Enhancing Neo4j Query Efficiency with Seamless Integration of the GOpt Optimization Framework

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

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

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

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

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

Yongfei Liu
Euler AI
fayer@eulerai.au

ABSTRACT

Graph databases have become increasingly popular for managing and querying complex interconnected data. Neo4j, a leading graph database, offers robust capabilities in processing graph queries. Despite its proficiency, however, Neo4j's default query optimizer can encounter performance bottlenecks due to inherent limitations. In this paper, we explore the integration of Neo4j with GOpt, a graph-native query optimization framework, aiming to overcome the constraints of Neo4j's default optimizer by harnessing GOpt's sophisticated mechanisms, including type inference, extensive optimization rules, and high-order statistic considered cost-based optimizations, with only slight modifications. Our empirical analysis reveals that the integrated system powered by GOpt's optimization strategies, achieves substantial performance improvements, boosting query execution efficiency by up to 19× compared to Neo4j's default optimizer.

VLDB Workshop Reference Format:

Bingqing Lyu, Xiaoli Zhou, Longbin Lai, Yufan Yang, Yunkai Lou, and Yongfei Liu. Enhancing Neo4j Query Efficiency with Seamless Integration of the GOpt Optimization Framework. VLDB 2024 Workshop: 3rd International Workshop on Large-Scale Graph Data Analytics (LSGDA 2024).

1 INTRODUCTION

In the realm of data management, graph databases [2, 4, 6, 11, 19] have proven to be a crucial instrument for effectively capturing complex relationships in various domains. In graph databases, graph pattern matching is a widely-studied problem [7, 8, 17, 21, 22, 29–31], which aims at identifying all subgraphs in a data graph that matches the given query pattern. It has been widely used in real applications, such as social network analysis[14, 18], bioinformatics [13, 27], and chemistry[16, 32].

As one of the most widely-used graph databases, Neo4j [4] enables complex graph queries through the declarative query language Cypher and has been widely adopted in both academia and industry due to its robust query processing capabilities. However, we found that Neo4j encounters some certain limitations particularly in the realm of pattern optimization.

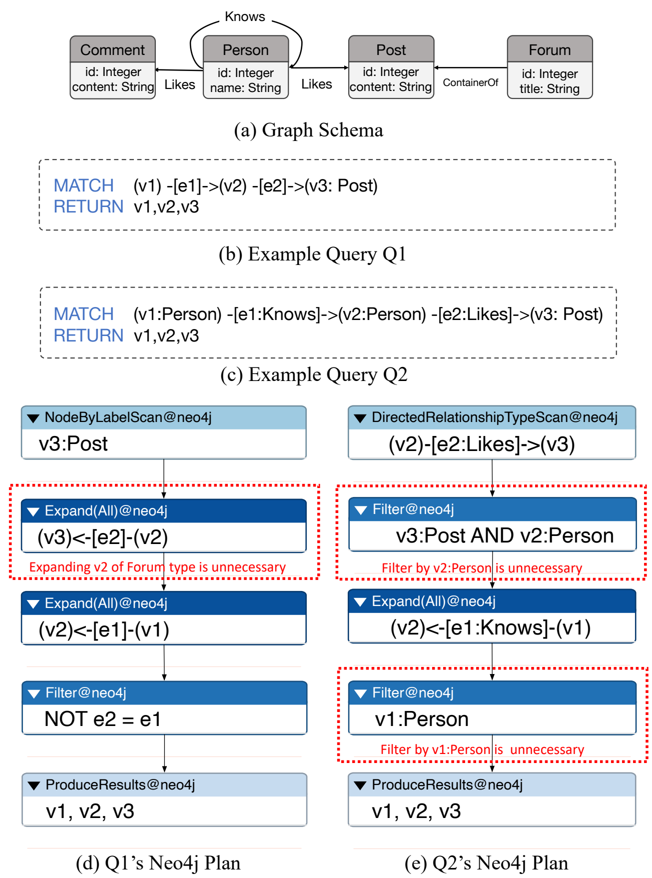


Figure 1: A Motivation Example

Example 1.1. We show an example in Fig. 1. Given a graph database with the schema shown in Fig. 1(a), we aim to identify a 2-hop traversal targeting to a `Post`, as illustrated in Fig. 1(b) and (c). The optimized execution plans generated by Neo4j for the queries Q_1 and Q_2 are shown in Fig. 1(d) and (e), respectively. From Q_1's execution plan, we observe that Neo4j only applies type constraint on v3, as specified in the query, while treating the other vertices and edges as with arbitrary types. For example, Fig. 1(d) shows

Proceedings of the VLDB Endowment. ISSN 2150-8097.

