Supplementary Material for A Graph-Native Query Optimization Framework

Bingqing Lyu[1], Xiaoli Zhou[1], Longbin Lai[1], Yufan Yang[1], Yunkai Lou[1], Wenyuan Yu[1], Ying Zhang[2], Jingren Zhou[1]

[1]Alibaba Group, [2]Zhejiang Gongshang University

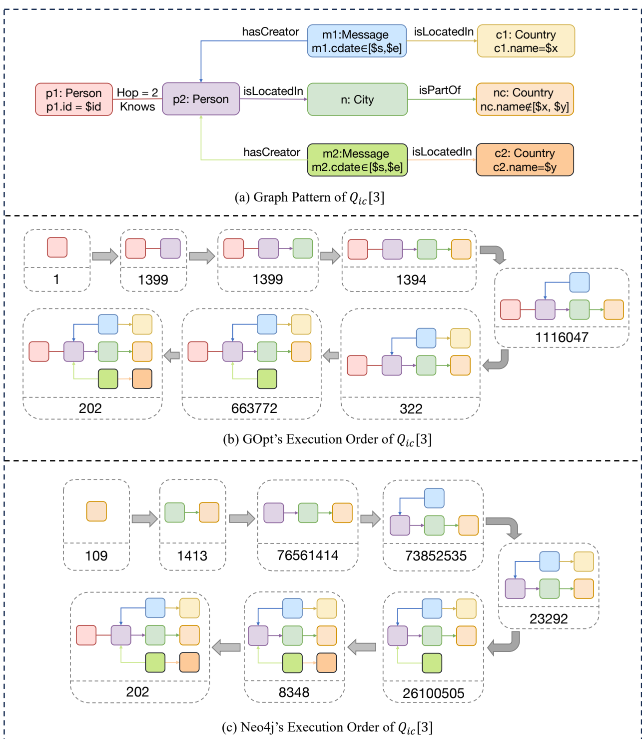
{bingqing.lbq,yihe.zxl,longbin.lailb,xiaofan.yyf,louyunkai.lyk,wenyuan.ywy,jingren.zhou}@alibaba-inc.com

ying.zhang@zjgsu.edu.cn

A APPENDIX

A.1 Queries and Execution Plans

In this section, we delineate the execution plans for queries, emphasizing the proficiency of GOpt in ascertaining the most efficient search order for the query execution. First, we provide detailed case study on the LDBC queries, taking $Q_{Ic}[3]$ as a representative example, to compare the execution plans optimized by GOpt and Neo4j. Then, we present the execution plans for queries $Q_c[1 \dots 4(a|b)]$ which are designed to assess the effectiveness of cost-based optimization techniques.



(a) Graph Pattern of $Q_{ic}[3]$

(b) GOpt's Execution Order of $Q_{ic}[3]$

(c) Neo4j's Execution Order of $Q_{ic}[3]$

Figure 1: Execution Plans for $Q_{ic}[3]$

A.1.1 Case Study on LDBC Queries. During our tests with the LDBC queries, GOpt was observed to produce optimized plans that match the quality of manually optimized ones from prior research [1]. This case study take $Q_{ic}[3]$ as an example to offer an in-depth comparison between the execution plans optimized by GOpt and Neo4j. The query is presented in Fig. 1(a), and the execution order optimized by GOpt and Neo4j are illustrated in Fig. 1(b) and Fig. 1(c), respectively. For example, GOpt starts the searching from `Person` p_1, while Neo4j starts from `Country` nc. We assessed both plans' performance, and illustrate the number of intermediate results produced during the query processing for each plan in Fig. 1, highlighting that GOpt's plan generates merely 1% of the intermediate results compared to Neo4j's, while being around 26× faster as verified in the previous small-scale experiments. From the figure, we can see that the main reason for the performance gap is that Neo4j's plan results in an explosion of intermediate results, especially when expanding from `Person` p_2 (which has $76,561,414$ matchings) to `Message` m_1 (produced $73,852,535$ intermediate results) and m_2 (produced $26,100,505$ intermediate results). In contrast, GOpt expands p_2 from a user-specified starting point p_1, thereby limiting the number of potential matches for p_2 to $1,399$, and effectively minimizing the matching numbers for subsequent expansions to `Message` m_1 (limiting to $1,116,047$) and m_2 (limiting to $663,772$). This case study underscores GOpt's efficacy in optimizing the execution of complex queries.

