A Fast Plan Enumerator for Recursive Queries

Amela Fejza, Pierre Genevès, Nabil Layaïda

A Fast Plan Enumerator for Recursive Queries

Amela Fejza*, Pierre Genevès†, Nabil Layaïda‡
Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
38000 Grenoble, France
Email: *amela.fejza@inria.fr, †pierre.geneves@inria.fr, ‡nabil.layaida@inria.fr,

***Abstract*—Plan enumeration is one of the most crucial components in relational query optimization. We demonstrate RLQDAG, a system implementation of a top-down plan enumerator for the purpose of transforming sets of recursive relational terms efficiently. We describe a complete system of query optimization with parsers and compilers adapted for recursive queries over knowledge and property graphs. We focus on the enumeration component of this sytem, the RLQDAG, and especially on its efficiency in generating plans out of reach of other approaches. We show graphical representations of explored plan spaces for queries on real datasets. We demonstrate the plan enumerator and its benefits in finding more efficient query plans.**

I. Introduction

Labeled graph data structures become increasingly popular in various applications such as social networks, knowledge extraction, transportation networks, biological and clinical data with interactions, etc. [13].

Two main labeled graph data models coexist: *knowledge graphs* in which edges are labeled with a single predicate (RDF graphs), and *property graphs* in which richer annotations are possible on both edges and nodes since they can in addition carry a list of key-value pairs.

In classical relational query optimization, plan enumeration is essential as it defines the boundaries of practical query efficiency. First, the theoretical plan space to be explored depends on the initial query term and the set of rewrite rules considered. Second, enumeration efficiency determines the portion of the theoretical plan space that will be effectively explored. Given a particular enumeration strategy, the greater the portion of the plan space, the more likely it is to contain efficient and practical plans. Optimizing plan enumeration is a well-known and hard topic that seeks to minimize redundant computations when exploring huge plan spaces. There are several ways to enumerate recursion-free plans: bottom-up [14] and top-down [7], [8].

We demonstrate a novel implementation of a top-down system for exploring recursive plan spaces. The system includes a complete implementation of the recursive relational algebra proposed in [9], [10]. It also includes parsers and compilers so that one can formulate, optimize and answer queries that navigate recursively in property graphs. The novelty of the system resides in its ability to efficiently explore recursive plan spaces, with an application for optimizing query answering with recursive path patterns on property graphs.

This research has been partially supported by MIAI@Grenoble Alpes, (ANR-19-P3IA-0003).

The purpose of this demonstration is to show in an interactive manner: (i) how we can explore recursive plan spaces efficiently by grouping sets of terms and applying rewrite rules to a set of terms at a time; and (ii) how this efficient exploration is actually beneficial in finding more efficient plans.

II. Preliminary Background

The RLQDAG introduced in [5] extends the Logical Query Directed Acyclic Graph (LQDAG) with the ability to capture and transform sets of recursive terms. The LQDAG is a directed acyclic graph data structure used to represent and generate the logical plan space in a compact manner, by allowing the sharing of common subparts. It was introduced in [8] and improved in [7]. It is also known as the AND-OR-DAG in [12] where it is used for detecting and unifying common subexpressions for multi-query optimization. It is also used for generating the space of cross-product free join trees in [15]. The LQDAG data structure comprises two distinct node types: equivalence nodes and operation nodes. Equivalence nodes exclusively accommodate operation nodes as their children, while operation nodes can solely exist as children within equivalence nodes. Equivalence nodes are designed to explicitly group together subterms that are equivalent. An operation node corresponds to an algebraic operation like: join, filter, etc. All variants of LQDAG considered so far only support recursion-free queries. To support recursion, one extension brought by the RLQDAG [5] is the introduction of the annotated equivalence node for representing a recursive part. This node is introduced to model the fixpoint operator (that captures recursion) found in recursive relational algebras [2], [3], [9]. The main difference with the classical equivalence node, is that under an annotated equivalence node one can find explicit occurrences of the recursive variable bound by a fixpoint operator.