that Neo4j first scans all `Post` vertices as `v3`, and then expands the `(v3)<-[e2]-(v2)`, which matches all the vertices that target to `v3` as `v2`, and the matched vertices can be either a `Person` or a `Forum`. However, the `Forum` vertices are not necessary to be expanded. This is because the query further searches vertices `v1` targeting to `v2`, while no vertex types targeting to `Forum` in the schema. Thus, the `Forum` vertices that matches `v2` can be pruned during the query optimization. For Q_2, we specified the type constraints for all vertices and edges to avoid the redundant computations as in Q_1. In this case, we observe that Neo4j always applies filters for the specified type constraints, while some of them are not necessary. For example, Fig. 1(e) shows that Neo4j first scans `(v2)-[e2:Likes]->(v3)` and then confirms the types by a filter step on `v3` as `Post` and `v2` as `Person`. However, according to the graph schema, the type filter on `v2` is not necessary, since after confirming `v3` as `Post`, the `v2` must be a `Person` as it is the only type that targets to `Post` in the schema. Thus the filter on `v2` (and `v1` similarly) can be pruned during the query optimization.

Drawbacks in Neo4j and Our Solution The above example highlights two principal shortcomings in Neo4j: (1) The lack of type inference. Neo4j requires users to explicitly define type constraints. In the absence of such specifications, it defaults to match all types, thereby incurring unnecessary computational costs due to the implicit type constraints. (2) Inefficient execution plans. The absence of certain optimization techniques in Neo4j can lead to suboptimal execution plans. Additionally, the system does not provide mechanisms for users to register custom optimization rules, restricting opportunities for enhancing the optimization process.

To address the limitations and harness the opportunities for improved performance in Neo4j, this paper explores the integration of GOpt [24], a state-of-the-art graph-native optimization framework, into Neo4j. Specifically, GOpt provides sophisticated optimization techniques, including: (1) GOpt introduces an efficient type inference algorithm that deduces implicit type constraints restricting by the graph schema in the query pattern, thereby enhancing query processing. (2) GOpt comes equipped with an extensive set of heuristic rules, broadening the scope and effectiveness of query optimization. Besides, it employs cost-based optimization techniques, particularly adept at handling pattern matching scenarios involving arbitrary types. (3) GOpt provides user-friendly interfaces for seamless integration, allowing users to define their own custom rules and cost models, thereby improving system adaptability and allowing for tailored optimization.

Contributions. The main contributions of this paper are summarized as follows:

(1) We introduce an integrated system that seamlessly incorporates GOpt into Neo4j, enhancing it by replacing the native Neo4j query optimizer with the sophisticated capabilities of GOpt.

(2) During the integration process, we have made some adaptations to fully harness the optimization capabilities of GOpt within Neo4j. Firstly, we aligned Neo4j's schema to match the specifications of GOpt, which is crucial for the type inference algorithm to work properly. Secondly, we crafted new optimization rules specifically tailored for Neo4j. Lastly, we refined the cost model to more accurately reflect the actual costs of Neo4j's operator implementations, thereby enhancing the precision of cost estimations in the cost-based optimization process.

(3) The empirical results reveal that with the integration of GOpt, the system exhibits performance improvements over Neo4j with its default optimizer by up to 19× in query execution time, demonstrating the impact of integrating GOpt coupled with our specialized optimizations in boosting the efficiency of graph query processing within the Neo4j System.

We organize the paper as follows. We first briefly review Neo4j and GOpt in Section 2. Then we delve into the details of the integration of GOpt into Neo4j in Section 3. We present the experimental results in Section 4. Finally, we discuss related works in Section 5 and conclude the paper in Section 6.

2 PRELIMINARIES

In this section, we briefly review Neo4j and GOpt, and discuss the challenges in the integration of GOpt into Neo4j.

2.1 Neo4j

Neo4j [4] stands as one of the most widely-used graph databases. It adopts the property graph model [9], where the data is represented as a typed, directed graph. The graph is composed of vertices that model entities and edges that represent the relationships connecting these entities. Each vertex and edge is associated with a type to indicate the category of the entity or relationship. For example, as shown in Fig. 1(a), vertices might be of types `Person`, `Comment`, or `Post`, and the edge type such as `Likes` could illustrate the relationship from a `Person` to either a `Post` or a `Comment`. The vertices and edges can have properties, which are key-value pairs.

To facilitate interaction with the graph data, Neo4j supports the Cypher query language [15]. Cypher is known for its declarative style, enabling users to perform complex graph queries in a concise and intuitive way. As an example, Figure Fig. 1(b) presents a Cypher query performing a 2-hop traversal that concludes at a vertex constrained to the type `Post`.

