunkai.lyk@alibaba-inc.com
Alibaba Group
Hangzhou, China

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

Ying Zhang
ying.zhang@zjgsu.edu.cn
Zhejiang Gongshang University
Hangzhou, China

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

Abstract

Complex Graph Patterns (CGPs), which combine pattern matching with relational operations, are widely used in real-world applications. Existing systems rely on monolithic architectures for CGPs, which restrict their ability to integrate multiple query languages and lack certain advanced optimization techniques. Therefore, to address these issues, we introduce GOpt, a modular graph-native query optimization framework with the following features: (1) support for queries in multiple query languages, (2) decoupling execution from specific graph systems, and (3) integration of advanced optimization techniques. Specifically, GOpt offers a high-level interface, `GraphIrBuilder`, for converting queries from various graph query languages into a unified intermediate representation (GIR), thereby streamlining the optimization process. It also provides a low-level interface, `PhysicalConverter`, enabling backends integration by converting the optimized plan into a backend-compatible execution plan. Moreover, GOpt employs a graph-native optimizer that encompasses extensive heuristic rules, an automatic type inference approach, and cost-based optimization techniques tailored for CGPs. Comprehensive experiments show that integrating GOpt significantly boosts performance, with Neo4j achieving an average speedup of 9.2× (up to 48.6×), and GraphScope achieving an average speedup of 33.4× (up to 78.7×), on real-world datasets.

CCS Concepts

• **Information systems** → **Query optimization**.

Keywords

Graph Query Optimization, Graph Database management

SIGMOD-Companion '25, June 22–27, 2025, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3724425

ACM Reference Format:
Bingqing Lyu, Xiaoli Zhou, Longbin Lai, Yufan Yang, Yunkai Lou, Wenyuan Yu, Ying Zhang, and Jingren Zhou. 2025. A Modular Graph-Native Query Optimization Framework. 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/10.1145/3722212.3724425

1 Introduction

Graph databases [5, 6, 11, 21, 38] have emerged as powerful tools for analyzing complex relationships across various domains. At the core of these databases lies graph pattern matching, a fundamental and extensively researched problem [15, 16, 33, 40, 41, 46, 54, 61, 63]. Graph pattern (abbr. pattern henceforth) matching aims to identify all subgraphs in a data graph that match a given query pattern and is widely used in numerous areas [25, 30, 32, 36, 50, 64].

Recently, there has been growing interest in complex graph patterns (CGPs) due to their utility in real-world applications [4, 18, 19, 28, 53, 62]. CGPs extend patterns by incorporating relational operations such as projection, selection, and aggregation, enabling more expressive and versatile query capabilities. To efficiently express and execute CGPs, existing graph databases often adopt a *monolithic* architecture design, where the entire process is tightly integrated from query parsing to execution. These systems handle specific query languages, starting with a parser that translates the query into an internal representation. A query optimizer then generates a physical execution plan, which is subsequently executed on an underlying engine. Two representative examples of graph systems are Neo4j [6] and GraphScope [29], illustrated in Fig. 1.

Neo4j processes CGPs written in the Cypher query language[31]. It employs a Cypher parser to convert the query into an intermediate structure, uses a CypherPlanner to optimize queries through heuristic rules and cost-based techniques, and executes the optimized plans on its single-machine backend. In contrast, GraphScope [29] is designed for industrial-scale distributed graph processing. It supports CGPs expressed in Gremlin [53], utilizes a Gremlin parser, applies `TraversalStrategy` to register heuristic optimization rules, and distributes the execution across its dataflow-based distributed backend engine [51].

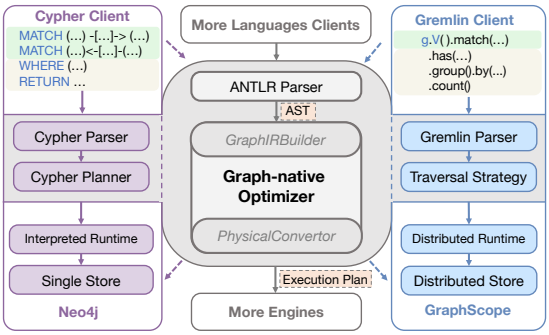


Figure 1: Example of CGPs processing in Neo4j and GraphScope monolithic systems versus GOpt's modular design.

Motivation and Challenges. The monolithic architecture used by existing systems like Neo4j and GraphScope for processing CGPs poses two key limitations.

Limitation 1. In practice, most graph systems support only one query language, but evolving user demands sometimes necessitate multiple language support. For example, at Alibaba, users transitioning from Neo4j to GraphScope for scalable query processing may require support for Cypher queries, despite GraphScope being originally designed for Gremlin. Additionally, the upcoming standardization of the graph query language GQL [4] further drives the need for systems to adapt and expand their capabilities. While a language converter is feasible, the only official converter currently available translates Cypher to Gremlin [26], limiting its applicability in other scenarios, such as adopting GQL in GraphScope. The monolithic architectures make it challenging for existing systems to integrate support for additional query languages.

Limitation 2. Different systems employ distinct optimizations for CGPs, each with strengths but often lack advanced techniques available in others. For example, Neo4j leverages cost-based optimization (CBO) for optimal search order but lacks advanced techniques like worst-case optimal join [48], which GraphScope employs for efficient cyclic pattern matching. Conversely, GraphScope relies on a rule-based optimizer, missing the sophisticated CBO strategies available in Neo4j. Both systems lack support for recent advancements like high-order statistics [41], crucial for improving cardinality estimation and query planning. Monolithic architectures hinder the seamless integration of these optimization techniques.

A potential solution is to adopt a *modular* and *unified* optimization framework with key features:

- **Multiple query language support**: The framework should inherently support parsing and processing multiple query languages, enabling seamless interoperability.
- **Decoupling from existing systems**: By being independent of specific graph systems, it can provide modular interfaces that simplify integration with various platforms.
- **Advanced optimization capabilities**: The framework should integrate advanced optimization techniques from academia and industry into a unified system.

Beyond the efforts of system design, the implementation of such a framework poses immediate challenges.

***C1**: How to support cross-language compatibility?* CGPs can be expressed in diverse query languages such as Cypher and Gremlin, each with distinct syntaxes and semantics, making integration into a unified framework challenging.

***C2**: How to effectively optimize CGPs?* Optimizing CGPs is inherently complex due to their hybrid semantics, combining patterns with relational operations. Existing graph optimization techniques [47, 63] focus on patterns and often overlook relational aspects. While converting patterns into relational equivalents [7, 57, 59] enables the use of relational optimization techniques, this *non-graph-native* approach may miss optimization opportunities specific to graph pattern matching [41, 63]. Another challenge is handling arbitrary type constraints. CGPs often include patterns without explicit type constraints when users have not assigned specific types on certain vertices and edges and can, in theory, match any type in the data graph. In practice, however, these elements are implicitly restricted by the underlying data. Current techniques [41] depend heavily on explicit type constraints, limiting their effectiveness in such cases.

Our Solutions. To address the challenges, We present GOpt, a modular *graph-native* optimization framework for CGPs, as illustrated in Fig. 1. It is designed to support multiple query languages and seamlessly integrate with existing graph systems.

***S1**: Modular System Design.* GOpt enables cross-language compatibility with a unified graph intermediate representation (GIR) for CGPs, combining patterns and relational operations. It offers a high-level interface, `GraphIrBuilder`, that translates the Abstract Syntax Tree (AST) generated by tools like ANTLR [1] from various query languages into a unified GIR, thereby decoupling query processing from specific languages. GOpt also provides a low-level interface, `PhysicalConverter`, enabling graph systems to convert the optimized GIR into backend-compatible execution plans. This flexibility enables integration with various graph systems while maintaining compatibility with multiple query languages. Currently, GOpt supports both Cypher and Gremlin and are integrated into Neo4j and GraphScope. The standardized GQL is not yet supported due to the lack of its official grammar interpretation tool.

***S2**: Graph-native Optimization for CGPs.* Building on the unified GIR, GOpt enables graph-native optimization by addressing the interaction of patterns and relational operations in CGPs. A comprehensive set of heuristic rules has been designed for the rule-based optimization (RBO), which are extensible and pluggable. To handle patterns with arbitrary types, we introduce an automatic type inference algorithm that deduces appropriate type constraints for pattern vertices and edges from graph data. By inferring type constraints, we can fully exploit CBO techniques, such as those in [41], to enhance cardinality estimation using high-order statistics. We also introduce a `PhysicalCostSpec` to allow backends to register cost models tailored for their operator implementations, enhancing the accuracy of cost estimation. Based on estimated cardinalities and costs, we propose a top-down search framework with branch-and-bound strategies to optimize CGPs, determining the optimal execution plan by minimizing the estimated cost. The unified framework allows advanced graph optimization techniques to be seamlessly integrated, benefiting all integrated graph systems.

Concerns naturally arise regarding the unified design of GOpt, particularly in: (1) the theoretical completeness of GIR in expressing diverse query languages with their syntactic and semantic differences; and (2) the compatibility with different data models,

from Neo4j's schema-loose model to GraphScope's schema-strict model [20]. In this paper, we briefly remark on these concerns and refer readers to our open-source project for more details [2].

Contributions and organization. Below, we summarize our contributions and the organization of the paper.
(1) Introduction of GOpt (Sec.4 and Sec. 5): GOpt stands as the first, as far as we know, graph-native optimization framework for industrial-scale graph systems, supporting multiple query languages through a unified intermediate representation and an extensible optimization architecture.
(2) Advanced optimization techniques (Sec.6): GOpt introduces a comprehensive set of heuristic rules, an automatic type inference algorithm, and a novel cardinality estimation method, integrated into a cost-based optimizer with a top-down search framework.
(3) Implementation and experiments (Sec. 7 and Sec. 8): We implemented GOpt as an open-source project [2] atop Apache Calcite [22], supporting Gremlin and Cypher, and integrating with Neo4j and GraphScope. Experiments show significant performance gains, with GOpt achieving an average speedup of 9.2× (up to 48.6×) on Neo4j and 33.4× (up to 78.7×) on GraphScope using real-world datasets. GOpt has also been deployed in real applications at Alibaba, demonstrating its practicality and scalability.

Additionally, Section 2 covers background and related work, Section 3 introduces the notations and definitions used in this paper, and Section 9 concludes the paper.

2 Background and Related Work

Corresponding to the main challenges in implementing a modular and unified optimization framework mentioned above, this section reviews the existing query languages and optimization techniques, then presents the existing databases for optimizing CGPs.

