The Mu-RA System for Recursive Path Queries over Graphs

Amela Fejza, Pierre Genevès, Nabil Layaïda, Sarah Chlyah

The μ-RA System for Recursive Path Queries over Graphs

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

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

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

Sarah Chlyah
Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
Grenoble, France
sarah.chlyah@inria.fr

ABSTRACT

We demonstrate a system for recursive query answering over graphs. The system is based on a complete implementation of the recursive relational algebra μ-RA, extended with parsers and compilers adapted for queries over knowledge and property graphs. Each component of the system comes with novelty for processing recursion. As a result, one can formulate, optimize and efficiently answer expressive queries that navigate recursively along paths in different types of graphs. We demonstrate the system on real datasets and show how it performs considering other state-of-the-art systems.

CCS CONCEPTS

• **Information systems → Network data models**; **Graph-based database models**; **Relational database query languages**.

KEYWORDS

query languages; query optimization; path queries; recursive relational algebra; property graphs; graphs

ACM Reference Format:
Amela Fejza, Pierre Genevès, Nabil Layaïda, and Sarah Chlyah. 2023. The μ-RA System for Recursive Path Queries over Graphs. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management (CIKM '23), October 21–25, 2023, Birmingham, United Kingdom.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3583780.3614756

1 INTRODUCTION

Graph data structures have become increasingly popular in various applications such as social networks, knowledge extraction, transportation networks, etc.

Recent developments have shown a significant interest in equipping query languages with recursive query constructs such as regular path queries (RPQs) and their combinations. They enable queries to exploit the linked nature of graph data by allowing for navigations along deep paths in the graph. The optimization of recursive queries is notoriously known as a difficult problem. Recently, μ-RA [12] introduced a clean generalization of Codd's relational algebra for the optimization of recursive queries, enabling in particular richer query evaluation plan spaces than with earlier approaches.

We demonstrate a system for recursive query answering over graph datasets. The system includes a complete implementation of the theory of the recursive relational algebra and concepts introduced in [4, 5, 7, 8, 12, 14]. The system also includes parsers and compilers so that one can formulate, optimize and answer queries that navigate recursively in knowledge and property graphs. We explain the main components of the μ-RA system (parser, optimizer and translator) and then the backend evaluator to evaluate the best estimated plan in a centralized or distributed way [4, 5].

One major novelty and advantage of this system resides in being a layer on top of an unmodified backend like PostgreSQL. In other terms, the optimization layer can be easily plugged or unplugged on top of existing relational backends, not requiring any modification of PostgreSQL for example. Yet, the overall system offers comparable or even superior performance with respect to recent graph-specific engines (such as *MilleniumDB* [20]) for recursive query answering over practical graphs such as Yago [10, 18].

The purpose of this demonstration is to present the system in an interactive manner, showing: (i) a discovery and practical tour of the different components and functionalities of the system using concrete queries over real practical graphs; (ii) demonstrating that the system also supports expressive path patterns (regular and not regular) not supported by other systems, yet useful in formulating meaningul queries on real datasets. In particular, one purpose of this demonstration is to disprove a wrong statement made in a recent paper [15] about the expressivity of the approach; and (iii) showing that the system provides comparable or superior performance when compared to some state-of-the-art systems for answering queries with recursive path patterns on real datasets.

2 PRELIMINARY BACKGROUND

The recursive relational algebra (μ-RA) [12] 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* $|c \rightarrow v|$

CIKM '23, October 21–25, 2023, Birmingham, United Kingdom.

ACM ISBN 979-8-4007-0124-5/23/10…$15.00
https://doi.org/10.1145/3583780.3614756

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 features 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. The theory also comes with a set of algebraic rewrite rules specifically designed to transform fixpoint terms into semantically equivalent (yet more efficient) variants, generalizing the initial idea of Codd to recursive queries. In particular, transformations include algebraic rewritings where filters, antiprojections and joins are "pushed through" fixpoint terms (modulo some conditions are satisfied), yielding other fixpoint terms whose evaluation can be way more efficient than the initial ones. Some fixpoint terms can also be merged into a single fixpoint term, replacing two recursions by a single one.

