Optimizing Enumeration of Recursive Plans in Transformation-based Query Optimizers

Amela Fejza, Pierre Genevès, Nabil Layaïda

Optimizing Enumeration of Recursive Plans in Transformation-based Query Optimizers

Amela Fejza
amela.fejza@inria.fr
Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble
France

Pierre Genevès
pierre.geneves@inria.fr
Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble
France

Nabil Layaïda
nabil.layaida@inria.fr
Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble
France

ABSTRACT

We consider the optimization of recursive queries over graphs, and more specifically the problem of efficiently enumerating recursive query plans. For a given query, the enumeration of logical plans – and in particular join enumeration – is an extensively researched topic. However, query operators such as union and recursion have been largely neglected in enumeration techniques. We propose an extension of the logical query DAG (LQDAG) used in modern query optimizers with the support of recursion, and develop appropriate techniques to enumerate recursive terms efficiently. Our LQ DAG extension is based on the fixpoint operator recently introduced in the μ-relational algebra. We show how to adapt transformation rules for individual fixpoint terms so that they can apply on a group of terms at once. The main benefit is to enable a much more efficient exploration of recursive query plan spaces. We report on practical experiments with graph queries over real and synthetic graphs of varying size. Experimental results illustrate the benefits of exploring faster larger query execution plan spaces.

KEYWORDS

query optimization, recursion, LQDAG

1 INTRODUCTION

Graph data structures are found almost everywhere in areas like: social networks, bioinformatics research, e-commerce, transportation networks, etc. Some of these graphs can be very large in practice, in particular in e.g. semantic web. There is a growing interest in querying these graphs efficiently. Recursion is an important capability to extract information from real world graphs. Most data science workloads today involve some form of recursion, as noticed in [18]. Recently, the optimization of recursive graph queries has been a very active research topic. An extension of the relational algebra with a fixpoint operator for modeling recursion was proposed in [10]. This operator and rewrite rules introduced in [10] enable the generation of recursive logical plans that were not reachable by earlier approaches (including those based on Datalog [4]). However, while these plans were shown to be effective in practice in [10] (depending on the graph topology), the approach proposed in [10] gives no clue on how to *efficiently* compute them to populate the space of equivalent logical plans. In an actual query optimizer, the plan enumeration phase is crucial as it may produce recursive terms which are drastically more efficient. It is also highly complex, due to the very large combinatorics involved in building plan spaces. The faster we generate the space of equivalent plans, the more likely we will be able to find terms with more efficient evaluation. Ultimately, if we manage to explore the entire space of recursive plans, we find the optimal term faster. This is particularly important for recursive terms since their rule sets usually generate huge plan spaces. It is common practice to allocate a time budget for this exploration phase, as it is notoriously known that exhaustive explorations may not be feasible in practice for certain queries. The exploration phase is a prerequisite for query evaluation in any transformation-based query optimizer that first needs to find an optimal – or best estimated – term. For all these reasons, the speed of plan space exploration represents one of the most critical aspect in recursive query optimization.

Related Works. A lot of research has been done on the enumeration phase for recursion-free queries. In particular, one big issue is the complexity of join enumeration. This is studied in [14, 16] where they consider transformations based on join enumeration and consider rule sets to avoid duplicates and optimize these queries. Most works on plan enumeration techniques focused on join enumeration (see e.g. [12, 13]). However, operators such as recursion (and even union) have been largely neglected in these studies so far. As pointed out in [18], this is because most database systems have been designed to support primarily non-recursive queries. The widespread approach found in modern/recent

query optimizers such as [5, 17] is the cost-based Volcano architecture [9]. Volcano is based on the LQDAG structure, which is designed for optimizing non-recursive queries.

The addition of recursive plans increases even more the combinatorial explosion encountered in the plan enumeration problem. With the proliferation of recursive graph queries such as RPQs and UCRPQs [7], finding efficient enumeration techniques for recursive plans becomes important.

Contribution. We propose an extension, named μ-LQDAG, of the LQDAG approach with a fixpoint operator for the optimization of recursive queries. This approach generalizes known enumeration techniques to encompass recursive terms, and enables to efficiently generate gigantic plan spaces by applying rewrite rules to a set of terms at a time. We report on an experimental assessment of the approach with a prototype implementation in the setting of queries over property graphs.