2.1 Graph Query Languages

Numerous graph query languages [4, 6, 18, 28, 53, 62] have been developed for querying property graphs [17]. For instance, Gremlin [53] represents graph traversal as a sequence of steps, each offering functions such as navigating through graph vertices and edges. Cypher [31] crafted for Neo4j [6], allows users to describe graph patterns with an ASCII-art syntax while integrating relational operators similar to those in SQL. The recently published GQL [4] is a new standard developed by ISO committee for querying property graphs, with a consensus effort that integrates previous advancements. While these languages focus on querying property graphs, SPARQL [10] is a W3C recommendation for querying Resource Description Framework (RDF) graphs [9]. Despite the proliferation of graph query languages, there is a notable gap: the lack of a dedicated optimizer to enhance query efficiency in these languages.

2.2 Graph Query Optimizations

Graph pattern matching is fundamental in graph query processing. In sequential graph pattern matching, Ullmann's first backtracking algorithm [61] was a key advancement. This has led to various optimizations, including tree indexing [54], symmetry breaking [33], and compression techniques [23]. Due to the challenges in parallelizing backtracking algorithms, join-based algorithms have been developed for distributed environments. These algorithms often hinge on cost estimation to optimize join order for matching patterns efficiently. For instance, binary-join algorithms [39, 40] estimate costs using random graph models. In contrast, the work in [16] adopts the WcoJoin algorithm [48], ensuring a cost that remains within a worst-case upper bound. Recognizing the limitations of both binary join and WcoJoin methods in consistently delivering optimal performance [40], researchers have turned to hybrid approaches [15, 47, 63] that adaptively select between binary and WcoJoin techniques based on the estimated costs. For more accurate cost estimation, [47] and [41] suggested to leverage high-order statistics. While these studies have significantly improved graph pattern matching, they are not designed to address graph queries in practice that often encompasses relational operations. There is extensive work on RDF graph query optimization [45, 52, 56, 60], which is not detailed here as this paper focuses on property graphs.

Table 1: Limitations of Existing Graph Databases. "Lang.", "Opt.", "H. Stats", and "T. Infer" represent "Language", "Optimization", "High-order Statistics", and "Type Inference".

Database	Lang.	Opt.	WcoJoin	H. Stats.	T. Infer
Neo4j	Cypher	RBO/CBO	×	×	×
GraphScope	Gremlin	RBO	✓	×	×
GLogS	Gremlin	CBO	✓	✓	×
GOpt	Multiple	RBO/CBO	✓	✓	✓

2.3 Databases for Optimizing CGPs

Graph databases [5, 6, 11, 21, 38, 41] enable users to perform complex graph queries using declarative query languages. However, they encounter limitations when it comes to fully supporting CGPs. For example, Neo4j, a leading graph database, is limited by its tight coupling with Cypher [31] for CGPs support. GraphScope [29], a distributed graph computing system coupled with the Gremlin, lacks CBO optimization techniques. GLogS [41] is designed specifically to optimize patterns, ensuring worst-case optimality as well as leveraging high-order statistics (i.e., the small pattern frequencies), but it cannot optimize CGPs due to its inability to handle relational operations or arbitrary types. More limitations are summarized in Table 1. Note that Neo4j does not perform type inference because of its schema-loose design for greater flexibility. In summary, current graph databases face challenges in optimizing CGPs effectively.

A *non-graph-native* approach involves converting CGPs into relational queries and leveraging relational optimizers, as seen in systems like IBM DB2 Graph[59], SQLGraph [57], and others [34, 37, 43]. The primary advantage of this method lies in its ability to build upon the well-studied relational optimization techniques. However, this approach fails to utilize graph-specific optimizations [41, 47, 63] and struggles to handle UnionTypes (i.e., a combination of multiple types, detailed in Section 3) in CGPs, a construct inherently tied to graph semantics and absent in relational models, leading to inefficient query execution. For example, matching the pattern with UnionTypes in Fig. 3(a) may require executing multiple queries—each with specific types for pattern vertices and edges—and then merging the results, leading to inefficiency.

3 Preliminaries

Data graph $G = (V_G, E_G)$ in this paper adheres to the definition of the property graph model [19]. V_G and E_G are the sets of vertices

Table 2: Frequently used notations.

Notation	Definiton
$G(V_G, E_G)$	A data graph with V_G and E_G
$P(V_P, E_P)$	A pattern graph with V_P and E_P
$N_G(v)$, $N_G^E(v)$	Out neighbors and out edges of v in graph G
$\lambda_G(v)$, $\lambda_G(e)$	The type of vertex v and edge e in graph G
$\tau_P(v)$, $\tau_P(e)$	The type constraint of vertex v and edge e in pattern P, can be BasicType, UnionType or AllType
$\mathcal{F}_{P,G}$	The number of mappings of pattern P in graph G

and edges, where $|V_G|$ and $|E_G|$ represent the number of vertices and edges. Given $u, v \in V_G$, $(u, v) \in E_G$ is an edge directed from u to v, and all the out neighbors and out edges of u are denoted as $N_G(v)$ and $N_G^E(v)$, respectively. Each vertex or edge in G is associated with a type, denoted as $\lambda_G(v)$ and $\lambda_G(e)$ respectively. Both vertices and edges can carry properties, which are key-value pairs. Note that if no ambiguity arises, we omit G from the subscript in the notations, and so as the follows. Besides, although our framework supports multi-type vertices as in [31], this paper focuses on vertices with single type for simplicity. Considering two graphs G_1 and G_2, we assert that G_2 is a subgraph of G_1, symbolized as $G_2 \subseteq G_1$, if and only if $V_{G_2} \subseteq V_{G_1}$, and $E_{G_2} \subseteq E_{G_1}$.

A pattern $P = (V_P, E_P)$ is a small *connected* graph. Note that if P is not connected, matching the pattern is equivalent to taking the Cartesian product of the matches for its connected components, which is naturally the problem of CGPs in the following. Each vertex and edge in the pattern graph is associated with a set of types as a type constraint, denoted as $\tau_P(v)$ and $\tau_P(e)$, respectively. Three categories of type constraints are supported: (1) BasicType contains a single type that matches a particular vertex (or edge) type in the data graph. (2) UnionType encompasses multiple types, allowing for a match with data vertices (or edges) of any type in this set. For instance, UnionType `{Post,Comment}` signifies that the matched vertices can be either of type `Post` or `Comment` in data graph. (3) AllType, considered as a special UnionType, indicates that any vertex or edge in data graph satisfies the constraint. Predicates can be specified to vertices and edges in the pattern graph as well, while we mostly focus on the type constraints in this paper.

Finding matches of P in G involves identifying all subgraphs G' in G where P can be mapped to G' via a homomorphism preserving edge relations and type constraints. The mapping function $h : V_P \rightarrow V_{G'}$ ensures that $\forall e = (u, v) \in E_P$, there is a corresponding edge $(h(u), h(v)) \in E_{G'}$. Additionally, the types of vertices and edges in G' must align with the type constraints specified in P: (1) $\forall v \in V_P, \lambda_{G'}(h(v)) \in \tau_P(v)$, and (2) $\forall e = (u, v) \in E_P, \lambda_{G'}((h(u), h(v))) \in \tau_P(e)$. The number of mappings of P in G is called *pattern frequency*, denoted as $\mathcal{F}_{P,G}$, or simply $\mathcal{F}_P$ when G is clear.

Remark 3.1. *The homomorphism semantics, which allows duplicate vertices or edges in matching results, is employed in our framework due to its transformability. Specifically, a plan using homomorphism semantics can be converted to a plan using other commonly adopted semantics, such as the no-repeated-edge semantics (used by Cypher) that excludes duplicate edges, by adding an all-distinct filter at the end of the pattern match. This makes our framework flexible and adaptable to different query language semantics.*

A complex graph pattern, termed as CGP, extends patterns with further relational operations. Optimizing CGPs is challenging due to their hybrid semantics. A straightforward way is to first identify matches of P in G, and then subject the matched subgraphs to the remaining relational operators in CGPs for further analysis, such as projecting the properties of matched vertices and edges and selecting the matched subgraphs that satisfy certain conditions.

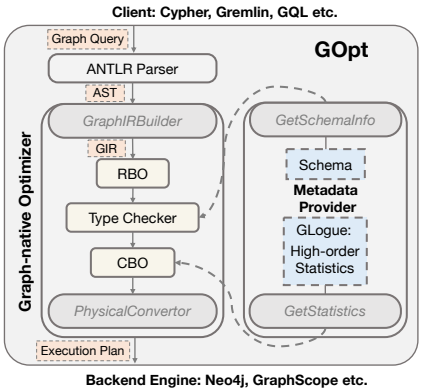


Figure 2: System Architecture Overview

4 System Overview

To address the limitations in existing systems, we propose a graph-native query optimization framework, GOpt, specifically for CGPs. This framework leverages state-of-the-art graph query optimization techniques (with refinement and extension for CGPs), offers a unified approach for supporting multiple query languages, and facilitates seamless integration with various backend engines. The system architecture is shown in Fig. 2, with three principal components of query parser, optimizer, and metadata provider.

Query Parser. GOpt employs the ANTLR parser tool [1] to generate an AST for queries in various graph query languages, such as Gremlin and Cypher. Then it utilizes a `GraphIrBuilder` to convert the AST into a *language-independent* plan based on a unified graph intermediate representation (GIR), detailed in Section 5.1. This GIR decouples GOpt from specific query languages and allows extensive reuse of techniques developed within GOpt.

Optimizer. The graph-native optimizer is the core module in GOpt, employing Rule-based Optimization (RBO), type checker, and Cost-based Optimization (CBO). RBO comprises a comprehensive set of heuristic rules to optimize interactions between patterns and relational operations in CGPs, allowing registration of new rules for further optimization. However, since RBO does not account for data characteristics, it may fail to fully optimize CGPs, particularly in the presence of complex patterns. Therefore, we need to explore additional data-driven optimization techniques for handling CGPs.

The first such technique is a type checker, specifically designed to infer implicit type constraints from the underlying data for patterns with arbitrary types. This technique is crucial for query optimization, enhancing execution efficiency and improving cardinality estimation accuracy in the subsequent CBO phase.