Such transformation rules, together with the necessary mechanisms to check the conditions under which they are valid, are detailed in [12]. This basically opens the way to much more efficient query evaluation plans for recursive queries, that are out of reach of earlier approaches, especially those based on Datalog [3, 16, 17, 19]. This is because in a Datalog optimizer there is no equivalent to merging fixpoints, therefore currently in a Datalog program corresponding to the optimized translation of a^+/b^+ at least one of the two transitive closures a^+ or b^+ will be fully materialized (even if there is no solution to a^+/b^+).

In μ-RA, this query can be evaluated by a single recursion that starts from a/b and recursively appends a on the left or b on the right. Since the size of such transitive closures may be an order of magnitude larger than the graph size, performance gains can be significant on real datasets.

3 SYSTEM ARCHITECTURE

The system is coded in Scala (39K lines of code). The overall system architecture is illustrated in Figure 1. It is composed of several components:

An interface. which is an interactive notebook that uses Almond [2], a modern Scala shell that provides features in Jupyter notebooks. The interface exposes an API to interact with the different μ-RA system components and displays information about the different stages of the query optimization process. The user provides a graph query with recursive path patterns. The interface also displays query answers and information and statistics on the evaluation process (query evaluation time, number of results, selected evaluation plan etc.)

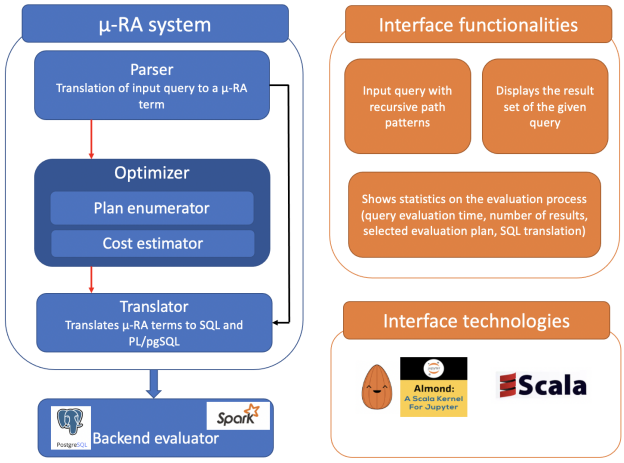

Figure 1: Architecture of the system.

A parser of high-level query languages. that parses the input query and translates it into a μ-RA term that operates on a relational representation of the queried graph. Specifically, μ-RA adopts a relational representation of graphs where each edge is represented as a relation. Property graph representations have extra relations for representing node properties. The parser is the only component that is dependent on the chosen graph relational representation. μ-RA supports the relational data model on which RA is based.

An optimizer. that has two sub-components. The first one is *a plan enumerator* that applies all the algebraic rewrite rules of classical relational algebra together with the rewrite rules concerning fixpoint terms presented in [12], in a fully compositional way. By composition, all these rules enable new query evaluation plans (in particular merged fixpoints, and their further transformations) that were not possible with earlier approaches. Overall, this results in a richer query evaluation plan space. For enumerating the plan space, there are two possibilities: (i) a bottom-up enumerator where the application of rewrite rules is applied to one term at a time; (ii) a top-down plan enumerator [7, 8] where the plan space is explored by transforming sets of terms at once. The second subcomponent is *a cost estimator* that picks an estimated most efficient term, based on cost estimations for the algebraic operators and cardinality statistics of the datasets. This cost estimation technique is inspired from [13, 14]. The optimizer component can be plugged or unplugged. It makes it easy to measure the performance gains brought by the logical optimizations.

A translator. that translates the estimated best term into SQL using Recursive Common Table Expression (CTE) or PL/pgSQL with WHILE loops for more general forms of recursion not expressible with CTEs. This is for instance the case of merged fixpoint terms for the PostgreSQL backend. The distributed evaluator component uses in one of its distributed evaluation plans the SQL translator to compute the fixpoint on PostgreSQL evaluators present on parallel workers [4].