A.1.2 Execution Plans for Queries. We present the execution plans for queries $Q_c[1 \dots 4(a|b)]$, which have been optimized by GOpt. These plans are illustrated in Fig. 2, with details includes a step-by-step breakdown of the query plan generation, with a focus on the decision-making at each stage. Additionally, we specify the quantity of intermediate results generated throughout the query execution, providing insight into the efficiency and performance implications of the optimization strategies employed by GOpt.

A.2 Intermediate Representation

In this subsection, we provide a more detailed description for the intermediate representation (IR) used by GOpt to capture both graph and relational operations. The IR abstraction defines a data model $\mathcal{D}$ that describes the structure of the intermediate results during query execution, and a set of operators Ω.

The data model $\mathcal{D}$ presents a schema-like structure in which each data field has a name, denoted as a String type, accompanied by a designated datatype. The supported datatypes encompass both graph-specific datatypes and general datatypes. Graph-specific datatypes include *Vertex*, *Edge*, and *Path*, as shown below:

- *Vertex* is a datatype to represent the vertices in data graph. It typically consists of: ID that serves as a unique identifier for the vertex; *type* that characterizes the vertex class; and *properties* that includes property names and property values as a set of attributes associated with the vertex's type.
- *Edge* is a datatype to represent the edges in data graph. It usually includes: EID that acts as a unique identifier for the edge, which

is a triplet that further includes `src_id` and `dst_id` to pinpoint the source and destination vertices; *type* that represents the edge kind, which is also a triplet that further includes `src_type` and `dst_type` to specify the source and destination vertex types; and *properties* that consist of property names and property values as a set of attributes associated with the edge's type.

- *Path* is a datatype of an array of vertices and edges that represents a sequence of connected vertices and edges in the data graph. It is denoted as $p = [v_1, e_1, v_2, e_2, ..., v_n]$, where v_i and e_i are the i-th vertex and edge in the path respectively. Specifically, *Path* includes `PID` as a unique identifier; and a specific property of `length`, denoting the number of edges in the path.

General datatypes comprise *Primitives* including *Integer*, *Float*, *String* etc., and *Collections* representing a group of elements, e.g., *List*, *Set*, and *Map*. Notice that the properties in vertices and edges are of general datatypes. For instance, a vertex with *type* `Person` may have *properties* of `name` (*String*), `age` (*Integer*), and `hobbies` (*List*).

The operators in Ω operate on data tuples extracted from $\mathcal{D}$, and produce a new set of data tuples as a result. The set Ω is composed of graph operators and relational operators. The graph operators are specifically for the retrieval of graph data and include the following:

- `GET_VERTEX` with a 4-tuple $(tag, alias, types, [SRC|TGT])$ parameter obtains the source or target vertices from the tagged edge with the specified type constraints, and output aliased results. If the *tag* is unspecified (NA), it retrieves vertices from data graph.
- `EXPAND_EDGE` with a 4-tuple $(tag, alias, types, [OUT|IN])$ parameter expands out or in edges from the tagged vertices with the specified type constraints, and output results with alias. Similarly, if the *tag* is unspecified, it retrieves edges from data graph.
- `EXPAND_PATH` is designed to expand paths from source vertices. The basic expand operation in `EXPAND_PATH` is a composite of `EXPAND_EDGE` and `GET_VERTEX`, with an additional parameter of the hop number explicitly specified.
- `MATCH_PATTERN` is a composite operator consists of the above three basic graph operators, and is employed to describe a series of operations to match a complex pattern within the data graph. Specifically, we use `MATCH_START` and `MATCH_END` to denote the start and end of a `MATCH_PATTERN`.

Notice the *types* in the graph operators denotes the type constraints, that can be either BasicType, UnionType, or AllType based on query requirements, to filter out desired classes of graph elements. The *alias* in the operators tells the backend to store intermediate results with the given alias for further reference by subsequent operations via its *tag*. We offer a special empty String tag to refer to the result of the immediate previous operation, allowing to avoid saving unnecessary data in execution. We also allow filter conditions fused into the operates, by the optimization rules that will be introduced later. Additionally, in the operator of `EXPAND_PATH`, the hop number is a positive integer, and we will support a range of hops in the future. The other category of operators in Ω are relational operators $\mathcal{R}$, includes `PROJECT`, `SELECT`, `JOIN`, `ORDER`, etc., which are widely used in RDBMS. These operators can be applied on graph-specific data as well, e.g., to project properties of vertices, to select edges with specific conditions, or to join two sub-paths into a longer one with the join key as the end vertices of the two sub-paths.

REFERENCES

[1] 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

Figure 2: Optimized Execution Plans by GOpt for $Q_c[1\ldots4(a|b)]$