2 PRELIMINARIES

2.1 Recursive Relational Algebra

The μ-relational algebra (μ-RA) [10] is an extension of Codd's relational algebra with a fixpoint operator so as to capture and to optimize recursive queries.

A term φ in μ-RA can be a *relation* X denoting a classical relational table, a *constant* $|c \to v|$ that assigns a value to a column name, a *filter* $\sigma_f(\varphi)$ that keeps only the tuples in φ that satisfy a condition f, an *antiprojection* $\widetilde{\pi}_a(\varphi)$ which removes the column named a from the term φ, a *renaming* operator $\rho_a^b(\varphi)$ that renames column a into b in the term φ, a binary operator such as *natural join* ($\varphi_1 \bowtie \varphi_2$), *antijoin*, and *union* ($\varphi_1 \cup \varphi_2$). Besides those rather classical operators, one of the main novelty of μ-RA is that a term can also be a *fixpoint operator* of the form $\mu X.\ \varphi$. This notation binds a relation X to the term φ in which X appears, hence it is an explicit way to write a recursive term in an algebraic manner.

For example, the term $\mu X.R \cup (D \bowtie X)$ denotes a relation obtained by repeatedly joining the initial relation R with a relation D until no new tuples are retrieved. The transitive closure relation R^+ is the particular case where $D = R$. The general fixpoint notation makes it possible to express not only transitive closures but also a variety of more expressive forms of recursion.

2.2 Logical Query DAG

The Logical Query DAG (LQDAG) is a data structure used to generate the logical plan space of a given query. It was introduced in [9] and also used in [15] for detecting and unifying the common subexpressions for multi-query optimization. The latter, uses a Volcano-like optimizer. LQDAG nodes are of two types: "equivalence" and "operation" nodes. Equivalence nodes can only have operation nodes as children and operation nodes can only have equivalence nodes as children. The goal of an equivalence node is to regroup a set of semantically equivalent subterms. An operation node correspond to a given algebraic operation like: join ($\bowtie$), filter (σ_θ) etc. LQDAG can be seen as a factorized representation of a large number of plans. One important aspect is that equivalence nodes are unique. In other terms one cannot find two equivalent subterms under different equivalence nodes. Each subterm appears only once. This type of representation enables the sharing of common subparts during the plan enumeration phase.

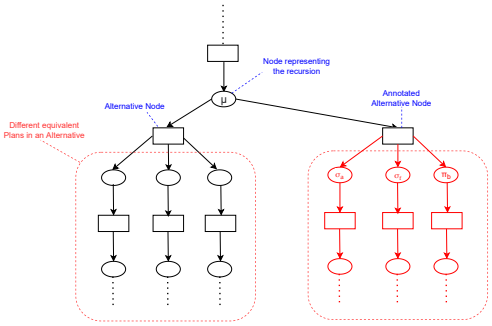

Fig. 1: Recursion operator in μ-LQDAG

3 EFFICIENT PLAN ENUMERATION

3.1 The μ-LQDAG

We propose an extended logical query dag, named μ-LQDAG, to support recursive algebraic terms. The main purpose of the μ-LQDAG is to introduce a compact representation of recursive terms that allows for their rewritings as sets of terms instead of individual terms. The goal is to enable a much faster exploration of equivalent recursive plans.

Specifically, we enrich the classical LQDAG with the introduction of a binary operator that models the μ decomposed fixpoint operator, as illustrated in Figure 1.

One of the operands models the constant part of the binary operator. It is expressed as an equivalence node that regroups all the semantically equivalent terms. The other operand models the recursive part. The main difference of the recursive part is that in this operand we will find an explicit occurrence of the recursive variable X which is bound by the fixpoint operator.

Using the representation in Figure 1, we will apply rewrite rules that will transform and create new fixpoint operators, with their respective constant and recursive parts, ensuring the correct application of rules for sets of subterms at once, and maximizing the sharing of the existing subparts.

3.2 Grouping recursive terms

One fundamental difference introduced by recursive terms when compared to the classical setting (of non-recursive terms) resides in the criteria used to trigger rule application. In the classical setting these criteria are immediate in the sense they only depend on top-level operators. In contrast, rules for transforming recursive terms rely on criteria that are more sophisticated as they require a whole traversal of the recursive part of a fixpoint term. With recursive terms,

opportunities for rule application depend on how the recursive variable is used within the recursive part of the fixpoint.