A backend evaluator. which provides two possibilities: (i) a relational database managment system in charge of evaluating the translation of the best estimated plan. For this demonstration, the relational backend used is PostgreSQL. (ii) a distributed evaluator that distributes the evaluation of μ-RA terms on Spark [4, 5]. After evaluation, query answers and information about the evaluation are presented in the user interface.

The fact that the optimizer is a separate component that sits on top of unmodified backends offers several advantages. In particular, performance gains do not rely on low level system optimizations of platforms that are subject to change, but rather on logical optimizations (of recursive path queries) that can be applied on any platform capable of evaluating μ-RA terms.

4 DEMONSTRATION

In this demonstration we give a complete overview and practical tour of the centralized part of the system that uses PostgreSQL as a backend [1]. This demonstration is performed with the μ-RA interactive interface. We propose three scenarios focusing respectively on the process that a user query goes through in the μ-RA system before being executed; expressivity; and performance comparisons.

4.1 Regular Path Query scenario

We start by showing an example of an UCRPQ (Unions of Conjunctions of Regular Path Queries) query of the form (a^+/b^+) over the Yago dataset. We use a preprocessed version of the real world Yago2s [10, 18] dataset. This dataset version contains 83 predicates and 62,643,951 rows (graph edges). They correspond to general knowledge about people, cities, countries, etc. This example is shown in Figure 2. The user writes the query, sets a time budget for the plan space enumeration, and then calls the optimizer with these inputs. The call to the `optimize` procedure successively invokes the following components: the parser, the plan enumerator, and the cost estimator (see Sec. 3). The user can then retrieve information such as the total number of explored plans and the chosen plan. The interface provides the user with a feature that automatically generates visual representations of plans. In this representation, squares represent the equivalence nodes and circles represent operation nodes. The purpose of equivalence nodes is to regroup plans that are semantically equivalent. Each operator node is an algebraic operator that can be unary (e.g. filter) or binary (e.g. join).

Once a plan is chosen, it is translated to SQL before execution on PostgreSQL. Figure 4 and Figure 5 show two possible translations for the initial query considered in Figure 2. Plan 2 translation in Figure 5 corresponds to the translation of the chosen plan graphically displayed in Figure 3. In this plan, a/b is computed first and each iteration in the PL/pgSQL WHILE loop adds to the result set all paths obtained by appending a to the left of the previously known paths as well as all paths obtained by appending b to the the right of those previous paths. In Figure 4, Plan 1 translation shows the SQL translation of a different plan for the same query which has two fixpoint terms (not merged). In this case, each recursion is translated into a Recursive Common Table Expression (CTE) individually and $a+$ and $b+$ are respectively first calculated fully and then joined

[1]We provide an accompanying video [6].

Write the query

```
In [11]: val query = "?x, ?y <- ?x(} islocatedin+/dealswith+ ?y(}"
Out[11]: query: String = "?x, ?y <- ?x(} islocatedin+/dealswith+ ?y(}"
```

Decide the optimization time budget (in seconds)

```
In [12]: val time_budget = 0.01
Out[12]: time_budget: Double = 0.01
```

Optimize the query

```
In [13]: val optimize = muRA_system.optimize(query, emap, allrelvars, time_budget, optimParams)
Out[13]: optimize: tests.OptimizeResultSetMu = ...
```

Number of terms

```
In [14]: optimize.nb_terms
Out[14]: res13: Int = 15
```

Create and show an Image of the chosen term

```
In [15]: muRA_system.createFigurechosenTerm(optimize.term_chosen, "chosenTerm")
In [16]: showTerm("chosenTerm")
```

Figure 2: An example in the interactive notebook.

