



HAL
open science

Distributed Evaluation of Graph Queries using Recursive Relational Algebra

Sarah Chlyah, Pierre Genevès, Nabil Layaïda

► **To cite this version:**

Sarah Chlyah, Pierre Genevès, Nabil Layaïda. Distributed Evaluation of Graph Queries using Recursive Relational Algebra. 2021. hal-03295445v1

HAL Id: hal-03295445

<https://inria.hal.science/hal-03295445v1>

Preprint submitted on 22 Jul 2021 (v1), last revised 7 Oct 2022 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed Evaluation of Graph Queries using Recursive Relational Algebra

Sarah Chlyah*, Pierre Genevès* Nabil Layaïda*

* Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

Abstract—We present a system called **Dist- μ -RA** for the distributed evaluation of recursive graph queries. **Dist- μ -RA** builds on the recursive relational algebra and extends it with evaluation plans suited for the distributed setting. The goal is to offer expressivity for high-level queries while providing efficiency at scale and reducing communication costs. Experimental results on both real and synthetic graphs show the effectiveness of the proposed approach compared to existing systems.

Index Terms—graph queries, recursion, distributed evaluation

I. INTRODUCTION

With the proliferation of large scale graphs in various domains (such as knowledge representation, social networks, transportation, biology, property graphs, etc.), the need for efficiently extracting information from these graphs becomes increasingly important. This often requires the development of methods for effectively distributing both data and computations so as to enable scalability. Efforts to address these challenges over the past few years have led to various systems such as MapReduce [1], Dryad [2], Spark [3], Flink [4] and more specialized graph systems like Google Pregel [5], Giraph [6] and Spark Graphx [7]. While these systems can handle large amounts of data and allow users to write a broad range of applications, they still require significant programmer expertise. The system programming paradigms and its underlying configuration tuning must be highly mastered. This includes for example figuring out how to (re)partition data on the cluster, when to broadcast data, in which order to apply operations in order to reduce data transfers between nodes of the cluster, etc.

One approach to this problem is to have an intermediate representation of queries (e.g. an algebra) in which high level queries are translated so that they can be optimized automatically. The idea is to relieve users from having to worry about optimization in the distributed setting, so that they can focus only on formulating domain-specific queries in a declarative manner. Relational Algebra (RA) is such an intermediate representation that has benefited from decades of research, in particular on algebraic rewriting rules in order to compute efficient query evaluation plans.

A very important feature of graph queries is recursion, which enables to express complex navigation patterns to extract

useful information based on connectivity from the graph. For instance, recursion is crucial for supporting queries based on transitive closures. Recursive queries on large-scale graphs can be very costly or even infeasible. This is due to a large combinatorial of basic computations induced by both the query and the graph topology. Recursive queries can generate intermediate results that are orders of magnitude larger than the size of the initial graph. For example, a query on a graph of millions of nodes can generate billions of intermediate results. Therefore being able to optimize queries and reduce the size of intermediate results as much as possible becomes crucial. Many works have addressed the optimization of recursion like Datalog [8] and extensions of Relational Algebra [9], [10] with recursion. A recent work on μ -RA [11] proposes an interesting balance between expressivity and optimization capabilities to provide state-of-the-art performance for evaluating recursive graph queries in a centralized setting.

We present **Dist- μ -RA**, a system that leverages and extends the μ -RA algebra to the distributed setting. Our goal is to offer expressivity for high-level queries while providing efficiency at scale. **Dist- μ -RA** is built on top of Spark. It combines the advantages of (i) RA and its large body of research work on its optimization, of (ii) μ -RA which provides optimizations for recursion with a fully compositional and seamless integration within the RA framework, and of (iii) those of Spark. Spark is better suited for iterative applications than other distributed systems like MapReduce, thanks to its distributed in-memory data collection abstraction (RDD) [12]. **Dist- μ -RA** provides optimized physical planning strategies and an architecture where μ -RA constructs are efficiently executed on Spark. **Dist- μ -RA** also provides a frontend where the programmer can formulate queries known as UCRPQs [13]–[16]. UCRPQs, discussed in more details in Sec. III, constitute an important fragment of expressive graph query languages: they correspond to unions of conjunctions of regular path queries. Our contributions can be summarized as follows:

- A system for the optimization and distribution of UCRPQs and μ -RA terms. We leverage the μ -RA formalism to enable advanced optimizations of logical plans with recursion (such the merge of several recursive subterms in a single one).
- A technique for generating optimized physical plans for recursion that distribute data on a cluster and reduce data transfers between cluster nodes.