This is because there are basically two ways for translating transitive closure R^+ of a basic relation R resulting in two different forms. These two forms correspond to the two possible directions with which a fixpoint term can compute R^+.

The first form amounts to computing R^+ from left to right:

$$\mu X.\ R\ \cup\ \widetilde{\pi}_{\mathsf{m}}(\rho^{\mathsf{m}}_{\mathsf{src}}(R)\ \bowtie\ \rho^{\mathsf{m}}_{\mathsf{trg}}(X) \qquad \text{(LR)}$$

The second form computes R^+ from right to left:

$$\mu X.\ R\ \cup\ \widetilde{\pi}_{\mathsf{m}}(\rho^{\mathsf{m}}_{\mathsf{trg}}(R)\ \bowtie\ \rho^{\mathsf{m}}_{\mathsf{src}}(X) \qquad \text{(RL)}$$

Transformation rules apply differently on each form. For example: a filter on the *src* column can be pushed in the LR form that navigates from left to right, but not on the other one. Indeed, such a transformation would be incorrect (breaking the semantics) as this would filter out nodes that are not part of the final result, but are necessary intermediate nodes for reaching nodes that are part of the final results. Similarly a filter on the *trg* column can be pushed only in the form RL.

Pushing filters into fixpoints is not the only rewrite rule that can be applied to recursive terms. Other rewrite rules concerning recursive terms with the fixpoint operator include: pushing projection in a fixpoint, pushing join in a fixpoint, pushing antijoin in a fixpoint and merging fixpoints. These rules were formally introduced in [10] for transforming individual terms. The applicability of each rewrite rule depends on criteria that vary depending on the direction (LR or RL form) of the fixpoint. For instance, the rewrite rule that pushes Join in Fixpoint is the following: a term

$$\varphi \bowtie \mu(X.\ R\ \cup\ \widetilde{\pi}_{\mathsf{m}}(\rho^{\mathsf{m}}_{\mathsf{src}}(R)\ \bowtie\ \rho^{\mathsf{m}}_{\mathsf{trg}}(\psi(X)))$$

can be rewritten into

$$\mu(X'.\ \varphi \bowtie R\ \cup\ \widetilde{\pi}_{\mathsf{m}}(\rho^{\mathsf{m}}_{\mathsf{src}}(R)\ \bowtie\ \rho^{\mathsf{m}}_{\mathsf{trg}}(\psi(X')))$$

whenever the following criteria [10] hold:
(1) $t_\varphi \subseteq \text{stab}(\psi, X)$
(2) $\forall c \in t_\varphi \backslash t_k\ \text{add}(\psi, X, c)$

In these criteria, $\text{stab}(\psi, X)$ returns the set of stable columns in ψ (columns that are not changed during a fixpoint iteration) and $\text{add}(\psi, X, c)$ returns true iff the column c can be added or removed from the term ψ which is recursive in X. The functions $\text{stab}(\psi, X)$ and $\text{add}(\psi, X, c)$ are formally defined in [10] for individual terms. This means that their computation requires a traversal of the term ψ to analyse how X is used.

An important consequence for the μ-LQDAG is that the two forms (LR and RL) of fixpoints cannot be regrouped under the same equivalence set (although they are obviously semantically equivalent). The reason is that this would break the fundamental idea of the LQDAG, which seeks to apply transformation rules to a set of terms at once, and not to individual terms.

3.3 Annotated equivalence nodes

The concept of *annotated equivalence sets* amounts to distinguishing the two forms (LR and RL) in the μ-LQDAG. This means that the two forms will coexist as distinct equivalence sets, each annotated with the necessary criteria calculation required for rule application. The goal of an annotated equivalence node is to maximize the grouping of rewrite rule application, and at the same time maximize the sharing of common subparts.

Notice that from the perspective of the LQDAG factorized representation, this has an impact on the unicity of equivalence nodes for recursive terms. For annotated equivalence sets, unicity in the μ-LQDAG now depends not only on the notion of semantic equivalence of subterms, but also on the criteria. For other equivalence sets (representing non-recursive terms), unicity remains solely based on semantic equivalence.