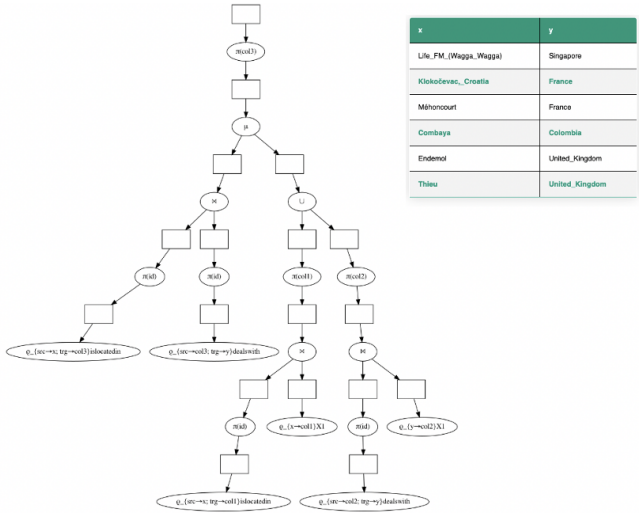

Figure 3: A plan graphically displayed in the notebook.

together. During the live demonstration, this translation in particular will be explained in more details. The user can also display the different evaluation times of the chosen plans in the interface. An excerpt of the retrieved query answers is shown in Figure 2.

4.2 Expressivity scenario

In the second scenario, we focus on the expressive power of queries supported by the system. This scenario focuses on two key aspects:

(i) An erroneous statement of [15] states that μ-RA cannot express $(abc(dbc)*)+$. This statement is wrong because the query corresponds to a regular path query, hence trivially supported in μ-RA. For example, $ab*$ stands for $a|ab+$. Hence, $(abc(dbc)*)+$ can be simply expressed as $(abc|abc(dbc)+)+$. We further explain using a concrete example why the statement is not true by writing and evaluating a query of this form over the Yago dataset. We use the query :

```
CREATE TEMPORARY VIEW const_subquery2 AS
  (SELECT col1, col3 FROM (SELECT id, src AS col1, trg AS col3 FROM islocatedin) AS t);
CREATE TEMPORARY RECURSIVE VIEW fixpoint_relation1_X1 (x, col3) AS
    SELECT x, col3 FROM (SELECT x, col3 FROM (SELECT id, src AS x, trg AS col3 FROM
        islocatedin) AS t) AS const
  UNION
    SELECT x, col3 FROM (SELECT col3, x FROM (SELECT * FROM (SELECT x, col3 AS col1 FROM
        fixpoint_relation1_X1) AS t NATURAL JOIN const_subquery2) AS t) AS rec;
CREATE TEMPORARY VIEW const_subquery4 AS
  (SELECT col2, y FROM (SELECT id, src AS col2, trg AS y FROM dealswith) AS t);
CREATE TEMPORARY RECURSIVE VIEW fixpoint_relation3_X2 (col3, y) AS
    SELECT col3, y FROM (SELECT col3, y FROM (SELECT id, src AS col3, trg AS y FROM dealswith)
        AS t) AS const
  UNION
    SELECT col3, y FROM (SELECT y, col3 FROM (SELECT * FROM (SELECT col3, y AS col2 FROM
        fixpoint_relation3_X2) AS t NATURAL JOIN const_subquery4) AS t) AS rec;
SELECT DISTINCT * FROM (SELECT x, y FROM (SELECT * FROM (SELECT * FROM fixpoint_relation1_X1)
    AS t1 NATURAL JOIN (SELECT * FROM fixpoint_relation3_X2) AS t2) AS t) AS t;
```