Building on this, we propose a unique CBO approach for further optimization. Our CBO introduces several key innovations for patterns. First, we leverage high-order statistics from GLogS [41] for more accurate estimation of pattern frequencies. Unlike GLogS, however, our approach handles arbitrary type constraints. Second,

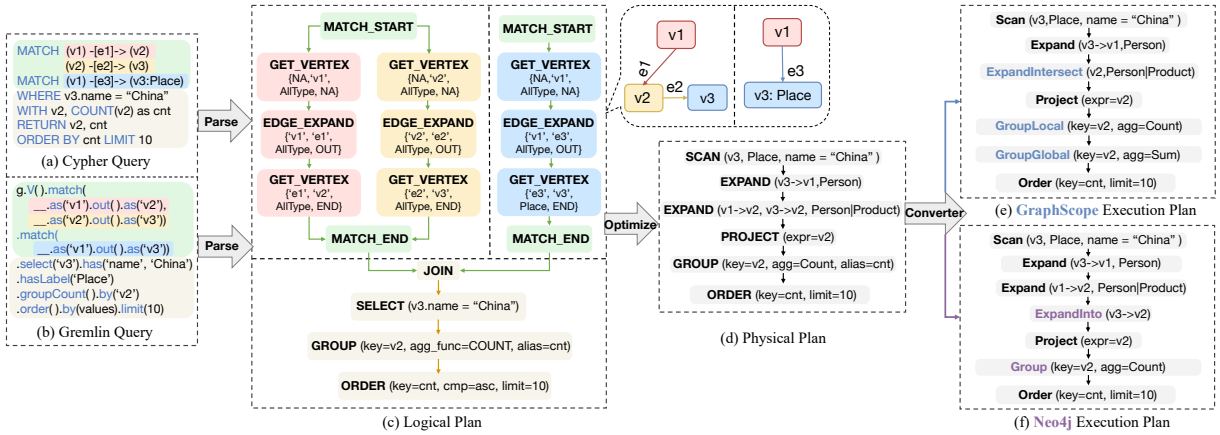


Figure 3: An example of query processing workflow. For simplicity, we draw the pattern graph for `MATCH_PATTERN` (examples are given) in the following. In this paper, GIR operators are in ALL_UPPERCASE format, and engine operators use CamelCase.

to accommodate diverse backend engines, we allow them to register operator cost models tailored to their specific implementations via a `PhysicalCostSpec`. Finally, we introduce a top-down search algorithm with a branch-and-bound strategy to determine the optimal query plan, aiming to minimize the estimated cost. Together, these techniques ensure robust and adaptable query optimization across various systems. We build GOpt based on Apache Calcite [22], a prominent open-source optimization framework for relational databases, to leverage Calcite's relational optimization techniques.

Metadata Provider. The metadata provider has two key components: the first is the graph schema, defining the types of vertices and edges in the data graph for the type checker. For simplicity, we initially assume a schema-strict context with an explicit schema, and more flexible contexts are discussed later in Remark 6.1. The second component, named GLogue following [41], serves as a data statistics provider. It precomputed the frequency of small graph patterns (a.k.a., motifs) with up to k vertices ($k \geq 3$) in the data graph, going beyond the low-order statistics which only count the vertices and edges. These advanced high-order statistics afford the optimizer with more precise cardinality estimation for query patterns, thereby enhancing the effectiveness of the optimization.

Overall Workflow. The query processing workflow is illustrated in Fig. 3. After parsing, the optimization process proceeds in the following order: RBO, type inference, and finally CBO. This sequence is designed to balance optimization effectiveness with computational cost, as both RBO and type inference can influence CBO outcomes. The optimized plan is then converted into a backend-compatible execution plan. Details on query parsing, RBO, type inference, CBO and backend integration are provided in Sec.5, Sec.6.1, Sec.6.2, Sec.6.3, and Sec.7 respectively. This paper focuses specifically on optimizing graph patterns. For instance, in CBO, our primary focus is on the costs associated with these patterns. The handling of relational operators and their costs is further detailed in Remark 7.1.

5 Query Transformation

To allow different query languages to plug into GOpt, we present the unified graph intermediate representation (GIR) for CGPs and describe interfaces to convert queries into this format.

5.1 GIR Abstraction

The GIR abstraction [35] provides a foundation for representing CGPs with both patterns and relational query semantics. It defines a data model $\mathcal{D}$ that presents a schema-like structure for intermediate results in which each data field has a name and a designated datatype, including graph-specific datatypes (e.g., *Vertex*, *Edge*, *Path*) and general datatypes (e.g., *Primitives* and *Collections*). It also defines a set of operators Ω that operate on data tuples from $\mathcal{D}$ and produce new data tuples as results, consisting of graph operators and relational operators. The graph operators are designed to retrieve graph data, including: (1) `EXPAND_EDGE` to expand adjacent edges from vertices; (2) `GET_VERTEX` to retrieve endpoints from edges; (3) `EXPAND_PATH` to support path expansion with a specified hop number l, corresponding to l consecutive edges in the pattern, and users can also define path constraints such as *Arbitrary* (no constraints), *Simple* (*No-repeated-node*), or *Trail* (*No-repeated-edge*) [19]; and (4) `MATCH_PATTERN` as a composite operator that encapsulates the above graph operators between `MATCH_START` and `MATCH_END` to represent a pattern, as illustrated in Fig. 3(c). The relational operators consist of the commonly used operators in RDBMS and can be applied on graph data, e.g., `PROJECT` for projecting vertex properties, `SELECT` for selecting edges based on conditions, `JOIN` for joining sub-paths, etc. We detail all the operators in GIR in [44].

Remark 5.1. *The definition of GIR is inspired by existing work on graph relational algebra [58] and is enhanced with* `MATCH_PATTERN` *to handle complex patterns. Moreover, GIR's development follows an engineer-oriented approach, prioritizing support for commonly used functionalities over theoretical completeness. It continuously evolves to meet new requirements from real applications.*

With the GIR abstraction, a CGP is represented as a unified directed acyclic graph (DAG), where the nodes represent operators in Ω, and the edges define the data flow between operators. The DAG has two representations, the *logical plan* for high-level query semantics (shown in Fig. 3(c)) and the *physical plan* for low-level operator details (shown in Fig. 3(d)). To construct the logical plan, GOpt offers a `GraphIrBuilder`, as exemplified in Section 5.2. For the physical plan, where operators are tied to the backend engine, GOpt enables backends to register their execution cost models through a

PhysicalCostSpec, ensuring accurate cost estimation, as detailed in Section 6.3. Once optimization is complete, the physical plan is converted into an execution plan via PhysicalConverter, making it executable on the backend. Two execution plans on different backends are exemplified in Fig. 3(e) and Fig. 3(f).

5.2 Query Parser

GOpt employs the ANTLR parser tool [1] to interpret queries in different query languages into an Abstract Syntax Tree (AST). Currently, GOpt supports Cypher[31] and Gremlin[53], as exemplified in Fig. 3(a) and Fig. 3(b), and preserves compatibility with the recent ISO standard GQL [4]. Due to the complexity of graph query languages, fully implementing the grammar for each query language is challenging. Therefore, the current implementation focuses on the key components, primarily involving query clauses for graph pattern matching and relational operations. The supported grammars can be found in [14]. Then, the AST is transformed into the GIR as a logical plan, as shown in Fig. 3(c), through a GraphIrBuilder. A code snippet for building the logical plan is as follows:

```
GraphIrBuilder irBuilder = new GraphIrBuilder();
pattern1 = irBuilder.patternStart()
  .getV(Alias("v1"),AllType())
  .expandE(Tag("v1"),Alias("e1"),AllType(),Dir.OUT)
  .getV(Tag("e1"),Alias("v2"),AllType(),Vertex.END)
  .expandE(Tag("v2"),Alias("e2"),AllType(),Dir.OUT)
  .getV(Tag("e2"),Alias("v3"),AllType(),Vertex.END)
  .patternEnd();
pattern2 = irBuilder.patternStart()
  .getV(Alias("v1"),AllType())
  .expandE(Tag("v1"),Alias("e3"),AllType(),Dir.OUT)
  .getV(Tag("e3"),Alias("v3"),BasicType("Place"),Vertex.END)
  .patternEnd();
query = irBuilder.join(pattern1,pattern2,
    Keys([Tag("v1"), Tag("v3")]), JoinType.INNER)
  .select(Expr("v3.name='China'"))
  .group(Keys(Tag("v2")),AggFunc.COUNT, Alias("cnt"))
  .order(Keys(Tag("cnt")), Order.ASC, Limit(10));
```

Note that Alias() defines an alias for results, accessible via Tag() in later operations. With the GraphIrBuilder, the logical plan is constructed in a language-independent manner, facilitating the subsequent optimization process. Henceforth, when optimizing CGPs, we are referring to optimizing their GIR form.

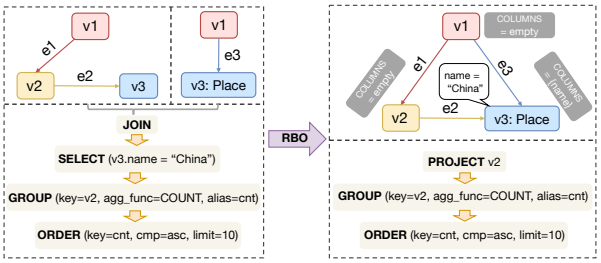
6 Optimization

In this section, we present our optimization techniques: RBO with heuristic rules for CGPs (Section 6.1), type inference for arbitrary patterns (Section 6.2), and CBO techniques (Section 6.3).

6.1 Rule-based Optimization

Given a CGP Q, GOpt first employs Rule-Based Optimizations (RBO) to optimize Q in a heuristic way. These pre-defined rules consider possible optimizations across patterns and relational operators. Note that the heuristic rules are designed to be extensible and pluggable. We introduce four representative rules in the following.

FilterIntoPattern. Consider a pattern followed by a SELECT. The FilterIntoPattern aims to push filters into patterns to reduce the number of intermediate results. Querying pattern in graph databases often yields numerous matches, so the users usually apply SELECT to narrow them down. This rule enhances efficiency by integrating filters with graph operators during pattern matching. For example,



(a) Logical Plan (b) Optimized Logical Plan after RBO

Figure 4: An Example of Rule-Based Optimizations

the SELECT operator in Fig. 4(a) is pushed down into the pattern, applying constraints during matching v_3 as shown in Fig. 4(b).

FieldTrim. Given a pattern, the FieldTrim is designed to eliminate unnecessary intermediate data during query execution, which is useful when the aliased intermediate results are no longer needed in the subsequent query processing. For example, in Fig. 4(a), since only v_3's name is used in the query, we specify retrieving just this attribute during pattern matching using COLUMNS. Similarly, since the query focuses solely on v_2's count after pattern matching, FieldTrim uses a PROJECT operator to trim the other vertices and edges.