Consistency of annotated equivalence nodes. We can show that it is valid to regroup semantically equivalent terms with the same criteria to allow rewrite rule application when possible. For this, we show that annotations remain valid throughout the application of rules (they are invariant per rule application). In other terms, the creation of new subparts during the plan enumeration phase, i.e. the expansion of the μ-LQDAG, may create new annotated equivalence nodes, or further populate existing ones, but cannot invalidate previously computed annotations. Furthermore, annotations can be computed only once (as early as possible) and never need to be updated.

3.4 Unification in μ-LQDAG

When a rewrite rule is applied new equivalence nodes can be created and others can be further populated. When an equivalence node is created, a check verifies whether there already exists an equivalent node (regrouping exactly the same set of subterms). For that purpose, in the implementation, each equivalence node has a unique hash code. This hash code is used to prevent the creation of a new instance of an already existing equivalence node. Instead, it is merged directly with the existing one.

For example, Figure 2 illustrates a μ-LQDAG that corresponds to a query that finds all the Users of a social network that are followed directly or undirectly by a young User of 20 years old. Figure 2 shows only the translation in RL form (for the sake of brevity). When the rewrite rule that pushes joins in fixpoints is applied, it first creates a new equivalence operation node (a fixpoint) at the same level as the join operation node being pushed (i.e. under the same equivalence node). In that particular case, one equivalence node is being populated whilst other equivalence nodes are being created. Each one of them is checked for pre-existence at the moment of creation to avoid duplicates. The first branch created is the one illustrated in Figure 3. The join operator is pushed in the constant part of the fixpoint operator and that is the first branch to be created. The term in green (on the left side) is the one being created, and the green one on the right side

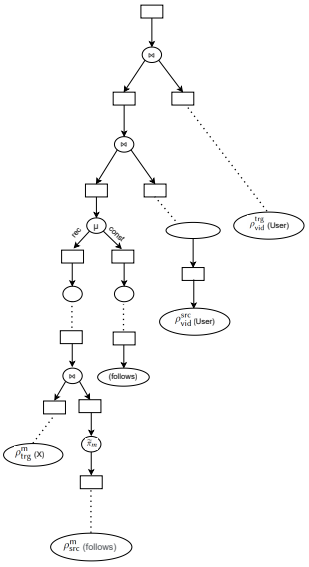

Fig. 2: μ-LQDAG example of a graph query.

is the pre-existing one. Both subparts are detected as equal (based on hash codes) and are thus merged. This results in two operation nodes at upper level pointing to the same equivalence node. This is depicted in Figure 4. Then, in a subsequent step, the other operand of join at the same level is created. This is shown in blue on the left side of Figure 5. Again, as in the previous example, a duplicate subpart is identified and merged in a single one. This is illustrated in Figure 6. In a later step, a similar situation happens with the creation of the yellow branch, as illustrated in Figure 7 which is also unified with its preexisting version. In the end, all the nodes created by this rewrite rule application are expanded and the final μ-LQDAG is the one shown in Figure 8. As we can observe, the cascade of unifications results in the creation of only the red part.

More generally, plan enumeration is conducted with all rewrite rules in a recursive top-down manner. This means that other equivalence nodes in the deeper levels are visited and expanded as well.

4 EXPERIMENTAL RESULTS

We report on experimental results obtained with a prototype implementation of the μ-LQDAG. In order to assess the efficiency of the μ-LQDAG in practice, we consider recursive graph queries formulated against various datasets (both real and generated).

Datasets. The datasets we consider are described in Table 1. Queries are chosen to reflect a variety of patterns commonly found in practice when querying property graphs. Queries and datasets used in experiments are available at [3]. We systematically compare results obtained with the μ-LQDAG with those obtained using the state-of-the-art μ-RA system implementation [10].