This research has been partially supported by the ANR project CLEAR(ANR-16-CE25-0010).

- A prototype implementation and an experimental comparison with state-of-the-art systems. Experiments are conducted on real datasets as well as synthetic datasets of various sizes. Experimental results indicate that improvements in query evaluation performance can be of an order of magnitude, compared to existing systems.

The outline of the paper is as follows: we first present a summary of useful preliminary notions from the μ -RA algebra in Section II. Section III presents the whole architecture of Dist- μ -RA. In Section IV, we show how μ -RA terms are distributed and how physical plans are generated. Experimental evaluation is presented in Section V and related works are discussed in Section VI.

II. PRELIMINARIES

A. μ -RA syntax

The μ -RA algebra [11] is an extension of the Codd's relational algebra with a recursive operator whose aim is to support recursive terms and transform them when seeking efficient evaluation plans. The syntax of μ -RA is recalled from [11] in Fig. 1. It is composed of database relation variables and operations (like join and filter) that are applied on relational tables to yield other relational tables. μ is the fixpoint operator. In $\mu(X = \Psi)$, X is called *the recursive variable* of the fixpoint term.

$\varphi ::=$	X	term
	$ c \rightarrow v $	relation variable
	$\varphi_1 \cup \varphi_2$	constant
	$\varphi_1 \bowtie \varphi_2$	union
	$\varphi_1 \bowtie \varphi_2$	natural join
	$\varphi_1 \triangleright \varphi_2$	antijoin
	$\sigma_f(\varphi)$	filtering
	$\rho_a^b(\varphi)$	renaming
	$\tilde{\pi}_a(\varphi)$	anti-projection (column dropping)
	$\mu(X = \Psi)$	fixpoint term

Figure 1. Grammar of μ -RA [11].

Like in RA, the data model in μ -RA consists of relations that are sets of tuples which associate column names to values. For instance, the tuple $\{src \rightarrow 1, dst \rightarrow 2\}$ is a member of the relation S of Fig. 2.

Let us consider a directed and rooted graph G , a relation E that represents the edges in G , and a relation S of starting edges (a subset of edges in E that start from the graph root nodes), as represented in Fig. 2. The following examples illustrate how μ -RA algebraic terms can be used to model graph operations, such as navigating through a sequence of edges in a graph:

Example 1. The term $\tilde{\pi}_c(\rho_{trg}^c(S) \bowtie \rho_{src}^c(E))$ returns pairs of nodes that are connected by a path of length 2 where the first element of the pair is a graph root node. For that purpose, the relation S is joined (\bowtie) with the relation E on the common

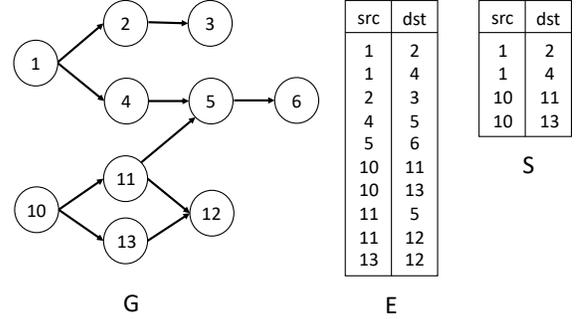


Figure 2. Graph example.

column c , after proper renaming (ρ) to ensure that c represents both the target node of S and the source node of E . After the join, the column c is discarded by the anti-projection ($\tilde{\pi}_c$) so as to keep only the two columns src , dst in the result relation.

Example 2. Now, the recursive term $\mu(X = S \cup \tilde{\pi}_c(\rho_{trg}^c(X) \bowtie \rho_{src}^c(E)))$ computes the pairs of nodes that are connected by a path in G starting from edges in S .