However, Neo4j encounters certain limitations particularly in the realm of pattern optimization. First, Neo4j lacks type inference capabilities and operates strictly according to the user-specified type constraints. Without these constraints, Neo4j will scan the entire graph, even if only a small portion of them are valid candidates against the underlying graph schema. This will not only lead to a substantial misestimate of the cost for the pattern in the optimization stage, but also takes unnecessary overhead computations in execution. Secondly, it may miss optimization opportunities due to a restricted set of optimization rules and the lack of user interfaces to define custom rules, which hinders users from extending Neo4j's optimization capabilities to meet specific optimization requirements. Thirdly, it optimizes patterns relying on low-order statistics, i.e., the number of vertices and edges of each type, and estimate the frequency of patterns based on the assumption that edges exist independently of one another. This assumption falls short when applied to real-world graphs, which can lead to inaccurate cost estimations and inefficient query execution plans. These challenges emphasize the need for enhancements in Neo4j's pattern optimization to increase the overall performance of the system.

2.2 GOpt

GOpt [24] is a graph-native query optimization framework designed to optimize complex graph queries. It is built on top of the property graph model and is compatible with various graph query languages. Specifically, GOpt defines a unified intermediate representation (IR) that includes both graph and relational operations, which, not only provides a common ground for the support of different query languages such as Cypher [15] and Gremlin [28], but also streamlines the application of optimization techniques. Based on this unified IR, GOpt proposes three key optimization techniques to improve the query performance. Firstly, GOpt offers the capability of automatic type inference. The basic idea is to infer the type constraints for the vertices and edges in the query pattern, based on the graph schema and the explicitly specified type constraints. If during the process, no valid type constraints can be further assigned, the type checker will return an INVALID flag. Secondly, GOpt incorporates a wide-ranging suite of optimization rules that explore the optimization possibilities among graph operators, relational operators, and between the two. For instance, it includes FilterIntoMatchRule for pushing down filters, FieldTrimRule for removing redundant intermediate results, and ExpandGetVFusionRule to fuse edge expansion and vertex retrieval operators together. Thirdly, GOpt utilizes sophisticated high-order statistics that capture the frequency of small patterns in the data graph for more precise cardinality estimation. In addition, it considers UnionTypes, the type constraints representing a set of possible types such as `{Person|Forum}`, through refined cost estimation methods. These capabilities contribute to more accurate cost estimations and, consequently, the generation of more efficient query execution plans.

In addition to the optimization techniques, GOpt also provides user-friendly interfaces for seamless integration. Firstly, it offers multiple layers of integration, including support for both Gremlin SDK and Cypher SDK, among others, for end-users to interact with the system. Furthermore, it provides IR Builder SDK which is query language independent, enabling the construction of the IR-based plan for the further application of GOpt's core optimization techniques. Secondly, it provides extensible rule-based optimization (RBO) and cost-based optimization (CBO) modules. Specifically, the RBO module enables users to introduce their own custom rules, thus enhancing the system's adaptability. The CBO module permits the registration of user-defined cost models. This capability is crucial for distinguishing the costs of physical operators implemented in the systems, enabling more accurate cost estimation.

Challenges. The integration of GOpt can bring significant benefits to Neo4j, but it also faces some challenges. Firstly, the schema-free nature of Neo4j permits users to load data and perform queries without the prerequisite of a defined schema. While this adds flexibility, it constrains the viability of applying GOpt's type inference feature, which relies on a well-defined graph schema. Secondly, despite both GOpt and Neo4j adhering to the property graph model, there are nuanced variances in their implementations. Neo4j represents edge types with a single identifier, such as `Likes` which may target to multiple vertex types, whereas GOpt employs a triplet format to specify edge types, such as `Person-Likes->Comment`. Bridging this structural disparity is vital for successful integration. Thirdly, there are inherent dissimilarities between the operator implementations of Neo4j and those assumed in GOpt. These variation in implementations can lead to divergent cost estimations, which must be accounted for during the integration process. Lastly, it is essential to ensure that the physical plan optimized by GOpt is appropriately translated into an executable plan that is compatible with Neo4j's execution engine.

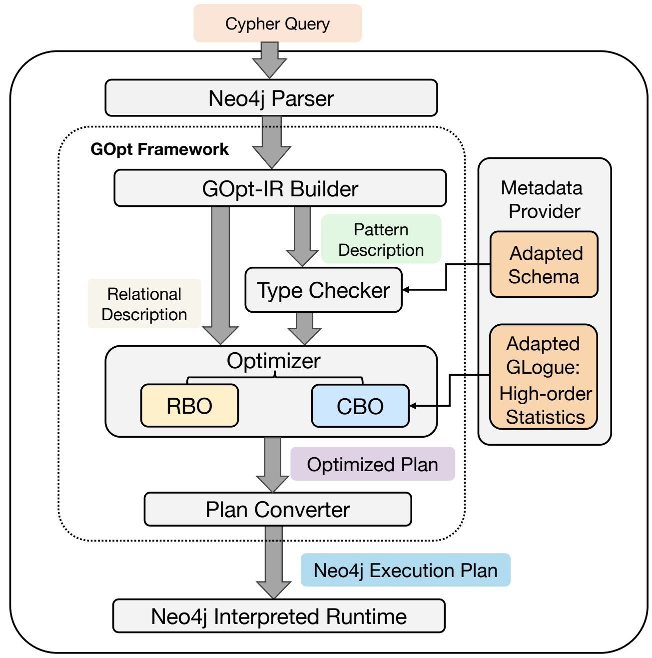


Figure 2: System Architecture of Integration of GOpt with Neo4j.