Figure 4: SQL translation of the first plan for query Q.

```
CREATE TEMPORARY VIEW const_subquery2 AS
  (SELECT x, col1 FROM (SELECT id, src AS x, trg AS col1 FROM islocatedin) AS t);
CREATE TEMPORARY VIEW const_subquery3 AS
  (SELECT col2, y FROM (SELECT id, src AS col2, trg AS y FROM dealswith) AS t);
DO $$BEGIN
CREATE TEMPORARY TABLE fixpoint_tmp AS
  (SELECT x, y FROM (SELECT x, y FROM (SELECT * FROM (SELECT x, col3 FROM (SELECT id, src AS x
      , trg AS col3 FROM islocatedin) AS t) AS t1 NATURAL JOIN (SELECT col3, y FROM (SELECT id
      , src AS col3, trg AS y FROM dealswith) AS t) AS t2) AS t) AS t);
CREATE TEMPORARY TABLE fixpoint_relation1_X1 AS (SELECT * FROM fixpoint_tmp);
WHILE EXISTS (SELECT 1 FROM fixpoint_relation1_X1) LOOP
  CREATE TEMPORARY TABLE nouvelles AS
    (SELECT x, y FROM (SELECT y, x FROM (SELECT y, x FROM (SELECT * FROM (SELECT x AS col1, y
        FROM fixpoint_relation1_X1) AS t NATURAL JOIN const_subquery2) AS t) AS t1 UNION
        SELECT y, x FROM (SELECT x, y FROM (SELECT * FROM (SELECT x, y AS col2 FROM
        fixpoint_relation1_X1) AS t NATURAL JOIN const_subquery3) AS t) AS t2) AS rec_req
        EXCEPT SELECT * FROM fixpoint_tmp);
  INSERT INTO fixpoint_tmp (SELECT * FROM nouvelles);
  DROP TABLE fixpoint_relation1_X1;
  ALTER TABLE nouvelles RENAME TO fixpoint_relation1_X1;
END LOOP;
DROP TABLE fixpoint_relation1_X1;
ALTER TABLE fixpoint_tmp RENAME TO fixpoint_relation1_X1;
END;$$;
```

Figure 5: SQL translation of the second plan for query Q.

Q=((haschild/islocatedin/dealswith/(ismarriedto/islocatedin/dealswith)+)
| (haschild/islocatedin/dealswith))+.

We also describe how this query is optimized in the μ-RA system. The optimized term is sent to PostgreSQL and evaluted in 0.6 seconds on a commodity laptop. Note that the query given in [15] and query **Q** cannot be expressed in Cypher 9 language [11].

(ii) μ-RA supports some queries with particular non-regular forms of recursion[2], currently not supported in [15].

An example of a supported non-regular query pattern is $a^n b^n$, that returns the pairs of nodes connected by a path composed of a number of edges labeled a followed by the same number of edges labeled b. We can express a query of this form on wikitree [9], a well-known genealogy dataset. We propose a query that retrieves pairs of people that are of the same generation (which is a special case of $a^n b^n$).

This query can be written in μ-RA [12] syntax as follows:

$$\sigma_{pred=\text{parent}}(\mu(X.\ \widetilde{\pi}_m(\rho^{\text{m}}_{\text{src}}(R)\ \bowtie\ \rho^{\text{m}}_{\text{src}}(R)) \cup \widetilde{\pi}_m(\widetilde{\pi}_n(\rho^{\text{src}}_{\text{trg}}(\rho^{\text{m}}_{\text{src}}(R)) \bowtie\ \rho^{\text{n}}_{\text{trg}}(\rho^{\text{m}}_{\text{src}}(X)\ \bowtie\ \rho^{\text{n}}_{\text{src}}(R))))))$$

Figure 6 shows the plan space explored when this query is processed by the plan enumerator of the μ-RA system.

4.3 Performance comparison scenario

In this scenario, we focus on the performance gains brought in practice. For this purpose, we consider a recent state-of-the-art system: *MilleniumDB* [20]. We assess the two systems on a common set of third-party queries over the public Yago dataset. Furthermore,

[2]going beyond the expressive power of regular patterns and UCRPQs in particular.