III. System Architecture

The overall system architecture, shown in Figure 1, is composed of several components:

- **an interface** based on an interactive notebook that uses a scala shell [4] providing features in Jupyter notebooks. The interface exposes an API for interacting with different components of the system. As shown in Figure 1 the interface acts at three different levels: (i) when writing an input query that is parsed and sent to the system; (ii) during the optimization part (when RLQDAG explores

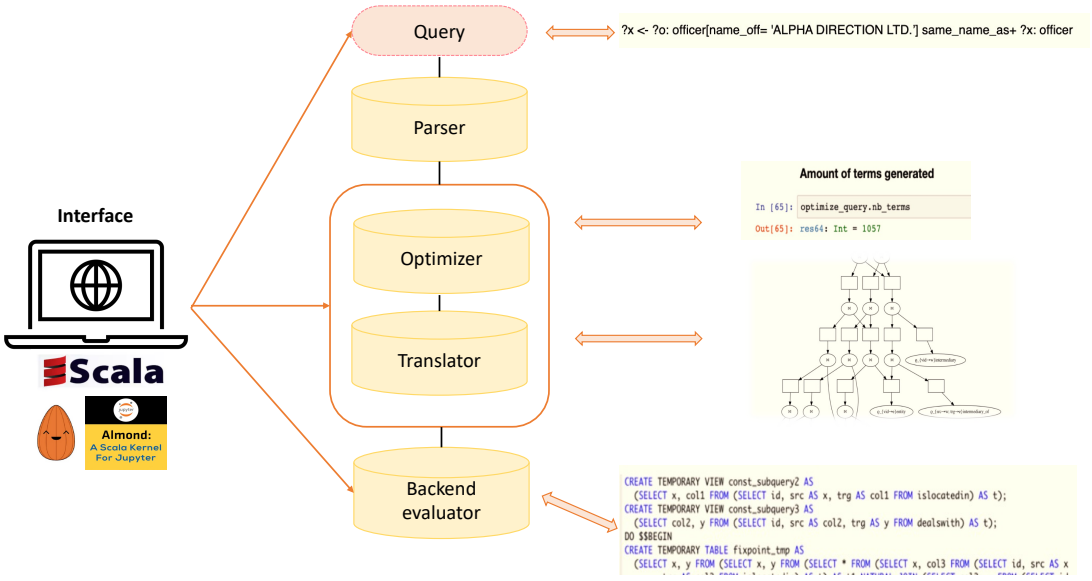

Fig. 1: Architecture of the system.

the plan space), the interface can access statistical information like: number of plans created, selected evaluation plan, etc. ; (iii) after the plan is sent to the backend evaluator (PostgreSQL in our case) it shows the SQL translation and the query evaluation time in milliseconds.

- **a parser and transformer** that parse an input query and transform it into an algebraic expression in the recursive relational algebra over the data model described in Section IV, where each node and each edge is represented in a separate table.
- **an optimizer** that generates new query evaluation plans by applying rewrite rules on the initial relational term. We demonstrate how the top-down plan enumerator efficiently explores the plan space. It enables the grouping of terms and the application of rewrite rules on sets of terms instead of individual terms. In order to select a most efficient estimated term from the generated plan space, we use a cost estimator inspired from [10].
- **translator and backend evaluator**: the selected term is then translated into SQL and sent to a relational database management system for evaluation. In our demonstration, we use PostgreSQL.

Advantages of this architecture: The demonstrated system is an end-to-end optimizer for recursive graph queries implemented as a layer on top of a widely used open-source relational database management system (PostgreSQL), without the need to modify it. A major novelty of the system is its capability to leverage the most advanced recursive algebraic transformations and query evaluation plans for recursive graph queries. The generated recursive query plans can be efficiently evaluated by PostgreSQL. However, a plain-vanilla PostgreSQL would not have been able to find such efficient plans because recursion is not supported natively. In PostgreSQL, recursion is a barrier for the application of rewrite rules in the sense that rewrite rules apply only on recursion-free subparts of the query (they cannot rearrange the structure of recursions).

