Supplementary Material for A Modular Graph-Native Query Optimization Framework

Bingqing Lyu, Xiaoli Zhou, Longbin Lai, Yufan Yang, Yunkai Lou, Wenyuan Yu, Ying Zhang‡, Jingren Zhou
{bingqing.lbq,yihe.zxl,longbin.lailb,xiaofan.yyf,louyunkai.lyk,wenyuan.ywy,jingren.zhou}@alibaba-inc.com
ying.zhang@zjgsu.edu.cn
Alibaba Group, ‡Zhejiang Gongshang University
China

1 APPENDIX

1.1 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 BI_9 as a representative example, to compare the execution plans optimized by GOpt and Neo4j. Then, we present the execution plans for queries $QC_{1\ldots4(a|b)}$ which are designed to assess the effectiveness of cost-based optimization techniques.

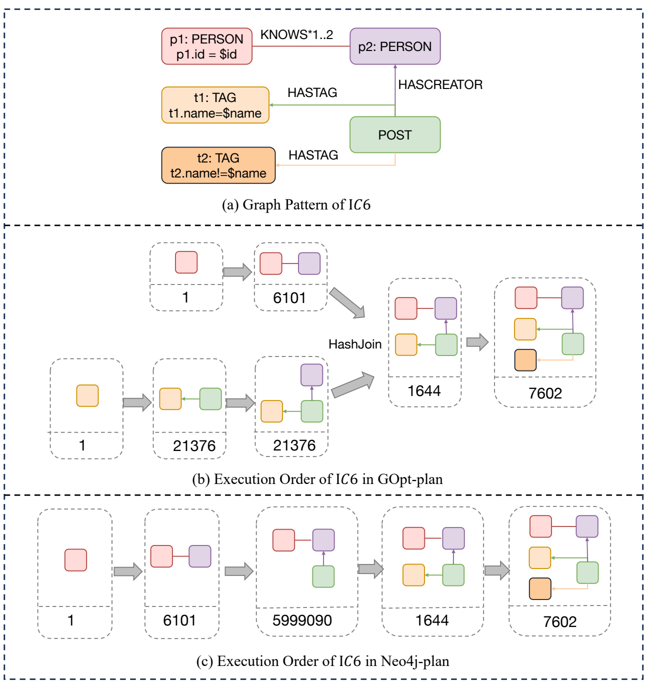

Figure 1: Execution Plans for IC_6

1.1.1 Case Study on LDBC Queries.

During our tests with the LDBC queries, GOpt produced optimized plans comparable in quality to manually optimized plans from previous research [1]. This case study examines IC_6 on G_{30} to provide an in-depth comparison between the execution plans optimized by GOpt and those by Neo4j, as IC_6 demonstrated one of the most significant improvements in efficiency when comparing GOpt-plan and Neo4j-plan in our experiments. The query is shown in Fig. 1(a), with the execution orders of GOpt-plan and Neo4j-plan illustrated in Fig. 1(b) and Fig. 1(c), respectively.

Our analysis reveals that GOpt-plan employs a hybrid join strategy, initiating searching subpatterns from `PERSON` p_1 and `TAG` t_2 simultaneously due to high selectivity filters on these nodes, followed by a `HashJoin` of the results, and further expansion to `TAG` t_2. In contrast, Neo4j-plan employs a sequential traversal, starting from `PERSON` p_1 and expanding to `PERSON` p_2, a `Post`, `TAG` t_1, and finally `TAG` t_2. Fig. 1 illustrates the number of intermediate results generated during query processing for each plan, demonstrating that GOpt-plan produces only 1% of the intermediates compared to Neo4j-plan. The comprehensive experiments also show that GOpt-plan outperforms Neo4j-plan by 48.6× in query execution time on Neo4j, and 78.7× on GraphScope for G_{100}. This case study highlights GOpt's effectiveness in optimizing complex query execution, significantly reducing the number of intermediate results and enhancing query performance.

1.1.2 Execution Plans for Queries.

We present the execution plans for queries $QC_{1\ldots4(a|b)}$, 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.