Dataset	#nodes & #edges	Type
Yago [8]	42M & 62M	Knowledge graph
Bahamas Leaks [2]	202K & 249K	Property Graph
Airbnb [1]	24K & 14K	Property graph
LDBC [6]	908K & 1,9M	Property Graph

Table 1: Datasets

4.1 Results

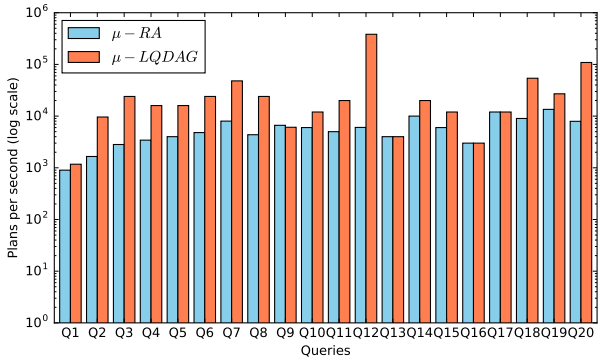

Fig. 9: Yago queries.

We report on experimentations in two stages of the query optimizer: (i) the enumeration phase, during which we assess the efficiency of the μ-LQDAG for the exploration of the space of recursive plans; (ii) the query evaluation phase, during which we assess how the more efficient exploration of plan spaces enabled by the μ-LQDAG impacts the overall query evaluation times.

Enumeration phase. First, we measure the enumeration capability of the μ-LQDAG in terms of the number of plans explored per second for each query. Figures 9, 10, 11 and 12 respectively show the results obtained for the queries over each dataset. In these figures, the y axis (in log scale) indicates the number of plans per second explored by each approach. Results suggest that the μ-LQDAG approach always enumerates plans much faster than μ-RA (up to two orders of magnitude).

Now, we set a time budget t for the plan space exploration. We let the two systems generate the plan spaces for that time budget t. This means that after t elapsed seconds we stop the exploration phase and look at the plan spaces obtained by the two approaches.

Figure 17 shows the results obtained for a time budget of $t = 0.5$ seconds. The y axis (in log scale) indicates the number of plans found. Figure 17 also indicates the size of the complete (exhaustive) plan space obtained without any time restriction for the plan exploration ($t = \infty$). For example, for query Q31, the complete plan space contains more than 21.4 million plans. The μ-LQDAG prototype explored 1,019,026 plans in 0.5 seconds, whereas the μ-RA system explored only

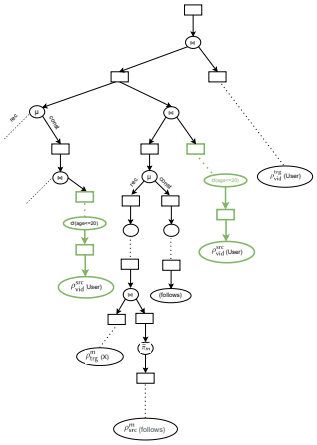

Fig. 3: Branch creation.

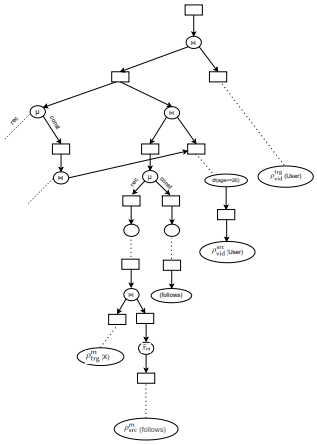

Fig. 4: Unification of constant parts.

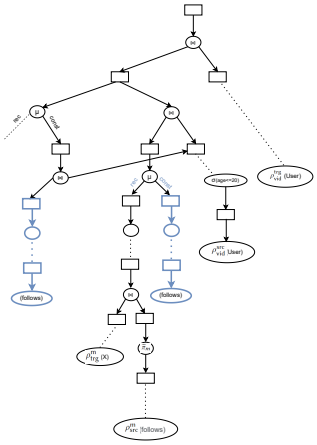

Fig. 5: Another tree branch creation.

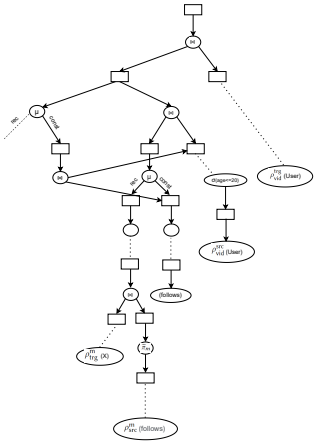

Fig. 6: Unification of branches.