IV. Application for property graphs

A property graph can be represented in the relational data model as a set of relations, with one relation per edge type and one relation per node type. Each relation encodes the specific properties of each node (or edge respectively) using one column per property. Figure 2 illustrates this representation with the Bahamas Leaks dataset [11]. This dataset is taken from the International Consortium of Investigative Journalists. It is based on the analysis of the island nation's corporate registry. It is a property graph that provides names of directors and owners of more than 175,000 Bahamian companies, trusts and foundations registered between 1990 and early 2016, together with their connections. Node tables are shown in orange and edge tables are shown in green.

Nodes are uniquely identified (the **vid** column) and those identifiers are disjoint between node types. For each edge type, the corresponding relation contains at least the source and target nodes (which are foreign keys to node's **vid**s). A knowledge graph is a particular case of a property graph with no property beyond source and target **vid**s in edge relations.

The demonstrated system optimizes and executes recursive path queries on this representation of property graphs.

V. Demonstration Scenario

To demonstrate the novelties of the system, we will rely on an interactive visualization of the RLQDAG's compact representation of terms. This visualization illustrates how sets of recursive terms are grouped together (See e.g. Figure 3). We

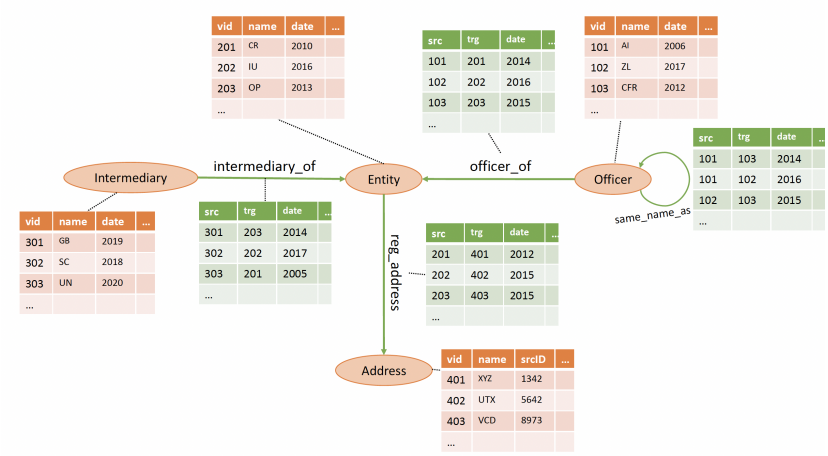

Fig. 2: Representation of the Bahamas Leaks property graph in the relational algebra data model.

will also focus on how recursive terms can be transformed. The visualization is progressively expanded with new terms obtained by the successive applications of rewrite rules, which makes the demonstration interactive.

Then, we will demonstrate the performance gains brought by the enhanced term exploration. For that purpose, we will use queries over property graphs. People in the audience will be able to formulate their own queries in addition to some predefined and third-party ones.

We will explore different scenarios with queries formulated as Unions of Conjunctions of Regular Path Queries (UCRPQs) on real datasets.

a) Scenario – Benefits of a compact representation and of sharing made possible by equivalence nodes: In this scenario we run a query over the Yago [6] dataset, which is a large graph containing more than 62 million edges between more than 42 millions of nodes. We start with a third-party query taken from [1]:

$?x \leftarrow$?x: influences/livesIn/isLocatedIn+/dealsWith+ Sweden

For the exploration of terms, we start from the initial translation of the query as a relational term. The rewrite rules of relational algebra together with the ones specific for recursion are applied, yielding many equivalent plans. These plans are presented in Figure 3. The blue squares in this figure represent the newly created equivalence nodes after the application of rewrite rules. All the other equivalence nodes were already created and reused. This is one of the main benefits of RLQDAG: the reusability of already created equivalence nodes. This means that the plan space is explored much faster, by maximizing subterm reuse and avoiding redundant computations.

b) Scenario – Exploring plan spaces: This system allows the user to write a query, to set a time budget for the plan space enumeration, and then to apply the optimization using these inputs. Among the results, we obtain statistics such as the amount of plans explored by RLQDAG (and their graphical visualization), the term chosen by the cost estimator, its SQL