JoinToPattern. For two patterns connected by a JOIN, the JoinToPattern aims to merge them into one with the join keys as common vertices and/or edges, effective under homomorphism-based semantics (Remark 3.1). Complexity arises when patterns are followed by other relational operators before connected by JOIN. If the patterns are followed by PROJECT, it does not affect join elimination because the PROJECT can be reordered with the JOIN without impacting the query results. If a SELECT follows, employing the FilterIntoPattern allows us to push the filter into the patterns and then eliminate the JOIN. However, if the patterns are followed by operators like GROUP, ORDER, or LIMIT, which alter the results of the pattern matching, the subsequent JOIN connecting the patterns cannot be eliminated. In Fig. 4(a), the two patterns can be merged into a single pattern with v_1 and v_3 as the join keys, as shown in Fig. 4(b).

ComSubPattern. For two patterns connected by binary operators such as UNION, JOIN, and DIFFERENCE, we design the ComSubPattern to identify common subpatterns in the two patterns, and save the computation cost by matching the common subpattern only once. For example, when querying the following CGP:

```
(v1:Person)-[]->(v2:Person)-[]->(:Product)
UNION (v1:Person)-[]->(v2:Person)-[]->(:Place),
```

ComSubPattern identifies the common subpattern (v1:Person)-[]->(v2:Person) across the binary UNION operator. It computes and clones the results, which are then expanded for (v2:Person)-[]->(:Product) and (v2:Person)-[]->(:Place) separately. Finally, the results are combined using the UNION operation.

Due to space constraints, more heuristic rules and complex optimization case studies are presented in [44].

6.2 Type Inference and Validation

In real applications, patterns in CGPs often contain arbitrary type constraints, which can be BasicType, UnionType, or AllType. While constraints like AllType enhance query flexibility by allowing patterns to be expressed without specific type constraints, they can significantly degrade performance if not properly optimized. Consider

Algorithm 1: Type Inference and Validation

Input : Pattern P, Graph Schema S
Output : Pattern P' with validated type constraints or INVALID

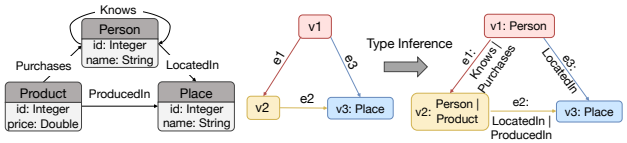
Initialize $Q = \{u | u \in V_P\}$ and sort Q by ascending $|\tau_P(u)|$;
while $Q \neq \emptyset$ **do**
  $u = Q.pop()$;
  CandVTypes = $\emptyset$ and CandETypes = $\emptyset$;
  for each $t \in \tau_P(u)$ **do**
    if $|N_P(u)| > 0$ **and** $|N_S(t)| = 0$ **then**
      remove t from $\tau_P(u)$;
    else
      CandVTypes = CandVTypes $\cup$ $N_S(t)$;
      CandETypes = CandETypes $\cup$ $N_S^E(t)$;
  for each $v \in N_P(u)$ **do**
    $\tau_P(v) = \tau_P(v) \cap$ CandVTypes;
    $\tau_P(e_{u,v}) = \tau_P(e_{u,v}) \cap$ CandETypes;
    if $\tau_P(v) = \emptyset$ **or** $\tau_P(e_{u,v}) = \emptyset$ **then**
      return INVALID
    if $\tau_P(v)$ is updated **and** $v \notin Q$ **then**
      insert v into Q;
Return P' with validated type constrains;

the pattern in Fig. 5(b), where both v_1 and v_2 use AllType, causing the engine to scan all vertices. However, according to the schema in Fig. 5(a), certain type combinations do not exist. For example, the engine scans all vertices including `Person`, `Product`, and `Place` for v_1, then expands to v_2. If expanding from `Product` vertices, v_2 is expected to match `Place` vertices. However, further expansion along e_2 reveals no connections from these `Place` vertices (for v_2) to any known `Place` vertex v_3, resulting in unnecessary intermediate results and wasted computation. Furthermore, during CBO, imprecise type constraints can lead to significant deviations between estimated and actual costs, resulting in suboptimal query plans. These issues raises the need for type inference in patterns.

For clarity, we first assume a schema-strict context [20], where the types of vertices and edges are predefined. A naive solution is to unfold each UnionType and AllType, checking all type combinations including BasicTypes only in the pattern against the graph schema. However, this is impractical as such combinations can be exponentially large. The Pathfinder algorithm [55] uses schema hints for inferring types in path patterns but does not apply to general patterns. Moreover, it requires inferred types to be BasicTypes, leading to enormous inferred patterns similar to the naive solution.

To address the drawbacks, we proposed Algorithm 1 to infer types with schema hints while preserve the flexibility with UnionType for multiple inferred types. The algorithm iteratively refines type constraints for each vertex and its neighboring edges and vertices based on graph schema connectivity. If no valid constraints can be assigned, it returns INVALID; otherwise, once all inferred type constraints are stable, it returns the pattern with validated type constraints. Though the algorithm focuses on outgoing adjacencies, incoming ones can be handled similarly. This inference and validation process removes invalid constraints from the original pattern, as exemplified in Fig. 5(c), enhancing execution performance by avoiding invalid type exploration in the backend engine.

Complexity Analysis. Assume the algorithm converges in k iterations, the overall complexity is $O(k|V_S|(d_S + |V_P|))$, where d_S is the maximum vertex degree in S. In practice, the algorithm converges quickly by starting with the most specific type constraints.



(a) Graph Schema (b) Origin Pattern (c) Pattern after Type Inference

Figure 5: An Example of Type Inference and Validation

Remark 6.1. *We address cases involving schema-loose systems, such as Neo4j, where incomplete schema information prevents the filtering of invalid type constraint assignments in the pattern. To overcome this, we can retrieve the schema information directly from the data graph during the initialization phase. Schema extraction can be performed using methods such as those described in [24] or system-provided tools like APOC [12] in Neo4j. Once the schema is established, we can incrementally track changes caused by data modification operations, ensuring the schema remains up to date for subsequent optimization. If the data is not initially present, optimization is unnecessary at this stage, and the schema extraction process can be deferred until a sufficient amount of data is available.*

6.3 Cost-based Optimization

We introduce Cost-Based Optimizations (CBO) for processing patterns in CGPs, incorporating data statistics, backend engine variations, and complex type constraints. Inspired by prior works[16, 41, 63], we focus on two principles: (1), high-order statistics [41] for precise cardinality estimation, and (2) hybrid pattern joins combining WcoJoin [48] and binary joins for improved performance. However, directly applying these techniques face challenges: (1) existing high-order statistics only support patterns with BasicTypes, and cannot handle UnionTypes, and (2) backend engines may lack efficient implementations of join operations, particularly WcoJoin (e.g. Neo4j). We explain how GOpt addresses these challenges below.

6.3.1 Cardinality Estimation. We leverage GLogue [41], which precomputed pattern frequencies for cardinality estimation. Techniques from [41], such as graph sparsification, pattern encoding, and efficient maintenance and lookup in GLogue, are directly utilized. Our focus is on processing patterns with arbitrary type constraints.

Basically, given two subgraphs P_{s_1}, P_{s_2} of P_t with $E_{P_t} = E_{P_{s_1}} \cup E_{P_{s_2}}$, by assuming the independent presence of patterns in the data graph, we compute the frequency of P_t as follows:

$$\mathcal{F}_{P_t} = \frac{\mathcal{F}_{P_{s_1}} \times \mathcal{F}_{P_{s_2}}}{\mathcal{F}_{P_{s_1} \cap P_{s_2}}} \tag{1}$$

where $P_{s_1} \cap P_{s_2}$ denotes the common parts of P_{s_1} and P_{s_2}.

For the case $V_{P_t} \setminus V_{P_s} = \{v\}$, that the edges $e_1, \ldots, e_n \in E_{P_t} \setminus E_{P_{s_1}}$ are iteratively appended on P_s and generates intermediate patterns $P_1, \ldots, P_n$, where $E_{P_i} \setminus E_{P_{i-1}} = \{e_i\}$ and $P_n = P_t$. We define the *expand ratio* σ_{e_i}, to reflect the change in pattern frequency when expanding edge $e_i = (v_i, v)$, as follows:

$$\sigma_{e_i} = \begin{cases} \frac{\sum_{t_{e_i} \in \tau(e_i)} \mathcal{F}_{t_{e_i}}}{\sum_{t_{v_i} \in \tau(v_i)} \mathcal{F}_{t_{v_i}}} & \text{if v } \notin V(P_i) \\ \frac{\sum_{t_{e_i} \in \tau(e_i)} \mathcal{F}_{t_{e_i}}}{\sum_{t_{v_i} \in \tau(v_i)} \mathcal{F}_{t_{v_i}} \times \sum_{t_v \in \tau(v)} \mathcal{F}_{t_v}} & \text{if v} \in V(P_i) \end{cases}$$

It should be noted that the type constraints here, $\tau(v)$ and $\tau(e)$, can be BasicType, UnionType, or AllType. Based on this, the frequency

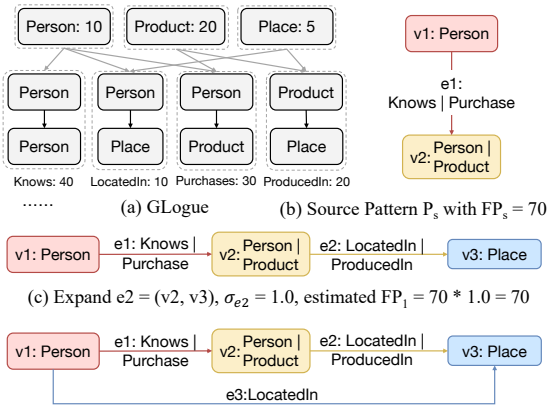


Figure 6: An Example of Cardinality Estimation

of P_t can be estimated as follows:

$$\mathcal{F}_{P_t} = \mathcal{F}_{P_s} \times \prod_{e_i \in E_{P_t} \setminus E_{P_s}} \sigma_{e_i} \tag{2}$$

This equation provides a cardinality estimation approach for patterns with arbitrary type constraints based on the subgraph frequencies and the *expand ratio* of the expanded edges.

Lastly, we design a GLogueQuery, providing a unified interface getFreq for the cardinality estimation of arbitrary patterns P, as follows: Given a pattern P, if it only contains BasicTypes, the frequency can be queried from GLogue directly. Otherwise, we alternately and iteratively apply Eq. 1 and Eq. 2 to estimate P's frequency. This process continues until the subpattern either exists in GLogue or GLogueQuery (which has been computed and cached), or simplifies to a single vertex or edge whose frequency can be obtained by summing up the frequencies of the contained basic types.