3 GOPT ON NEO4J

In this section, we delve into the integration of GOpt with Neo4j, and outline the strategic modifications and extensions applied.

3.1 System Overview

Fig. 2 illustrates the system architecture of the integrated system. The architecture holistically encompasses the initial parsing of queries using the native Neo4j Cypher parser, followed by the advanced optimization phase managed by GOpt framework, and ultimately, the execution facilitated by Neo4j's native interpreted runtime engine. Specifically, the query processing pipeline consists of four key steps:

(1) Query Parsing and Transformation. The query is first parsed by Neo4j's Cypher parser to generate the query AST, which is then transformed into GOpt's intermediate representation (IR) using the IR Builder SDK provided by GOpt.

(2) Type Inference. The IR-based logical plan then undergoes a type inference step where GOpt's native type checker infers precise type constraints for the query pattern. This step works with the adapted schema in Neo4j.

(3) Query Optimization. With the inferred type constraints, the logical plan is passed through GOpt's optimizer. This phase leverages both rule-based and cost-based optimization methods to refine the query plan.

(4) Plan Conversion and Execution. Finally, the optimized plan is converted into a format compatible with Neo4j's execution plan, and the converted plan is executed in Neo4j's native interpreted runtime based engine.

The subsequent sections will delve into a detailed discussion of each step in the query processing pipeline.

3.2 Query Parsing and Transformation

To ensure seamless integration with Neo4j's established support for the Cypher query language, we opt to use the IR Builder SDK provided by the GOpt framework instead of interfacing directly with the Cypher SDK within GOpt. By doing so, we leverage Neo4j's own parser to deconstruct the Cypher query into an abstract syntax tree (AST) and then employs the IR Builder SDK to systematically construct the intermediate representation (IR)-based plan that adheres to GOpt's specifications.

Specifically, by utilizing the IR Builder SDK, we can build an IR-based logical plan that includes graph operators, such as `SCAN` for scanning vertices or edges, `EXPAND_EDGE` and `EXPAND_PATH` to find connected edges or paths, `GET_VERTEX` to retrieve endpoint vertices, and `MATCH_PATTERN` as a composed operator to match complex patterns. It also includes relational operators, such as `PROJECT` for data projection and `SELECT` for filtering. Through these operators, the system is able to bridge the expressiveness of the Cypher query language with the powerful optimization capabilities of the GOpt framework, fulfilling both the relational-semantic and graph-semantic requirements of complex query processing.

3.3 Type Inference

Upon constructing the IR-based logical plan, we proceed with the type inference step, where GOpt's type checker will deduce the most appropriate type constraints for the query pattern. This process, however, encounters certain challenges: First, in Neo4j, the schema is not strictly enforced, while the type inference algorithm requires a complete schema. Secondly, Neo4j represents the edge type as a single identifier such as `Knows` and `Likes`, which may not be sufficient for the type inference algorithm. To overcome these hurdles, we make specific schema adaptations in Neo4j to provide more detailed information. Firstly, we employ the APOC procedures [1] provided by Neo4j to extract the graph schema information. Notice that we adopt the extraction method that guarantee the completeness of the schema, thereby ensuring the correctness of the type inference algorithm. Secondly, we adapt the schema in Neo4j to provide more detailed information about the edge type, which is modeled as a triplet of the source vertex type, the edge type, and the destination vertex type. Specifically, if the edge type starts from (or targeting to) multiple vertex types, we define a union of the vertex types as the source (or destination) type in the edge type. For example, the edge type `Knows` in Fig. 1(a) is defined as `Person-Knows-Person`, and the edge type `Likes` is defined as `Person-Likes->Comment|Post`, with `"->"` symbolizing the edge direction, and `"|"` denoting the union of types indicating that any of them is valid.

This adapted schema then allows the type inference algorithm to operate effectively. For example, consider query Q_1 in Fig. 1(b). GOpt's type checker will initiate the process with `Post` v_3. It infers that v_2's type constraint could be `Person|Forum`, based on `Forum-ContainerOf->Post` and `Person-Likes->Comment|Post` in the adapted schema. The type checker then moves on to evaluate the neighbors of v_2. When it comes to v_1, which is an in-neighbor of v_2, and knowing that `Person|Forum`'s in-neighbors in the graph schema are exclusively of type `Person` via edge type `Person-Knows-Person`, the type checker infers that v_1 must be a `Person`. This inference further refines the constraint for v_2, solidifying it as a `Person`.