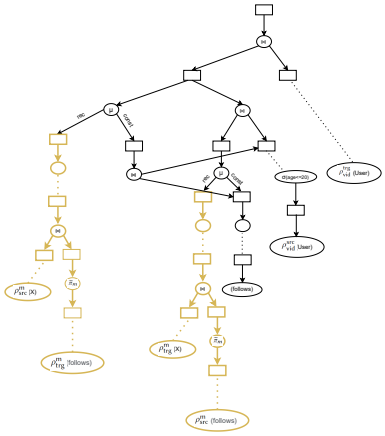

Fig. 7: Recursive branch creation.

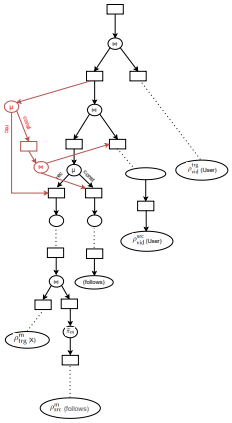

Fig. 8: Final unification.

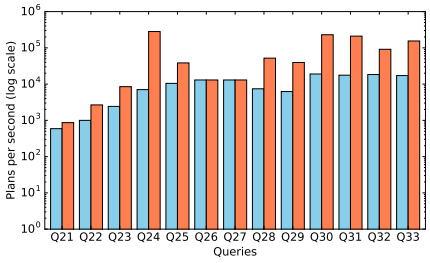

Fig. 10: Bahamas Leaks queries.

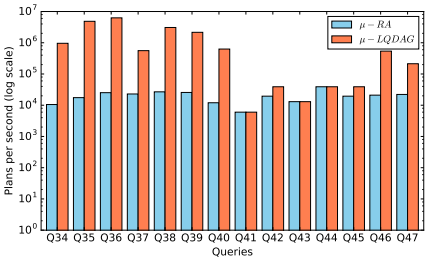

Fig. 11: Airbnb queries.

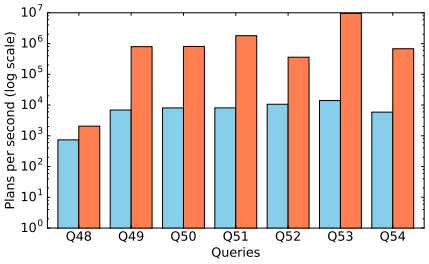

Fig. 12: LDBC queries.

5,751 terms in the same time budget of 0.5 seconds. Overall, results indicate that, for the same time budget, the μ-LQDAG prototype explores many more terms in comparison to the μ-RA system, in all cases. For some cases, the μ-LQDAG generates a number of plans which is greater by up to two orders of magnitude for the same time budget. In some cases, this speedup enables a complete exploration of the whole plan space.

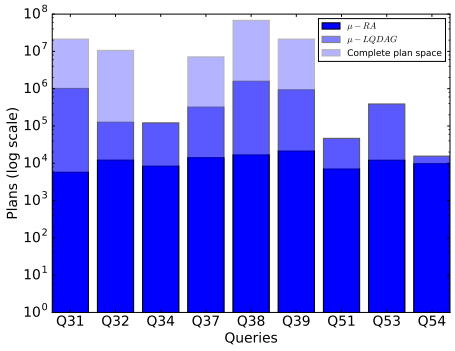

Fig. 17: Plan spaces explored in a fixed time budget (0.5 s).

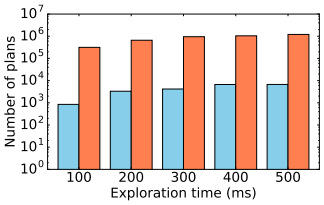

Fig. 13: Plans explored for different time budgets for Q31.

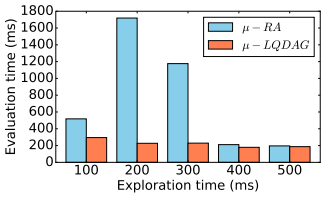

Fig. 14: Evaluation of the most efficient plan found in the explored space for Q31.

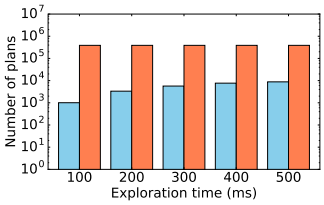

Fig. 15: Plans explored for different time budgets for Q53.