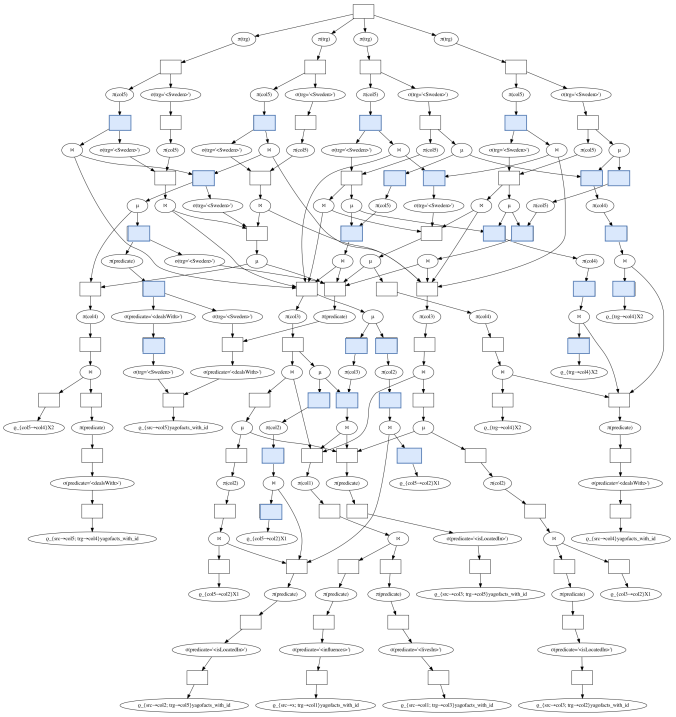

Fig. 3: The set of plans after applying all rewrite rules with RLQDAG representation.

translation, the time spent by PostgreSQL for its execution (in comparison to the execution without our optimization), etc. We will consider a query from Bahamas Leaks [11] dataset. Its schema is shown in Figure 2.

We will consider different time budgets and show the amount of plans that RLQDAG is able to generate. We will start with the following query:

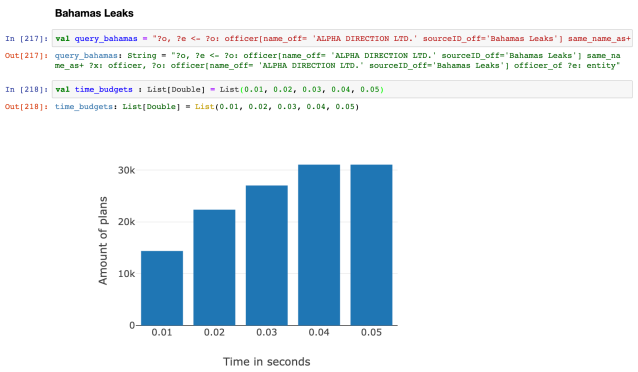

Fig. 4: Number of plans explored by RLQDAG for different time budgets.

$$?o, ?e \leftarrow ?o : \text{officer[name='AL' sourceID='Bahamas']}$$
$$\text{same\_name\_as+ } ?x : \text{officer,}$$
$$?o : \text{ officer[name='AL' sourceID='Bahamas']}$$
$$\text{officer\_of } ?y : \text{entity}$$

Figure 4 shows a generated chart presenting the sizes of the explored plan spaces for varying time budgets. The plan space is fully explored in 0.04 seconds. This is why the size of the produced plan space remains the same for time budgets 0.04 and 0.05 seconds. There are more than 31,000 plans in total. For a given time budget, we will also review the number of plans explored by RLQDAG in comparison with the mu-RA enumerator [9] that operates on individual terms. For example, we consider the following query on the Bahamas Leaks dataset:

$$?x, ?y, ?z, ?w \leftarrow ?x : \text{officer[name = 'Z L']} \text{ same\_name\_as+}$$
$$?y : \text{officer,}$$
$$?x : \text{officer[name = 'Z L']} \text{ officer\_of}$$
$$?z : \text{entity[date = '1992'],}$$
$$?z : \text{entity[date = '1992'] reg\_address } ?w\text{: address}$$

The results are shown in Figure 5. We can observe the differences between the plan space explorations made by both systems. For example, for a budget of 0.03 seconds, RLQDAG (in blue color) was able to explore more than 36,000 plans whereas the mu-RA enumerator of [9] (in orange color) was only able to explore a little more than 1,400 plans. We also measure the amount of plans explored per second by each system. Furthermore, we report the evaluation times of the best estimated plans on the same backend (PostgreSQL version 15.1) for both systems. During the live demonstration, more queries might be proposed for further experiments.