Example 6.1. Given the GLogue in Fig. 6(a) and the source pattern P_s in Fig. 6(b) with $\mathcal{F}_{P_s} = 70$, we show an example of cardinality estimation for pattern P_t in Fig. 6(c-d). First we expand e_2 of UnionType, as shown in Fig. 6(c), and get $\sigma_{e_2} = \frac{\mathcal{F}_{\texttt{LocatedIn}} + \mathcal{F}_{\texttt{ProducedIn}}}{\mathcal{F}_{\texttt{Person}} + \mathcal{F}_{\texttt{Product}}} = 1.0$. Then we consider e_3 of BasicType, as shown in Fig. 6(d), and compute $\sigma_{e_3} = \frac{\mathcal{F}_{\texttt{LocatedIn}}}{\mathcal{F}_{\texttt{Person}} \times \mathcal{F}_{\texttt{Place}}} = 0.2$. Thus, we have $\mathcal{F}_{P_t} = 70 \times 1.0 \times 0.2 = 14$.

6.3.2 Registerable Physical Plan. The PatternJoin equivalent rule ensures the correctness of pattern transformations during CBO.

PatternJoin. Given data graph G and pattern P_t, with P_{s_1} and P_{s_2} where $P_t = P_{s_1} \bowtie_k P_{s_2}$ and $\bowtie_k$ is the join operator with join key $k = V_{P_{s_1}} \cap V_{P_{s_2}}$. Let $R(P, G)$, or $R(P)$ for brevity, represent the results of matching P in G. Under homomorphism-based matching semantics (with an all-distinction operator for other semantics, as discussed in Remark 3.1), $R(P_t)$ can be computed by:

$$R(P_t) = R(P_{s_1}) \bowtie_k R(P_{s_2}) \tag{3}$$

`PhysicalCostSpec`. Following existing works [47, 63], two execution strategies, binary join and vertex expansion, are commonly used to implement PatternJoin. However, different backends may implement these strategies differently, potentially leading to suboptimal query plans if costs are not accurately calculated. To address this, we introduce a `PhysicalCostSpec` interface, allowing backends to register the cost models for their specific operator implementations:

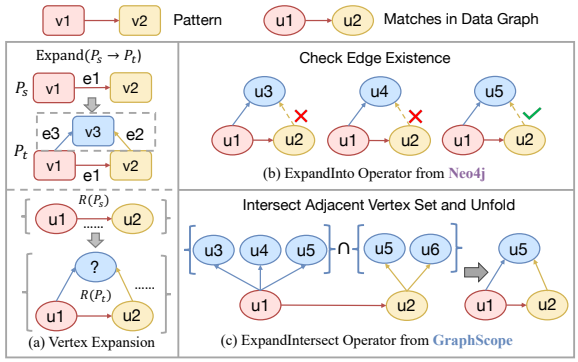


Figure 7: An Example of Vertex Expansion Implementations

```
interface PhysicalCostSpec {
    // Compute the cost of joining two patterns.
    // Recall that GlogueQuery provides cardinality of patterns.
    Double computeCost(GlogueQuery gq, Pattern Ps1, Pattern Ps2);
}
```

We use Neo4j and GraphScope as examples to show how to register their cost models via `PhysicalCostSpec`:

(1) **Binary Join**: denoted as $\texttt{JOIN}(P_{s_1}, P_{s_2} \rightarrow P_t)$, this strategy computes the mappings of P_{s_1} and P_{s_2} separately and then joins the results to produce the mappings of P_t. Both Neo4j and GraphScope implement this using HashJoin and register the corresponding cost as $\mathcal{F}_{P_{s_1}} + \mathcal{F}_{P_{s_2}}$ according to [41] in the `PhysicalCostSpec`.

(2) **Vertex Expansion**: Denoted as $\texttt{EXPAND}(P_s, P_v \rightarrow P_t)$, this strategy applies when $V_{P_v} = V_{P_t} \setminus V_{P_s} = \{v\}$, meaning P_t is formed by appending a vertex v to P_s. Neo4j implements it with the ExpandInto operator, while GraphScope utilizes the ExpandIntersect operator, based on algorithms proposed in [47]. Each system can register the cost model as shown below:

```
// Neo4j's registration
class ExpandIntoSpec implements PhysicalCostSpec {
    @Override
    Double computeCost(GlogueQuery gq, Pattern Ps, Pattern Pv) {
        Pattern Pi = Ps, cost = 0
        for each e in Pv {
            Pi <- append e to Pi
            cost += gq.getFreq(Pi)
        }
        return cost;
    }
}
// GraphScope's registration
class ExpandIntersectSpec implements PhysicalCostSpec {
    @Override
    Double computeCost(GlogueQuery gq, Pattern Ps, Pattern Pv) {
        return |Pv| * (gq.getFreq(Ps));
    }
}
```

Please note that during the CBO process, the plan remains in GIR format, which will be converted into backend-executable forms after the optimization is completed, as discussed in Section 7.

Example 6.2. We illustrate the different vertex expansion implementations used by Neo4j and GraphScope in Fig. 7. Initially, pattern P_s matches to (u_1, u_2) in the data graph (Fig. 7(a)). To match P_t by expanding e_3 and e_2, Neo4j first expands e_3, yielding three mappings (Fig. 7(b)), and then performs ExpandInto to find matches for e_2 connecting u_2 to u_3, u_4 and u_5, respectively. As ExpandInto flattens the intermediate matching results, the cost is the sum of frequencies of intermediate patterns. In contrast, GraphScope uses

Algorithm 2: Graph Optimizer

```
   Input  : Pattern P, GLogueQuery GQ
   Output : Optimal (plan, cost) for P
1  (plan*, cost*) ← GreedyInitial(P, GL);
2  Initialize map M = {p:(plan, cost)} with patterns of size 1 and 2 precomputed;
3  RecursiveSearch(P, GQ, M, cost*);
4  Return M.get(P);
5  Procedure RecursiveSearch(P, GQ, M, cost*)
6  if M contains P then Return;
7  for e ∈ getCands(P) do
8      if getLowerBound(M, e) ≥ cost* then continue;
9      if e is of EXPAND(P_s, P_v → P) then
10         RecursiveSearch(P_s, GQ, M, cost*);
11         cost'=M.getCost(P_s)+GQ.getFreq(P)+e.computeCost(GQ,P_s,P_v);
12     else if e is of JOIN({P_s1, P_s2} → P) then
13         RecursiveSearch(P_s1, GQ, M, cost*);
14         RecursiveSearch(P_s2, GQ, M, cost*);
15         cost' = M.getCost(P_s1) + M.getCost(P_s2) + GQ.getFreq(P) +
               e.computeCost(GQ, P_s1, P_s2);
16     if cost' < M.getCost(P) then
17         update P in M with the new plan' and cost';
```

ExpandIntersect, which begins by finding the match set $R(P_1)$ by expanding e_3, then expands e_2 and intersects the matched set with $R(P_1)$ to obtain $R(P_t)$, and finally unfolds the match set (Fig. 7(c)). ExpandIntersect reduces computation by avoiding flattening intermediate results, with the cost defined in the snippet.

Cost Model. The computational cost of the physical plan is defined as the total cost of its operators, calculated by computeCost method in `PhysicalCostSpec` and adjusted by a normalized factor α_{op} to reflect each operator's relative processing expense. For distributed graph databases, we also consider the communication cost. Assuming vertices are randomly distributed across machines, with edges located on those machines hosting their source and destination vertices, we measure communication costs by the number of intermediate results in a simplified way, since distributed systems need to fetch remote data for further processing.

6.3.3 Top-Down Search Framework. Based on new cardinality estimation methods and backend-registered cost models, we introduce a top-down search framework for optimizing patterns in Algorithm 2. We begin with a greedy search to find an initial solution as a tight bound for further exploration. A map M is then initialized to store optimal plans and costs for the subgraphs of P. We then search for the optimal plan for P in a top-down framework. In the recursive search, getCands identifies candidate edges for pattern transformation targeting P by enumerating the subgraphs. For each candidate, we prune non-optimal search branches based on getLowerBound. After that, we continue searching for the optimal plan for sub-patterns (lines 10, 13-14), until they have already been computed and stored in M (line 6). Using the subplan's cost, we compute P's cost by accumulating communication costs (if on distributed backend) and computation cost based on the backend-registered `PhysicalCostSpec` (line 11, 15). If P's cost is lower than previous bests, we update the optimal plan for P in M (lines 16-17).

Initialize with Greedy Search. GreedyInitial is designed to identify a good initial solution as a tight bound for branch pruning. Starting with the queried pattern P, it iteratively removes candidate edges with the smallest cost, including `EXPAND` and `JOIN`, in a greedy manner, with the edges' costs and subpatterns' frequencies being accumulated to the pattern cost, until reducing the pattern to a single vertex. This initial solution helps Algorithm 2 prune unpromising branches, enhancing overall efficiency.

Prune Branch with Lower Bound. We design pruning strategies to filter out the search branches that cannot be optimal (line 9-10). Consider a candidate edge $\texttt{EXPAND}(P_s, P_v \rightarrow P)$. If P_s has been previously searched and its cost already exceeds $cost^*$, or, if the noncumulative cost of P (i.e., $\mathcal{F}_{P_s} + e.\text{computeCost}(P_s, P_v)$) exceeds $cost^*$, the branch can be pruned. Similar pruning strategies can be applied when the candidate edge e is of $\texttt{JOIN}(\{P_{s_1}, P_{s_2}\} \rightarrow P)$.

7 System Implementation

GOpt is built upon the Apache Calcite framework [22], a dynamic data management platform with a focus on relational databases. By integrating with Calcite, GOpt provides a unified approach for optimizing CGPs, combining proposed graph optimization techniques with the relational optimization techniques already present in Calcite. To build GIR, the graph operators are implemented as subclasses of the fundamental `RelNode` structure in Calcite to ensure smooth integration, facilitating a cohesive optimization process for both graph patterns and relational operations.

Rule-based Optimization. In GOpt, we employ the HepPlanner, an RBO planner in Calcite, to implement our RBO techniques. Developing new heuristic rules involves two primary steps: (1) define a condition that determines the applicability of the rule, and (2) specify an action that modifies the subplan when the condition is met. For instance, in FilterIntoPattern, when a `SELECT` operator with filters targeting a tag associated with a graph operator is identified (the condition), we move the filter expression into the graph operator (the action). Furthermore, we incorporate Calcite's existing rules for relational operators into GOpt, thereby leveraging the established relational optimizations.