3.4 Query Optimization

With the inferred type constraints, the logical plan is then passed through GOpt's optimizer to refine the query plan. The optimization process is divided into two main phases: rule-based optimization (RBO) and cost-based optimization (CBO).

3.4.1 Rule-Based Optimization. GOpt comes equipped with a comprehensive suite of optimization rules that considers both graph and relational operators, as well as the interactions between the two. For example, the FilterIntoMatchRule aims to push filters into the graph operators as early as possible. The early application of filters serves to minimize the volume of intermediate results, thereby improving the overall query performance.

Nevertheless, we have identified opportunities for additional optimizations in the integration, especially in how type constraints are handled within queries. By default, Neo4j's native planner applies type filters whenever they are specified, but some filters are unnecessary and can slow down query execution.

Considering the graph schema in Fig. 1(a) and a simple query `(v1:Person)-[e1:KNOWS]-(v2:Person)`, Neo4j starts searching from a `Person` v_1, going through the `Knows` relationship to v_2, and imposes a filter to ensure v_2 is a `Person`, yet this is redundant since v_2 would inherently be a `Person` due to the schema. However, blindly removing all type filters is not ideal. Consider the pattern `(v3:Person)-[e2:LIKES]-(v4:Comment)`. Here, when expanding v_4 from v_3, filtering is essential because v_4 could represent either a `Comment` or a `Post` based on the schema. This optimization consideration lies in Neo4j's original schema design, which allows a single relationship type to connect either a single vertex type or a union of vertex types. To accommodate this, we design a new optimization rule to remove the unnecessary type filters while guaranteeing the correctness of the query.

TypeFilterRemovalRule. In the query pattern, when dealing with an edge expansion from source vertex s to target vertex t, the type filter on t can be safely disregarded if the possible types of t as defined in the schema (which could be a union of different types) are enclosed within the inferred types for t.

For example, consider query Q_2 in Fig. 1(c), where we start looking from vertex v_1 through edge e_1 to v_2, then through edge e_2 to v_3. v_2 is inferred to be a `Person` and the schema confirms this, thus we remove the type filter on v_2. Conversely, while further expanding e_2, the target type of v_3 in schema is `Comment|Post`, and the inferred type for v_3 is solely Comment, thus we must keep the type filter. Here, as we consider both user-given type constraints and graph schema in the type checker, the inferred types are exact what users give in this example. Note that TypeFilterRemovalRule take the expanding direction into consideration. For instance, if we traverse edge e_2 from v_3 to v_2, the destination type is Person based on the Person-Likes->Post|Comment edge in schema in the in-direction, which matches the inferred type for v_2, and thus the filter on v_2 can be removed.

We develop this new rule by implementing the interfaces defined in GOpt's RBO module, which requires to (1) define the condition for applying the rule and (2) specify the transformation to be applied. Then we register the rule in the RBO module, which will be automatically applied during the optimization process.

3.4.2 Cost-Based Optimization. In the integration, we leverage the advanced cost-based optimization techniques in GOpt with minor modifications to fit the Neo4j's execution engine, including the high-order statistics and cost model.

High-Order Statistics. Unlike the default Neo4j planner that uses low-order statistics for its optimizations, GOpt employs higher-order statistics to achieve more accurate cardinality estimates. Specifically, GOpt enumerates small patterns (a.k.a. motifs) up to a certain size based on the graph schema, and precomputed their occurrences to serve as the higher-order statistics. However, as the schema in Neo4j is adapted to include the union of vertex types in edge types, the motif enumeration process needs to be adjusted to accommodate this schema. For example, when dealing with edge type like Person-Likes->Comment|Post, we enumerate and count not just the patterns contains Person-Likes->Comment and Person-Likes->Post, but also the Person-Likes->Comment|Post edge. This enables the integrated system to directly retrieve statistics for queries involving such type constraints, such as Match (v1)-[e1:LIKES]-(v2), making the cost estimation more precise.

Cost Model. We also accommodate the cost model in GOpt with modifications to align with Neo4j's execution engine. GOpt adopts a generic cost model that calculates query plan costs by accounting for both communication cost, based on the number of intermediate results, and computation cost, defined as the summarized cost of the operators in the physical plan with a normalized factor α_{op} to reflect the weight of each operator to differentiate processing expenses. However, since Neo4j's operator implementations may differ from those assumed by GOpt's cost model, we tailor the normalized factors α_{op} to better match Neo4j's actual performance. For instance, to search a triangle pattern (v1)->(v2)->(v3)->(v1), GOpt may generate a candidate physical plan with EXPAND_EDGE(v_1, v_2), indicating the expansion from v_1 to v_2, and then a VertexExpandOpr , which consists of a EXPAND_EDGE(v_2, v_3) and a EXPAND_EDGE(v_3, v_1), which, executes with a worst-case optimal join [8] by intersect the results of the two expansions, as assumed in the original GOpt cost model. However, Neo4j does not support the worst-case optimal join implementation. Instead, It executes the VertexExpandOpr by first expanding edges from v_2 to v_3, and then checking if the there exists an edge from v_3 to v_1 by an EXPAND_INTO operation. Recognizing that EXPAND_INTO is less efficient than the worst-case optimal join with micro benchmarks, GOpt adjusts the α_{op} value in the cost model to reflect this difference. By updating the cost model, GOpt ensures that the optimal plan generated by the CBO module accurately reflects Neo4j's performance characteristics, to produce more efficient execution plans.