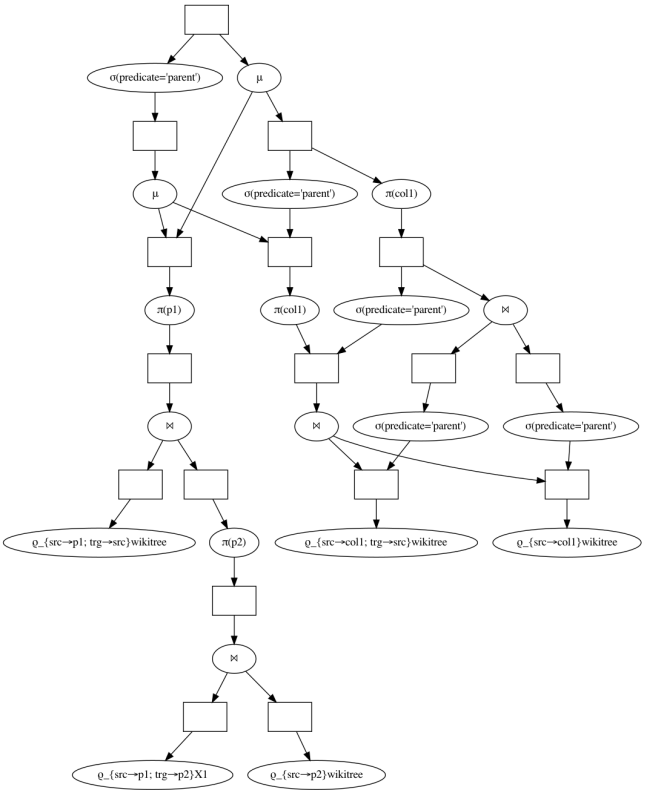

Figure 6: Some explored plans for the non-regular subquery

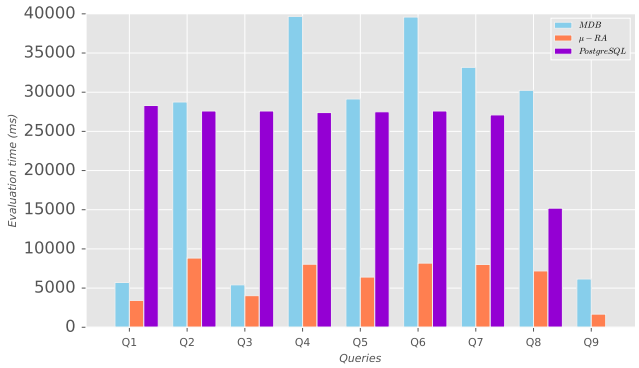

Figure 7: Comparison with state-of-the-art systems.

to evaluate the impact of the μ-RA optimizer, we also disable it and compare with the evaluation times obtained with a plain-vanilla PostgreSQL that receives a (non-optimized) translation of the initial query (see the black and red arrows in Figure 1).

We evaluate 7 queries (Q1-Q7) taken from [1] and 2 queries (Q8-Q9) taken from [21], on μ-RA, *MilleniumDB* (MDB) and PostgreSQL. Results are shown in Figure 7. μ-RA outperforms clearly the other two systems. On Q9, PostgreSQL does not even respond. During the demo, attendees will also be able to propose their own queries to experiment with.

REFERENCES

[1] Zahid Abul-Basher, Nikolay Yakovets, Parke Godfrey, Shadi Ghajar-Khosravi, and Mark H Chignell. 2017. Tasweet: optimizing disjunctive regular path queries in graph databases. In *EDBT/ICDT 2017 joint conference 20th international conference on extending database technology. https://doi. org/10.5441/002/edbt.*

[2] Almond. 2023. Almond : A Scala kernel for Jupyter. https://almond.sh/.

[3] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. ACM, New York, NY, USA, 1371–1382. https://doi.org/10.1145/2723372.2742796

[4] Sarah Chlyah. 2022. *On Algebraic Foundations for the Optimization of Iterative Programming with Distributed Data Collections. (Fondements algébriques pour l'optimisation de la programmation itérative avec des collections de données distribuées)*. Ph. D. Dissertation. Grenoble Alpes University, France. https://tel.archives-ouvertes.fr/tel-03783672

[5] Sarah Chlyah, Pierre Genevès, and Nabil Layaïda. 2021. Distributed Evaluation of Graph Queries using Recursive Relational Algebra. arXiv:2111.12487 [cs.DB]

[6] Amela Fejza. 2023. Accompanying video for the paper "The μ-RA System for Recursive Path Queries Over Graphs". https://shorturl.at/AXZ23