Cost-based Optimization. We further adopt the VolcanoPlanner, a CBO planner in Calcite, to implement the CBO techniques for patterns. First, we introduce a new metadata provider, `GraphMdQuery`, which provides metadata for both graph and relational operators. For graph operators, we implement the handlers for `RowCount` using GLogueQuery and `NonCumulativeCost` using `PhysicalCostSpec`, as detailed in Section 6.3, to deliver more accurate cardinality estimates and backend-specific cost estimations. For relational operators, we leverage Calcite's built-in metadata handlers. Then to optimize the patterns, we implement PatternJoin through `JOIN` and `EXPAND`, corresponding to the hybrid join strategy. Leveraging these transformation rules, the VolcanoPlanner generates equivalent plan sets for patterns, where each set includes all possible plans for a specific induced subgraph of the original pattern. With `GraphMdQuery` and PatternJoin integrated into VolcanoPlanner, we invoke the CBO process to optimize CGPs in a top-down manner. As the search progresses, the VolcanoPlanner consults `GraphMdQuery` for estimated cardinalities and costs to calculate the total cost of the plan.

Remark 7.1. *To optimize CGPs, we encapsulate patterns within a specialized relational* `SCAN` *operator, with its cost determined by the pattern frequencies, which enables us to leverage Calcite's built-in metadata handlers and relational optimization rules to enhance the relational components of CGPs. This approach allows GOpt to benefit*

from Calcite's robust relational optimization techniques. For example, when executing a GROUP *operator, Calcite's AggregatePushDown rule can help reduce intermediate results and improve efficiency.*

Moreover, more sophisticated cost estimation is required for patterns with filters after they are incorporated into patterns (as described in FilterIntoPattern in Section 6.1). Currently, we followed [41] to predefine selectivity values (e.g., 0.1) for vertices and edges with filtering conditions. In future work, we aim to integrate advanced selectivity estimation techniques, such as histograms and sampling provided by Calcite, to further improve optimizing patterns with filters.

Backend Integration. The `PhysicalConverter` interface is designed for GOpt integration, defined as follows:

```
interface PhysicalConverter {
    ExecOp convert(JOIN op);
    ExecOp convert(EXPAND op);
    // Other physical operator conversions omitted...
}
```

Based on this interface, we offer two layers of backend integration: (1) Backends can directly implement the interface to convert each operator from the optimized physical plan into a backend executable ExecOp. Using `PhysicalConverter`, GOpt traverses and converts the physical plan into a backend executable plan. We have integrated GOpt with Neo4j, which is Java-native, using this method [13]. (2) GOpt includes a build-in `PhysicalConverter` implementation that converts the optimized physical plan into a Google Protocol Buffers (protobuf) [49] format, and submits it directly to the backend engine for integration. The backend engine can parse this plan, transform it into its native execution plan, and execute it. We have utilized this method to integrate GOpt with Alibaba's GraphScope platform [3], enabling execution on its distributed dataflow engine [51]. More details on the integration are provided in [44].

8 Experiments

We conducted experiments to demonstrate the effectiveness of the proposed optimization techniques in GOpt and its ability to integrate with multiple query languages and existing systems.

8.1 Experiment Settings

Benchmarks. We utilized the Linked Data Benchmark Council (LDBC) social network benchmark [42] to test GOpt. 4 datasets in Table 3 were generated using the official data generator, where G_{sf} denotes the graph with scale factor sf. We adopted the LDBC Interactive workloads $IC_{1\ldots12}$, and Business Intelligence workloads $BI_{1\ldots14,16,17,18}$, excluding $IC_{13,14}$ and $BI_{15,19,20}$ since they either reply on shortest-path algorithm or stored procedures that are not supported in GraphScope. Additionally, we designed query sets $QR_{1\ldots8}$ for heuristic rules, $QT_{1\ldots5}$ for type inference, and $QC_{1\ldots4(a|b)}$ for CBO. All the queries can be found in [8]. By default, these queries were implemented in Cypher, with LDBC workloads using the official Cypher implementations [27]. Certain queries written in Gremlin were manually translated from their Cypher counterparts.

Compared Systems. We conducted experiments on two systems: (1) GraphScope v0.29.0 [3], a distributed graph system supporting Gremlin queries. It optimizes queries using a rule-based optimizer and executes them on the distributed dataflow engine Gaia [51]. (2) Neo4j v4.4.9 [6], a widely used graph database supporting Cypher queries. It optimizes queries using its CypherPlanner, which employs heuristic rules and cost-based optimization, and runs on a single machine using Neo4j's interpreted runtime.

Table 3: The LDBC datasets.

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

For each query, we highlight the following two execution plans:

- GOpt-plan: The optimized plan generated by GOpt. Note that the GOpt-plans for running on Neo4j and GraphScope may be different due to the `PhysicalConverter` (Section 7).
- Neo4j-plan: The optimized plans generated by Neo4j's optimizer CypherPlanner. Although Neo4j's execution engine may not be the most competitive, its optimizer remains as a strong baseline, being one of the most well-developed in the community.

Configurations. We conducted tests on a cluster of up to 16 machines, each with dual 8-core Intel Xeon E5-2620 v4 CPUs at 2.1GHz, 512GB memory, and a 10Gbps network. To avoid disk I/O effects, all graph data was stored in memory. For the distributed experiments on GraphScope, vertices and properties were randomly assigned to machines, while edges and properties were placed on machines hosting their source and destination vertices. We excluded query parsing and optimization time from our analysis due to its minimal impact on runtime. Queries exceeding 1 hour were marked as `OT`.

8.2 Micro Benchmarks

We evaluated GOpt 's optimization techniques in Section 6, namely heuristic rules, type inference, and CBO. To demonstrate multiple query language optimization, we further write queries in Gremlin for evaluation. The tests were conducted on G_{30} using a single machine with 32 threads, with the queries executed on GraphScope. Besides the "Optimizing Gremlin Queries" test, where all GOpt optimization techniques were applied, we ran controlled experiments to isolate specific techniques. For example, when assessing heuristic rules, we used queries with only explicit types and disabled type checker and CBO to avoid interference from other techniques.

Heuristic Rules. To evaluate the heuristic rules, we compared performance with and without their application. As shown in Fig. 8(a), FilterIntoPattern ($QR_{1,2}$) demonstrated the most significant impact, improving performance by orders of magnitude. FieldTrim ($QR_{3,4}$) achieved an average performance improvement of 4.8×, JoinToPattern ($QR_{5,6}$) enhanced performance by 40.2×, and the ComSubPattern ($QR_{7,8}$) achieved a 34.3× improvement. These findings, particularly the substantial gains achieved with the FilterIntoPattern, underscore the importance of GIR, which facilitates the effective co-optimization between pattens and relational operators.

Type Inference. We tested the impact of type inference by comparing query execution performance with it enabled versus disabled on queries without explicit type constraints. As illustrated in Fig. 8(b), type inference consistently accelerates all queries, with an average speedup of 51×. Notably, QT_2 achieves a 209× improvement, thanks to GOpt's capability to infer type constraints, thereby avoiding unnecessary traversal of irrelevant vertices and edges.

Cost-based Optimization. We tested CBO using $QC_{1\ldots4(a|b)}$, focusing on pattern matching for a triangle, square, 5-path, and a

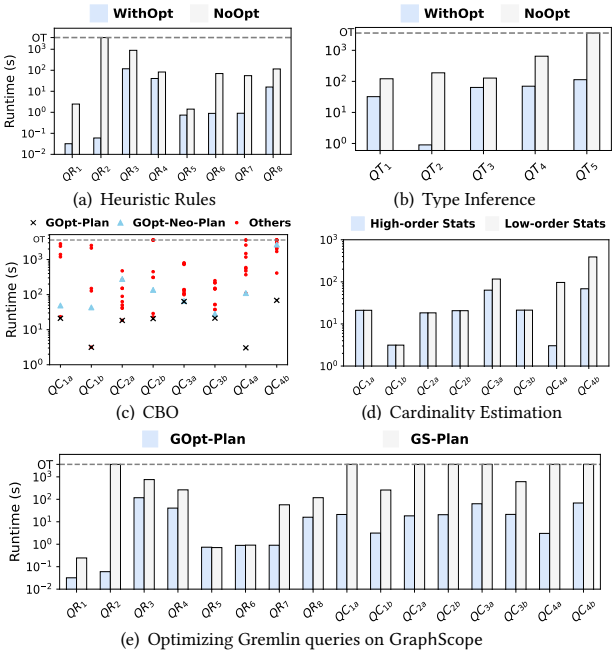


Figure 8: Results of Micro Benchmarks on GraphScope.

complex pattern consists of 7 vertices and 8 edges. Each query has two variants: *a* for queries with BasicTypes and *b* with UnionTypes. To compare performance, we designed up to 12 execution plans for each query: one GOpt-plan (marked "×"); one GOpt-Neo-Plan (marked "Δ"), which is generated by GOpt using Neo4j's cost model for `EXPAND` deliberately; and up to 10 randomly generated plans (marked "∘"). Results in Fig. 8(c) emphasize the critical role of CBO in pattern processing and demonstrate GOpt's effective use of backend-specific cost estimations. The GOpt-plan consistently outperformed, averaging 14.4× faster than GOpt-Neo-Plan due to its `PhysicalCostSpec` enabling backend-specific costs. This highlights the necessity of a cost model tailored to the backend system, as mismatches can lead to suboptimal plans. Furthermore, GOpt's plan was 117.8× faster than the average of the randomized plans. While GOpt may not always achieve the absolute best performance due to its reliance on estimated costs, it generally performs close to the optimal, demonstrating robustness and reliability.

We further evaluated GOpt's cardinality estimation by comparing GOpt-plans using GLogueQuery with high-order statistics against those with low-order statistics. As shown in Fig. 8(d), for 3 of 8 queries, the plans using high-order statistics achieved better performance, with an average speedup of 13.1×. High-order statistics were particularly effective for complex patterns such as QC_{4a}, where the plan utilizing high-order statistics was 31.9× faster.