3.5 Plan Conversion and Execution

We use Neo4j's interpreted runtime as the backend execution engine due to its flexibility and ease of integration. After the optimization strategies are applied, GOpt translates the optimized physical plan into a form recognizable by Neo4j's execution engine. In many cases, there is a one-to-one mapping where operators like PROJECT and SELECT in GOpt correspond to Project and Filter in Neo4j, respectively. For more complex operators, such as VertexExpandOpr, a sequence of Expand operations are generated, and when it forms a cyclic pattern, the last Expand would be replaced by an EXPAND_INTO. Once the conversion is complete, the resultant execution plan can be run on Neo4j's engine, leveraging the optimizations made by GOpt to enhance the query performance.

4 EXPERIMENTS

We conducted a series of experiments to compare the performance the integrated system that substitutes Neo4j's native optimizer with GOpt, against Neo4j with its native optimizer. As constrained by the open-source version of Neo4j, which only supports standalone deployment, our experiments were conducted on a single machine.

4.1 Experiment Settings

Hardware. Our experiments are conducted on a machine equipped with dual 8-core Intel Xeon E5-2620 v4 CPUs (each featuring 8 cores and a 2.1GHz clock speed), 512GB of memory, and a 1TB disk. We utilize the native Neo4j graph database for data storage, ensuring all data is loaded into memory to eliminate extra I/O overhead. The number of threads used corresponds to the number of available cores, adhering to the default Neo4j configuration.

Datasets. We employ LDBC Social Network Benchmark (SNB) [23] dataset as our primary data source. The LDBC SNB dataset is a synthetic dataset that simulates a social network comprising various entities and relationships. The dataset is generated using the official LDBC SNB Data Generator.In the experiments, we use the default scale factor of 1, resulting in a dataset with 3 million nodes and 17 million relationships, encompassing 8 types of nodes and 15 types of relationships, to allow Neo4j processing all queries in a reasonable time.

Queries. We design three distinct groups of queries labeled as "Q_t.", "Q_r.", and "Q_c.", and introduced a pattern matching query set called the Labelled Subgraph Query Benchmark (LSQB) [3], labeled as "Q_{ls}.". The details are as follows:

(1) $Q_t[1, \ldots, 5]$: This group aims to verify the type inference capability of GOpt. Accurate type inference is a critical feature of GOpt's core design, enabling the generation of more precise logical plans and significantly reduce the intermediate data amount during query execution.

(2) $Q_r[1, \ldots, 4]$: GOpt incorporates a set of heuristic rules in to optimize the query, such as FilterIntoMatchRule, which pushes down filters into the match clause, and the TypeFilterRemovalRule, which trims unnecessary label filtering when fetching adjacent

vertices from edge data. $Q_r[1]$ and $Q_r[2]$ test the effectiveness of FilterIntoMatchRule, while $Q_r[3]$ and $Q_r[4]$ test the effectiveness of TypeFilterRemovalRule.

(3) $Q_c[1, \ldots, 4(a|b)]$: This group emphasizes the optimization effect of GOpt on complex patterns. We design four commonly used graph patterns: triangles ($Q_c[1a]$ and $Q_c[1b]$), squares ($Q_c[2a]$ and $Q_c[2b]$), 5-paths ($Q_c[3a]$ and $Q_c[3b]$), and 4-cliques ($Q_c[4a]$ and $Q_c[4b]$). The "a" and "b" variants denote the exclusion and inclusion of union types, respectively, further validating the optimization across varying type constraints.

(4) $Q_{ls}[1, \ldots, 6]$: For a comprehensive evaluation, we further introduce a set of standard LSQB queries, adhering to the LDBC schema standard but focusing on simulating complex query patterns in real-world scenarios. We select 6 out of the original 9 query sets ($Q_{ls}[1, ..., 6]$), omitting $Q_{ls}[7, 8, 9]$ due to their reliance on optional/anti-edges, which are not the primary focus.

The queries are in Cypher and can be found in [5].

4.2 Experiment Results

In this section, we evaluate the performance of GOpt on Neo4j, focusing on type inference, rule-based optimization (RBO), and cost-based optimization (CBO). We integrated GOpt into Neo4j and added configuration options to the original Neo4j configuration file. These options fall into two main categories:

(1) Optimization Control: These settings control whether the optimization rules in GOpt are enabled, allowing us to assess the performance improvements brought by each optimization rule in GOpt.