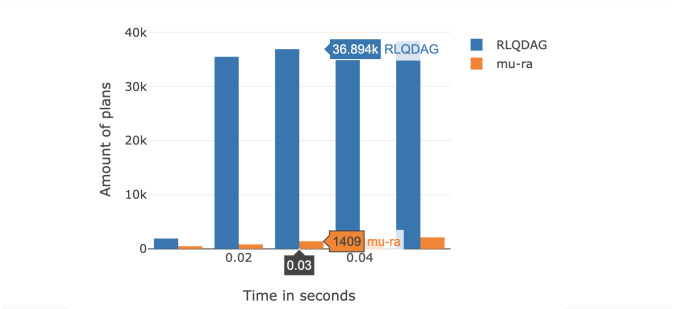

Fig. 5: Sizes of plan spaces explored by RLQDAG and mu-RA for different time budgets.

REFERENCES

[1] Z. Abul-Basher, N. Yakovets, P. Godfrey, S. Ghajar-Khosravi, and M. Chignell. Tasweet : optimizing disjunctive regular path queries in graph databases. In B. Mitschang, V. Markl, S. Bress, P. Andritsos, K.-U. Sattler, and S. Orlando, editors, *20th International Conference on Extending Database Technology, 21-24 march 2017, Venice, Italy*, pages 470–473, Mar. 2017. EDBT/ICDT 2017 Joint Conference 20th International Conference on Extending Database Technology, EDBT 2017 ; Conference date: 21-03-2017 Through 24-03-2017.

[2] R. Agrawal. Alpha: an extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering*, 14(7):879–885, July 1988.

[3] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 110–119, New York, NY, USA, 1979. ACM.

[4] Almond. Almond : A scala kernel for jupyter. almond.sh, 2023.

[5] A. Fejza, P. Genevès, and N. Layaïda. Efficient Enumeration of Recursive Plans in Transformation-based Query Optimizers. preprint: inria.hal.science/hal-03692274/document, Jan. 2023.

[6] M. P. I. for Informatics and T. P. University. Yago: A high-quality knowledge base. www.mpi-inf.mpg.de/yago-naga/yago/, july 2019.

[7] G. Graefe. The cascades framework for query optimization. *Data Engineering Bulletin*, 18, 1995.

[8] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 209–218, Washington, DC, USA, 1993. IEEE Computer Society.

[9] L. Jachiet, P. Genevès, N. Gesbert, and N. Layaïda. On the optimization of recursive relational queries: Application to graph queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 681–697, 2020.

[10] M. Lawal, P. Genevès, and N. Layaïda. A Cost Estimation Technique for Recursive Relational Algebra. In *CIKM 2020 - 29th ACM International Conference on Information and Knowledge Management*, pages 1–4, Virtual Event, France, Oct. 2020.

[11] I. C. of Investigative Journalists. Bahamas leaks. www.kaggle.com/datasets/zusmani/paradisepanamapapers, 2017.

[12] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Rec.*, 29(2):249–260, May 2000.

[13] S. Sakr, A. Bonifati, H. Voigt, A. Iosup, K. Ammar, R. Angles, W. Aref, M. Arenas, M. Besta, P. A. Boncz, et al. The future is big graphs: a community view on graph processing systems. *Communications of the ACM*, 64(9):62–71, 2021.

[14] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[15] A. Shanbhag and S. Sudarshan. Optimizing join enumeration in transformation-based query optimizers. *Proc. VLDB Endow.*, 7(12):1243–1254, aug 2014.