Optimizing Gremlin Queries. We wrote the queries in Gremlin for GraphScope to generate GS-plans using its native rule-based optimizer and GOpt-plans after integrating GOpt. We compared GS-plans and GOpt-plans on GraphScope for *QR* and *QC* queries, excluding *QT* due to GraphScope's lack of type inference. As shown in Fig. 8(e), for *QR* queries testing heuristic rules, GOpt-plan outperformed GS-plan on most queries (except QR_5 and QR_6 due to GraphScope native optimizer's support of JoinToPattern), with an average speedup of 13.3×, especially for QR_2 where GS-plan runs

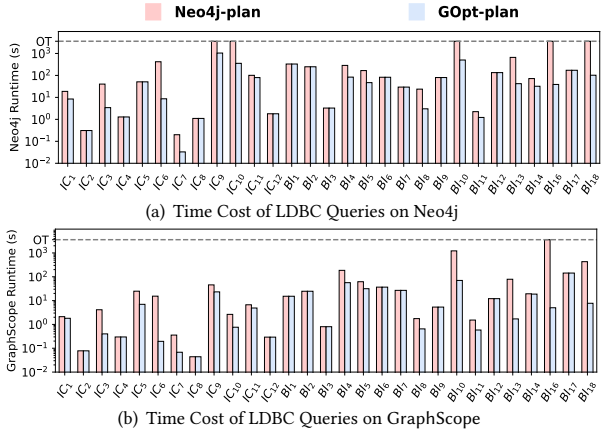


Figure 9: Results of Comprehensive Experiments.

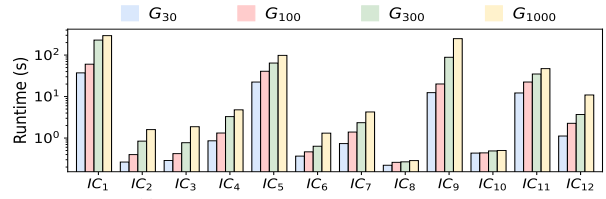
`OT`. This demonstrates the effectiveness of GOpt's new heuristic rules. For *QC* queries, GS-plans followed user-specified orders, most running `OT`, while GOpt-plan leveraged cost-based optimization with enhanced cardinality and cost estimation for better search orders, achieving an average speedup of 243.4×. Since GS-plans frequently yielding suboptimal results, we excluded them from further comparisons in the subsequent experiments. These results demonstrate that integrating GOpt into GraphScope can immediately enhance its performance for Gremlin queries. Moreover, GOpt extends GraphScope by enabling support for Cypher queries.

8.3 Comprehensive Experiments

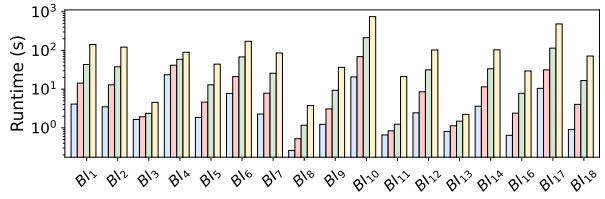
In this subsection, we compare GOpt with Neo4j's CypherPlanner by testing GOpt-plans and Neo4j-plans on both Neo4j and GraphScope. This comparison highlights GOpt's advanced optimization techniques and its ability to utilize backend-specific operators for superior performance. Experiments were conducted on G_{100} using a single machine with 32 threads, as Neo4j supports only single-machine deployment. For fairness, we explicitly specify all type constraints in the queries, given Neo4j's schema-loose design and lack of type inference support in its optimizer.

Neo4j-plan v.s. GOpt-plan on Neo4j. We began by comparing the performance of Neo4j-plan and GOpt-plan on Neo4j, as shown in Fig. 9(a). GOpt-plan outperformed Neo4j-plan in 16 of 29 queries, with an average speedup of 15.8× for these queries and 9.2× overall, due to GOpt's advanced optimization techniques. In IC_6, GOpt achieved the most significant improvement of 48.6× by employing hybrid-join strategy, whereas Neo4j relies on multiple Expand, causing excessive intermediate results. In IC_{10}, BI_{10} and BI_{18}, where Neo4j-plan runs `OT`, GS-plan optimizes search orders with more precise cost estimation, enhancing the performance. In IC_9 and BI_{13}, GS-plan used Calcite's AggregatePushDown rule, pushing aggregation earlier to reduce intermediate results, a feature unsupported by CypherPlanner. Due to the space limit, we provide more details of GOpt's performance gains of these queries in [44]. Overall, these experiments highlight the efficacy of GOpt's optimization techniques in improving query processing in Neo4j.

Neo4j-plan v.s. GOpt-plan on GraphScope. We further compared the performance of the two plans on GraphScope, as shown in Fig. 9(b). Note that to execute Neo4j-plans on GraphScope, we



(a) Data Scale of IC queries on GraphScope



(b) Data Scale of BI queries on GraphScope

Figure 10: Results of Data Scale Experiments.

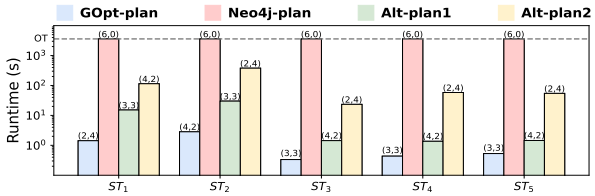


Figure 11: Performance of S-T Paths.

manually translated them into GraphScope-compatible versions. GOpt-plan outperformed Neo4j-plan in 17 of 29 queries, with an average speedup of 56.2× for these queries and 33.4× overall, the most significant being 78.7× in IC_6. This improvement is greater than on Neo4j since GOpt enables optimized operator implementations like ExpandIntersect in GraphScope via `PhysicalConverter`. For example, in the cyclic pattern IC_5, both plans use ExpandInto on Neo4j and show similar performance. However, on GraphScope, GOpt-plan selects the more efficient ExpandIntersect, achieving a 3.6× speedup, whereas Neo4j-plan sticks with ExpandInto, executed as a Expand followed by Select. Other optimized queries also show improved performance on GraphScope for similar reasons, additionally benefiting from the efficient ExpandIntersect. Overall, these results underscore the effectiveness of GOpt's optimization techniques in enhancing query processing on GraphScope.

8.4 Data Scale Experiments

We also tested the scalability of GraphScope integrated with GOpt, which supports distributed execution via Gaia, on a cluster of 16 machines, each equipped with 2 threads. We conducted IC and BI queries across various dataset sizes. For IC queries, we randomly selected 8 parameter values per query, aggregating their runtimes to obtain a representative result and mitigate bias from short execution times. As shown in Fig. 10(a), when graph size increases by 30× from G_{30} to G_{1000}, runtime for the most impacted query (IC_9) rises by 20.0×, while the average performance degradation is only 6.3×, which is impressive given the substantial data scale. Fig. 10(b) presents the results for BI queries, demonstrating effective scalability with the growth of graph data. As graph size increases from G_{30} to G_{1000}, average performance degradation is 30.4×. This is expected, as BI queries are typically more complex, causing intermediate results to grow steadily with the graph size. These findings highlight the importance of effective query optimization, particularly the CBO, in preventing poor search orders that could generate intermediate results exponentially proportional to the graph size.

8.5 Case Study

GOpt with GraphScope backend is widely deployed in production at Alibaba. This case study focuses on its application in fraud detection. Fraudsters transfer funds through multiple intermediaries before another fraudster withdraws the money. This problem can be described as an s-t path in CGP, with the Cypher query as follows:

```
MATCH (p1:PERSON)-[p:*$k]->(p2:PERSON)
WHERE p1.id IN $S1 and p2.id IN $S2
RETURN p
```

Finding paths in large graphs is challenging due to exploding intermediate results. Two strategies are common: (1) single-direction expansion, expanding k hops from S_1 and filtering ends in S_2; (2) bidirectional search, which starts from S_1 and S_2 simultaneously and joins sub-paths in the middle. While bidirectional search is generally more efficient, our case study shows the middle vertex is not always optimal for joining. We conducted experiments on a real-world graph with 3.6 billion vertices and 21.8 billion edges, setting $k = 6$ and creating five queries $ST_{1\ldots5}$ with distinct (S_1, S_2) pairs randomly obtained from real applications. For each query, we generated Neo4j-plan and two alternatives with different search orders to compare with GOpt-plan. As shown in Fig. 11, path join positions are indicated above each bar, e.g., $(2, 4)$ means a 2-hop sub-path from S_1 joins with a 4-hop sub-path from S_2, while all Neo4j-plans execute single-direction expansion from S_1 to S_2. GOpt-plans outperform alternatives from 3× to 20×, whereas all single-direction plans run `OT`. In GOpt-plans, join positions are not always centered, as seen in ST_1 and ST_2, due to the varying number of vertices in S_1 and S_2 in real queries. By properly configuring the costs for scans (now they are the number of vertices in the source sets), GOpt's CBO can automatically determine the optimal join positions. This demonstrate the effectiveness of GOpt in optimizing CGPs in real-world applications.

9 Conclusion

In this paper, we present GOpt, a modular, graph-native query optimization framework for CGPs, which combine patterns and relational operations. CGPs can be expressed in multiple graph query languages and executed on various backends, necessitating a unified optimization approach. GOpt addresses this need by employing a unified GIR, and using a `GraphIrBuilder` to translate queries from diverse languages into this GIR, facilitating advanced optimization techniques. To optimize CGPs, GOpt utilizes comprehensive heuristic rules to refine the interactions between patterns and relational operations. It also includes an automatic type inference algorithm to uncover implicit type constraints in query patterns, and employs CBO techniques for further refinement. During the CBO process, GOpt offers a `PhysicalCostSpec`, allowing backends to register their operator cost models, thereby enhancing the accuracy of cost estimation. After optimization is complete, GOpt generates an execution plan via `PhysicalConverter`, which can be executed on the backend. Extensive experiments demonstrate that GOpt significantly improves query performance compared to state-of-the-art systems in both synthetic benchmarks and real-world applications.

References

[1] 2024. ANTLR: ANother Tool for Language Recognition. https://www.antlr.org/

[2] 2024. GOpt's open-source project on GitHub. https://github.com/alibaba/GraphScope/tree/main/interactive_engine/compiler

[3] 2024. GraphScope's open-source project on GitHub. https://github.com/alibaba/GraphScope

[4] 2024. Information technology — Database languages — GQL. https://www.iso.org/standard/76120.html

[5] 2024. JanusGraph: A Transactional Graph Database. https://janusgraph.org/

[6] 2024. Neo4j Graph Database. https://neo4j.com/.