[7] Amela Fejza. 2023. *On the Optimization of Recursive Plan Enumeration with an Application to Property Graph Queries. (Sur l'optimisation de l'énumération de plans récursifs avec une application aux requêtes de graphes de propriétés)*. Ph. D. Dissertation. Grenoble Alpes University, France. https://tel.archives-ouvertes.fr/tel-04128256

[8] Amela Fejza, Pierre Genevès, and Nabil Layaïda. 2023. Efficient Enumeration of Recursive Plans in Transformation-based Query Optimizers. (Jan. 2023). https://hal.inria.fr/hal-03692274 preprint.

[9] Michael Fire and Yuval Elovici. 2015. Data mining of online genealogy datasets for revealing lifespan patterns in human population. *ACM Transactions on Intelligent Systems and Technology (TIST)* 6, 2 (2015), 28.

[10] Max Planck Institute for Informatics and Telecom ParisTech University. 2019. YAGO: A high-quality knowledge base. https://www.mpi-inf.mpg.de/yago-naga/yago/.

[11] 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* (Houston, TX, USA) *(SIGMOD '18)*. New York, NY, USA, 1433–1445. https://doi.org/10.1145/3183713.3190657

[12] Louis Jachiet, Pierre Genevès, Nils Gesbert, and Nabil Layaïda. 2020. On the Optimization of Recursive Relational Queries: Application to Graph Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 681–697. https://doi.org/10.1145/3318464.3380567

[13] Muideen Lawal. 2021. *On Cost Estimation for the Recursive Relational Algebra. (Sur l'estimation des coûts pour l'algèbre relationnelle récursive)*. Ph. D. Dissertation. Grenoble Alpes University, France. https://tel.archives-ouvertes.fr/tel-03322720

[14] Muideen Lawal, Pierre Genevès, and Nabil Layaïda. 2020. A Cost Estimation Technique for Recursive Relational Algebra. In *CIKM 2020 - 29th ACM International Conference on Information and Knowledge Management*. Virtual Event, France, 1–4. https://doi.org/10.1145/3340531.3417460

[15] Wilco v. Leeuwen, Thomas Mulder, Bram van de Wall, George Fletcher, and Nikolay Yakovets. 2022. AvantGraph Query Processing Engine. *Proc. VLDB Endow.* 15, 12 (aug 2022), 3698–3701. https://doi.org/10.14778/3554821.3554878

[16] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. 2006. The DLV System for Knowledge Representation and Reasoning. *ACM Trans. Comput. Logic* 7, 3 (July 2006), 499–562. https://doi.org/10.1145/1149114.1149117

[17] John D. Ramsdell. 2004. Datalog version 2.2, a lightweight deductive database system. http://www.ccs.neu.edu/home/ramsdell/tools/datalog/datalog.html (retrieved in october 2019).

[18] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: A Core of Semantic Knowledge. In *Proceedings of the 16th International Conference on World Wide Web* (Banff, Alberta, Canada) *(WWW '07)*. ACM, New York, NY, USA, 697–706. https://doi.org/10.1145/1242572.1242667

[19] Jacopo Urbani, Ceriel J. H. Jacobs, and Markus Krötzsch. 2016. VLog: A Column-Oriented Datalog System for Large Knowledge Graphs. In *Proceedings of the ISWC 2016 Posters & Demonstrations Track co-located with 15th International Semantic Web Conference (ISWC 2016), Kobe, Japan, October 19, 2016.* http://ceur-ws.org/Vol-1690/paper113.pdf

[20] Domagoj Vrgoc, Carlos Rojas, Renzo Angles, Marcelo Arenas, Diego Arroyuelo, Carlos Buil Aranda, Aidan Hogan, Gonzalo Navarro, Cristian Riveros, and Juan Romero. 2021. MillenniumDB: a persistent, open-source, graph database. *arXiv preprint arXiv:2111.01540* (2021).

[21] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2015. WAVEGUIDE: Evaluating SPARQL Property Path Queries.. In *EDBT*, Vol. 2015. 525–528.