(2) Optimizer Switching: These settings allow us to switch between GOpt and Neo4j's native optimizer. This enables a direct comparison of execution plans and performance metrics between the two optimizers, testing the improvements achieved by GOpt.

Type Inference. We conducted experiments to evaluate the impact of type inference optimization on query performance by enabling and disabling this feature. The results, shown in Fig. 3(a), illustrate that Type Inference significantly reduces latency for most queries, with an average decrease of 5×. Notably, query $Q_t[4]$ shows particularly significant improvement, with latency reduced by 17×. This query involves the most complex pattern among all queries, forming a triangle. Type Inference optimization has a more pronounced impact on complex queries due to several factors: the computation of complex patterns requires operators with higher complexity, which are more inclined to be influenced by the volume of data. In Neo4j, for cyclic patterns, operators like `ExpandInto`/`Join` are crucial, and their performance degrades significantly with increased intermediate data volume. Type inference optimization effectively mitigates this issue by eliminating unnecessary intermediate data, thereby enhancing overall query performance. Please note that performance improvements may vary based on specific queries with the optimization enabled or disabled, as different query patterns have varying complexities and involve differing numbers of intermediate results. Similar situations may occur in the following experiments.

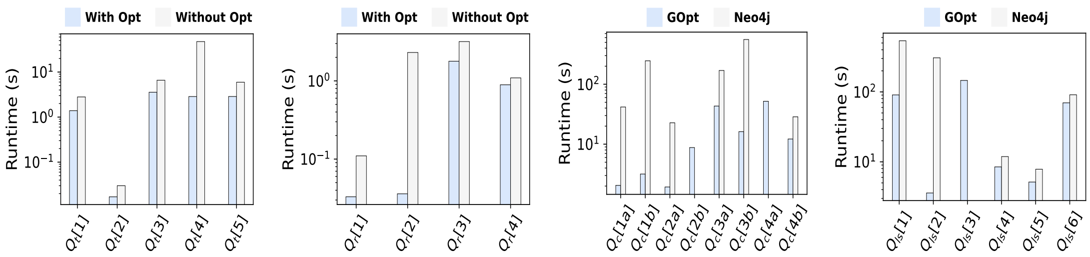
Heuristic Rules. We assessed the effectiveness of two heuristic rules in GOpt: FilterIntoMatchRule and TypeFilterRemovalRule. We primarily evaluated the impact of these rules on query performance by enabling and disabling them. The results, presented in Fig. 3(b), illustrate that these rules significantly enhance query performance. $Q_r[1]$ and $Q_r[2]$ show the most substantial improvements, with latency reduced by 33× on average. This improvement can be attributed to the intuitive and impactful optimization effects of FilterIntoMatchRule, which operates in two main aspects: (1) It optimizes by pushing conditions down into the Match clause, thereby reducing intermediate data generated by Match. (2) Within the Match clause, RBO prioritizes starting from nodes with conditions, minimizing unnecessary graph traversal in the pattern. $Q_r[3]$ and $Q_r[4]$ demonstrate the optimization effects of TypeFilterRemovalRule, reducing latency by nearly 2×. TypeFilterRemovalRule optimizes edge data retrieval by avoiding redundant label filtering when the source node and edge uniquely determine the target label type, thereby reducing unnecessary computational overhead.

Cost-based Optimization. In this section, we compare the performance improvements of GOpt against the Neo4j native optimizer on various queries. We use two different query sets: Q_c and Q_{ls}. Fig. 3(c) and Fig. 3(d) illustrate the performance results for these two sets, respectively. It is evident that GOpt significantly outperforms the Neo4j's native optimizer, with an average performance improvement of 19× on Q_c and 16× on Q_{ls}. Notably, for queries $Q_c[2b]$, $Q_c[4a]$, and $Q_{ls}[3]$, the plan generated by Neo4j's native optimizer runs out of memory (OOM) due to excessive intermediate data generation. This performance gain can be attributed to three key factors:

(1) Type Inference Optimization: GOpt's Type Inference accurately infers type information within patterns, leading to more precise cardinality estimations and better execution plans. Without accurate type information, Neo4j's native optimizer defaults to Rule-Based Optimization (RBO), which tends to prioritize nodes or edges with explicit type constraints. This order may result in excessive intermediate data generation, negatively impacting query performance. For instance, in query $Q_c[2a]$, where the `Person` type is missing, the Neo4j native optimizer prefers starting from other node types, producing 9× more data than the optimal execution order and resulting in a 12× increase in latency. However, in some cases, Neo4j's RBO optimizer coincidentally finds an execution order close to the optimal one. This happens when starting from nodes with explicit type constraints aligns with the optimal execution order. Even in such cases, GOpt achieves an average improvement of 1.5× (e.g., $Q_c[4b]$, $Q_{ls}[4]$, $Q_{ls}[5]$, $Q_{ls}[6]$), due to the following two factors;

(2) Fine-grained Cost Estimation: Drawing from traditional relational optimization frameworks, GOpt implements specific cost estimation methods for each physical execution operator. For certain graph traversal operators (e.g., `source`, `expand`), it considers the impact of the graph database interface on cost estimation. This fine-grained approach allows GOpt to more accurately reflect actual physical execution, thus improving the accuracy of query optimization.