[7] 2024. OrientDB. https://orientdb.com/

[8] 2024. Queries used for the experiments. https://github.com/alibaba/GraphScope/tree/main/interactive_engine/benchmark/queries/cypher_queries/experiments

[9] 2024. RDF Primer. https://www.w3.org/TR/rdf-primer/

[10] 2024. SPARQL Query Language for RDF. https://www.w3.org/TR/rdf-sparql-query/

[11] 2024. TigerGraph: Graph Analytics Platform. https://www.tigergraph.com/

[12] 2024. Viewing schema data with APOC Procedures in Neo4j. https://neo4j.com/docs/apoc/current/overview/apoc.schema/

[13] 2025. Integration of GOpt with Neo4j. https://github.com/shirly121/neo4j/tree/gopt_planner/community/cypher/cypher-planner/src/main/java/org/neo4j/cypher/internal/compiler/common

[14] 2025. Supported Grammars in GOpt. https://github.com/alibaba/GraphScope/tree/main/interactive_engine/compiler/src/main/antlr4

[15] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)* 42, 4 (2017), 1–44.

[16] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows. *Proc. VLDB Endow.* 11, 6 (oct 2018), 691–704. https://doi.org/10.14778/3184470.3184473

[17] Renzo Angles. 2018. The Property Graph Database Model. In *Alberto Mendelzon Workshop on Foundations of Data Management*. https://api.semanticscholar.org/CorpusID:43977243

[18] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1421–1432. https://doi.org/10.1145/3183713.3190654

[19] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5, Article 68 (sep 2017), 40 pages. https://doi.org/10.1145/3104031

[20] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savkovic, Michael Schmidt, Juan Sequeda, Slawek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Dusan Zivkovic. 2023. PG-Schema: Schemas for Property Graphs. *Proceedings of the ACM on Management of Data* 1, 2 (June 2023), 1–25. https://doi.org/10.1145/3589778

[21] AWS Neptune. 2024. https://aws.amazon.com/neptune/.

[22] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD/PODS '18)*. ACM. https://doi.org/10.1145/3183713.3190662

[23] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*. 1199–1214.

[24] Angela Bonifati, Stefania-Gabriela Dumbrava, Emile Martinez, Fatemeh Ghasemi, Malo Jaffré, Pacome Luton, and Thomas Pickles. 2022. DiscoPG: Property Graph Schema Discovery and Exploration. *Proc. VLDB Endow.* 15, 12 (2022), 3654–3657.

[25] Mario Cannataro, Pietro Hiram Guzzi, and Pierangelo Veltri. 2010. Protein-to-protein interactions: Technologies, databases, and algorithms. *ACM Comput. Surv.* 43, 1 (2010), 1:1–1:36. https://doi.org/10.1145/1824795.1824796

[26] Cypher For Gremlin. 2024. https://github.com/opencypher/cypher-for-gremlin. [Online; accessed 24-December-2024].

[27] Cypher Implementations of LDBC Social Network Benchmark. 2024. https://github.com/ldbc/ldbc_snb_interactive_v1_impls/. [Online; accessed 24-December-2024].

[28] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2020. Aggregation Support for Modern Graph Analytics in TigerGraph. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 377–392. https://doi.org/10.1145/3318464.3386144

[29] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, Youyang Yao, Qiang Yin, Wenyuan Yu, Jingren Zhou, Diwen Zhu, and Rong Zhu. 2021. GraphScope: A Unified Engine for Big Graph Processing. *Proc. VLDB Endow.* 14, 12 (jul 2021), 2879–2892.

[30] Gary William Flake, Steve Lawrence, C Lee Giles, and Frans M Coetzee. 2002. Self-organization and identification of web communities. *Computer* 35, 3 (2002), 66–70.

[31] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*. 1433–1445.

[32] Benoit Gaüzère, Luc Brun, and Didier Villemin. 2015. Graph kernels in chemoinformatics. In *Quantitative Graph TheoryMathematical Foundations and Applications*, Matthias Dehmer and Frank Emmert-Streib (Eds.). CRC Press, 425–470. https://hal.archives-ouvertes.fr/hal-01201933

[33] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. 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, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 337–348. https://doi.org/10.1145/2463676.2465300

[34] Mohamed S. Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G. Aref, and Mohammad Sadoghi. 2018. GRFusion: Graphs as First-Class Citizens in Main-Memory Relational Database Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1789–1792. https://doi.org/10.1145/3183713.3193541

[35] Tao He, Shuxian Hu, Longbin Lai, Dongze Li, Neng Li, Xue Li, Lexiao Liu, Xiaojian Luo, Bingqing Lyu, Ke Meng, Sijie Shen, Li Su, Lei Wang, Jingbo Xu, Wenyuan Yu, Weibin Zeng, Lei Zhang, Siyuan Zhang, Jingren Zhou, Xiaoli Zhou, and Diwen Zhu. 2024. GraphScope Flex: LEGO-like Graph Computing Stack. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 386–399. https://doi.org/10.1145/3626246.3653383

[36] Yanqing Hu, Shenggong Ji, Yuliang Jin, Ling Feng, H. Eugene Stanley, and Shlomo Havlin. 2018. 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. https://doi.org/10.1073/pnas.1710547115 arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.1710547115

[37] Guodong Jin and Semih Salihoglu. 2022. Making RDBMSs Efficient on Graph Workloads through Predefined Joins. *Proc. VLDB Endow.* 15, 5 (jan 2022), 1011–1023. https://doi.org/10.14778/3510397.3510400

[38] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1695–1698. https://doi.org/10.1145/3035918.3056445

[39] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment* 8, 10 (2015), 974–985.

[40] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. 2019. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1099–1112.

[41] Longbin Lai, Yufan Yang, Zhibin Wang, Yuxuan Liu, Haotian Ma, Sijie Shen, Bingqing Lyu, Xiaoli Zhou, Wenyuan Yu, Zhengping Qian, Chen Tian, Sheng Zhong, Yeh-Ching Chung, and Jingren Zhou. 2023. GLogS: Interactive Graph Pattern Matching Query At Large Scale. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 53–69. https://www.usenix.org/conference/atc23/presentation/lai

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

[43] Yunkai Lou, Longbin Lai, Bingqing Lyu, Yufan Yang, Xiaoli Zhou, Wenyuan Yu, Ying Zhang, and Jingren Zhou. 2024. Towards a Converged Relational-Graph Optimization Framework. arXiv:2408.13480 [cs.DB] https://arxiv.org/abs/2408.13480

[44] Bingqing Lyu, Xiaoli Zhou, Longbin Lai, Yufan Yang, Yunkai Lou, Wenyuan Yu, and Jingren Zhou. 2025. A Graph-native Optimization Framework for Complex Graph Queries. arXiv:2503.22091 [cs.DB] https://arxiv.org/abs/2503.22091

[45] Marios Meimaris and George Papastefanatos. 2017. Distance-Based Triple Reordering for SPARQL Query Optimization. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 1559–1562. https://doi.org/10.1109/ICDE.2017.227

[46] Lingkai Meng, Yu Shao, Long Yuan, Longbin Lai, Peng Cheng, Xue Li, Wenyuan Yu, Wenjie Zhang, Xuemin Lin, and Jingren Zhou. 2024. A survey of distributed graph algorithms on massive graphs. *Comput. Surveys* 57, 2 (2024), 1–39.

[47] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *arXiv preprint arXiv:1903.02076* (2019).

[48] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case optimal join algorithms. *Journal of the ACM (JACM)* 65, 3 (2018), 1–40.

[49] Protocol Buffers. 2024. https://protobuf.dev/overview/.

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

[51] Zhengping Qian, Chenqiang Min, Longbin Lai, Yong Fang, Gaofeng Li, Youyang Yao, Bingqing Lyu, Xiaoli Zhou, Zhimin Chen, and Jingren Zhou. 2021. 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)*. USENIX Association, 321–335. https://www.usenix.org/conference/nsdi21/presentation/qian-zhengping

[52] Kashif Rabbani, Matteo Lissandrini, and Katja Hose. 2021. Optimizing SPARQL Queries using Shape Statistics. In *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, Yannis Velegrakis, Demetris Zeinalipour-Yazti, Panos K. Chrysanthis, and Francesco Guerra (Eds.). OpenProceedings.org, 505–510. https://doi.org/10.5441/002/EDBT.2021.59

[53] Marko A. Rodriguez. 2015. The Gremlin Graph Traversal Machine and Language (Invited Talk). In *Proceedings of the 15th Symposium on Database Programming Languages* (Pittsburgh, PA, USA) *(DBPL 2015)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2815072.2815073

[54] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proc. VLDB Endow.* 1, 1 (aug 2008), 364–375. https://doi.org/10.14778/1453856.1453899

[55] Benjamin A. Steer, Alhamza Alnaimi, Marco A. B. F. G. Lotz, Félix Cuadrado, Luis M. Vaquero, and Joan Varvenne. 2017. Cytosm: Declarative Property Graph Queries Without Data Migration. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*, Peter A. Boncz and Josep Lluís Larriba-Pey (Eds.). ACM, 4:1–4:6. https://doi.org/10.1145/3078447.3078451

[56] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. 2008. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, Jinpeng Huai, Robin Chen, Hsiao-Wuen Hon, Yunhao Liu, Wei-Ying Ma, Andrew Tomkins, and Xiaodong Zhang (Eds.). ACM, 595–604. https://doi.org/10.1145/1367497.1367578

[57] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guo Tong Xie. 2015. SQLGraph: An Efficient Relational-Based Property Graph Store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1887–1901. https://doi.org/10.1145/2723372.2723732

[58] Gábor Szárnyas, József Marton, János Maginecz, and Dániel Varró. 2018. Reducing Property Graph Queries to Relational Algebra for Incremental View Maintenance. *ArXiv* abs/1806.07344 (2018). https://api.semanticscholar.org/CorpusID:49313167

[59] Yuanyuan Tian, En Liang Xu, Wei Zhao, Mir Hamid Pirahesh, Suijun Tong, Wen Sun, Thomas Kolanko, Md. Shahidul Haque Apu, and Huijuan Peng. 2020. IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 345–359. https://doi.org/10.1145/3318464.3386138

[60] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter A. Boncz. 2012. Heuristics-based query optimisation for SPARQL. In *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, Elke A. Rundensteiner, Volker Markl, Ioana Manolescu, Sihem Amer-Yahia, Felix Naumann, and Ismail Ari (Eds.). ACM, 324–335. https://doi.org/10.1145/2247596.2247635

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

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

[63] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. 2021. Huge: An efficient and scalable subgraph enumeration system. In *Proceedings of the 2021 International Conference on Management of Data*. 2049–2062.

[64] Qian Zhu, Jianhua Yao, Shengang Yuan, Feng Li, Haifeng Chen, Wei Cai, and Quan Liao. 2005. Superstructure Searching Algorithm for Generic Reaction Retrieval. *J. Chem. Inf. Model.* 45, 5 (2005), 1214–1222. https://doi.org/10.1021/CI0496402