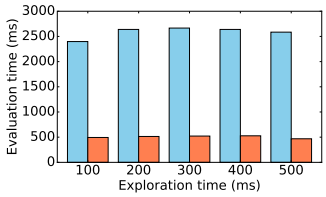

Fig. 16: Evaluation of the most efficient plan found in the explored space for Q53.

Query evaluation phase. We assess the impact of the usage of μ-LQDAG in the complete query evaluation process. This means assessing the practical performance not only during the optimization phase but on the overall time spent for query evaluation. For this we use a cost estimation function for picking a best estimated term from the plan space generated by the μ-LQDAG. We take the cost estimation function from [11] which is the cost estimation used by the μ-RA system, in order to ensure a fair comparison of the two systems with the same term selection mechanism.

We conduct experiments on queries where we set again a time budget for plan space exploration. The difference is that we now let both systems select a best (less expensive) estimated plan using the same cost estimation model. We then run the selected terms on the same backend (PostgreSQL) and measure the time spent for query evaluation.

For a given query, Figure 13 presents the number of plans explored (on the y axis) for different optimization time budgets shown on the x axis. Figure 14 shows the time spent by PostgreSQL to evaluate the best plan selected from each plan space that was generated with a given time budget (shown on the x axis). Results shown in subfigure 13 indicate that the μ-LQDAG explores significantly more plans than the other approach for all considered time budgets (the y axis is in log scale). Results shown in Figure 14 clearly show the benefits of exploring larger plan spaces since the best estimated term selected from the larger plan space is indeed more efficiently evaluated. Similar results are obtained for other queries on other datasets, as shown by e.g. in Figures 15 and 16.

5 CONCLUSION AND PERSPECTIVES

The μ-LQDAG provides an efficient technique for enumerating recursive plans, that can be used in a transformation-based query optimizer. Experiments with a prototype implementation show the practical benefits brought by a faster exploration of larger plan spaces for evaluating recursive queries over graphs. As a perspective for future work we plan to experiment the μ-LQDAG approach with other cost estimation techniques. We also plan to investigate branch-and-bound pruning techniques to direct the search space exploration.

REFERENCES

[1] Airbnb. http://insideairbnb.com/get-the-data.html.
[2] Bahamas leaks. https://www.kaggle.com/zusmani/paradisepanamapers.
[3] Datasets and queries used in experiments with the μ-LQDAG. https://gitlab.inria.fr/tyrex-public/mulqdag, 2022.
[4] P. Alvaro, W. Marczak, N. Conway, J. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in time and space. pages 262–281, 01 2010.
[5] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire. 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 18, page 221230, New York, NY, USA, 2018. Association for Computing Machinery.
[6] P. Boncz. Ldbc: Benchmarks for graph and rdf data management. In *Proceedings of the 17th International Database Engineering ; Applications Symposium*, IDEAS '13, page 12, New York, NY, USA, 2013. Association for Computing Machinery.
[7] A. Bonifati, G. Fletcher, H. Voigt, and N. Yakovets. *Querying graphs*. Morgan & Claypool Publishers, 2018.
[8] M. P. I. for Informatics and T. P. University. Yago: A high-quality knowledge base. https://www.mpi-inf.mpg.de/yago-naga/yago/, july 2019.
[9] 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.
[10] 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. `https://hal.inria.fr/hal-01673025v5/document`.
[11] 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.
[12] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204215, Nov. 2015.
[13] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal*, 27(5):643668, oct 2018.
[14] A. Pellenkoft, C. Galindo-Legaria, and M. Kersten. The complexity of transformation-based join enumeration. pages 306–315, 01 1997.
[15] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Rec.*, 29(2):249260, May 2000.
[16] A. Shanbhag and S. Sudarshan. Optimizing join enumeration in transformation-based query optimizers. *Proc. VLDB Endow.*, 7(12):12431254, aug 2014.
[17] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, F. Rahman, M. Petropoulos, F. Waas, S. Narayanan, K. Krikellas, and R. Baldwin. Orca: A modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD 14, page 337348, New York, NY, USA, 2014. Association for Computing Machinery.
[18] Y. R. Wang, M. A. Khamis, H. Q. Ngo, R. Pichler, and D. Suciu. Optimizing recursive queries with program synthesis, 2022.