1.2 Intermediate Representation

In this subsection, we provide a more detailed description for the intermediate representation (GIR) used by GOpt to capture both graph and relational operations. The GIR 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` is designed to retrieve vertices from the data graph. It has a 4-tuple $(tag, alias, types, [SRC|END])$ parameter to obtain 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` is to retrieve edges from the data graph. It has a 4-tuple $(tag, alias, types, [OUT|IN])$ parameter to expand 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 specified source vertices. It has a 5-tuple $(tag, alias, expand\_base, length, opt)$ parameter. Similarly, the tag is used to refer to the source vertices, and the *alias* denotes to the output results. *expand_base* is a composite of `EXPAND_EDGE` and `GET_VERTEX`, defining the specific logic of each hop in the path expansion. The $length$ indicates the number of hops in the path. The *opt* is a path option, which can be "Arbitrary", "Simple", or "Trail", to specify that in the result paths, all vertices and edges can be duplicated, vertices cannot be duplicated, or edge cannot be duplicated, respectively. These path options help to manage and limit the potential for generating unbounded path results.
- `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 in GOpt, i.e., the FilterIntoPattern. 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

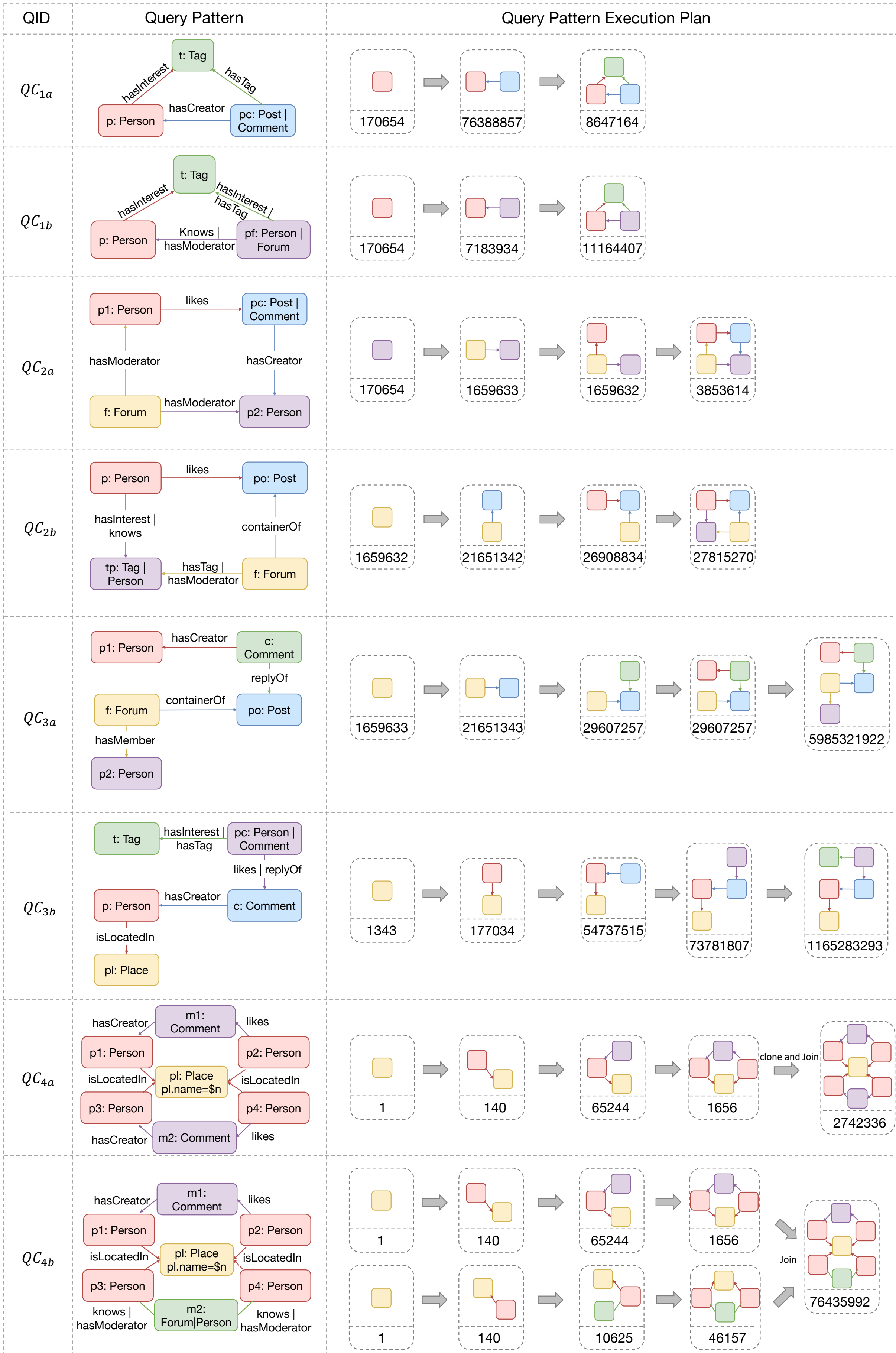

Figure 2: Optimized Execution Plans by GOpt for $Q_c[1 \ldots 4(a|b)]$