(3) Hybrid Join Strategies: GOpt supports hybrid join strategies, optimizing for complex graph patterns. For example, in query $Q_{ls}[3]$, which consists of 7 nodes and 9 edges, forming multiple



(a) Type Inference (b) Heuristic Rules (c) CBO (d) LSQB Queries

Figure 3: Experimental Results.

cyclic structures in the pattern, GOpt can produce an optimal execution plan consists of VertexExpandOpr (which in the integration would be translated to Neo4j's `ExpandInto` operator), and significantly enhancing query performance. In contrast, Neo4j's native optimizer uses a traditional relational approach, converting the pattern into multiple join operations. As join nesting increases, the intermediate data volume grows, eventually leading to OOM errors.

5 RELATED WORKS

Graph Query Optimization. Graph pattern matching, a key task in graph database queries, has been extensively studied with the goal of optimizing this process. The traditional standard for this task is subgraph isomorphism, a concept rooted in Ullmann's algorithm introduced in [30]. This algorithm inspired many other improvements, such as methods for indexing [29], breaking symmetries to speed up the search [17], and compressing graph structures [12]. In distributed systems, researchers have shifted from backtracking approaches to join-based algorithms, which are better suited for parallel processing. These newer algorithms focus on predicting the cost of different join orders to match patterns effectively, using either binary-join methods informed by random modeling [20, 21], or by employing worst-case-optimal joins that keep the cost within a theoretical upper limit [8, 26]. However, no single method has consistently proved to be the best, leading to the development of hybrid algorithms that combine binary and worst-case-optimal joins, choosing the most cost-effective option for each scenario [7, 25, 31]. To further enhance the calculation of costs, some researchers recommend using detailed pattern frequency data to inform the planning of graph pattern queries [22, 25]. Yet, while these research advancements have improved graph pattern matching, they still don't fully accommodate the practical query needs that often include complex relational operations alongside graph pattern matching.

Graph Databases. Graph databases and systems have garnered significant attention in recent years as they provide flexible, efficient ways to store and query complex data that naturally maps onto a graph structure. Neo4j[4] is one of the most popular open-source NoSQL graph database systems. It employs native graph storage and processing capabilities, and provide a powerful query language called Cypher[15] for querying graph data. TigerGraph[6] is a distributed graph database that supports the GSQL query language. It is designed to handle large-scale graph data and complex graph queries. TinkerPop[10] is a traversal-based graph computing framework that provides the Gremlin query language[28]. JanusGraph[2] is a distributed graph database that supports the Gremlin query language. Despite their advancements, these systems have limitations in optimizing graph queries, particularly in the context of graph pattern matching, which is the focus of our work.

6 CONCLUSION

In this paper, we explore the integration of Neo4j with GOpt, a graph-native query optimization framework, to enhance the performance of graph query processing in Neo4j. By embedding GOpt, Neo4j enables the application of sophisticated optimization techniques, such as automatic type inference, a comprehensive set of heuristic rules, and improved cost estimation models, with minimal adjustments to the existing infrastructure. As evidenced by the experiments, GOpt delivers substantial performance enhancements, improving the query execution time by up to 19× compared to Neo4j's default optimizer.

REFERENCES

[1] 2024. https://neo4j.com/labs/apoc/4.4/overview/apoc.meta/.

[2] 2024. JanusGraph: A Transactional Graph Database. https://janusgraph.org/

[3] 2024. Labelled Subgraph Query Benchmark. https://github.com/ldbc/lsqb.

[4] 2024. Neo4j Graph Database. https://neo4j.com/.

[5] 2024. Queries used for the experiments. https://github.com/alibaba/GraphScope/tree/main/interactive_engine/benchmark/queries/cypher_queries/experiments/neo4j

[6] 2024. TigerGraph: Graph Analytics Platform. https://www.tigergraph.com/

[7] 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.

[8] 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

[9] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–40.

[10] Apache TinkerPop. 2024. http://tinkerpop.apache.org/.

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

[12] 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.

[13] 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

[14] 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.

[15] 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.

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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.

[21] 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.

[22] 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

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

[24] Bingqing Lyu, Xiaoli Zhou, Longbin Lai, Yufan Yang, Yunkai Lou, Wenyuan Yu, and Jingren Zhou. 2024. A Graph-Native Query Optimization Framework. arXiv:2401.17786

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

[26] 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.

[27] 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.

[28] 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

[29] 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

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

[31] 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.

[32] 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