The subterm $\varphi = \tilde{\pi}_c(\rho_{trg}^c(X) \bowtie \rho_{src}^c(E))$ computes new paths by joining X (the previous paths) and E such that the destinations of X are equal to the sources of E .

The fixpoint is computed in 4 steps where X_i denotes the value of the recursive variable at step i :

$$\begin{aligned}
X_0 &= \emptyset \\
X_1 &= \left\{ \{src \rightarrow 1, dst \rightarrow 2\}, \{src \rightarrow 1, dst \rightarrow 4\}, \right. \\
&\quad \left. \{src \rightarrow 10, dst \rightarrow 11\}, \{src \rightarrow 10, dst \rightarrow 13\} \right\} \\
X_2 &= X_1 \cup \left\{ \{src \rightarrow 1, dst \rightarrow 3\}, \{src \rightarrow 1, dst \rightarrow 5\}, \right. \\
&\quad \left. \{src \rightarrow 10, dst \rightarrow 5\}, \{src \rightarrow 10, dst \rightarrow 12\} \right\} \\
X_3 &= X_2 \cup \left\{ \{src \rightarrow 1, dst \rightarrow 6\}, \{src \rightarrow 10, dst \rightarrow 6\} \right\} \\
X_4 &= X_3 \quad (\text{fixpoint reached})
\end{aligned}$$

At step 1 it is empty, at step 2 it is a relation of two columns src and trg that contains four rows, and the iteration continues until the fixpoint is reached.

B. Semantics and properties of the fixpoint

The semantics of a μ -RA term is defined by the relation obtained after substituting the free variables in the term (like E and S in example 2) by their corresponding database relations. The notions of free and bound variables and substitution are formally defined in [11].

As an slight abuse of notation, we sometimes use a recursive term Ψ (i.e. a term that contains a recursive variable X) as a function $R \rightarrow \Psi(R)$ that takes a relation R and returns the relation obtained by replacing X in the term Ψ by the relation R . In the above example

$$\varphi(S) = \tilde{\pi}_c(\rho_{trg}^e(S) \bowtie \rho_{src}^e(E)) = \left\{ \{src \rightarrow 1, dst \rightarrow 3\}, \{src \rightarrow 1, dst \rightarrow 5\}, \{src \rightarrow 10, dst \rightarrow 5\}, \{src \rightarrow 10, dst \rightarrow 12\} \right\}.$$

Under this notation, $\mu(X = \Psi)$ is defined as the fixpoint F of the function Ψ , so $\Psi(F) = F$.

Let us consider the following conditions (denoted F_{cond}) for a fixpoint term $\mu(X = \Psi)$ (as also considered in [11]).

- *positive*: for all subterms $\varphi_1 \triangleright \varphi_2$ of Ψ , φ_2 is constant in X (i.e. X does not appear in φ_2);
- *linear*: for all subterms of Ψ of the form $\varphi_1 \bowtie \varphi_2$ or $\varphi_1 \triangleright \varphi_2$, either φ_1 or φ_2 is constant in X ;
- *non mutually recursive*: when there exists a subterm $\mu(Y = \psi)$ in Ψ , then any occurrence of X in this subterm should be inside a term of the form $\mu(X = \gamma)$.

These conditions guarantee the following properties (see [11]):

Proposition 1. *If $\mu(X = \Psi)$ satisfies F_{cond} then*

$$\Psi(S) = \Psi(\emptyset) \cup \bigcup_{x \in S} \Psi(\{x\})$$

and thus Ψ has a fixpoint with $\mu(X = \Psi) = \Psi^\infty(\emptyset)$.

Proposition 2. *Every fixpoint term $\mu(X = \Psi)$ that satisfies F_{cond} can be written like the following: $\mu(X = R \cup \varphi)$ where R is constant in X and $\varphi(\emptyset) = \emptyset$. R is called **the constant part** of the fixpoint and φ **the variable part**.*

In the rest of the paper, we only consider fixpoint terms satisfying the conditions F_{cond} in their decomposed form $\mu(X = R \cup \varphi)$ since their existence is guaranteed thanks to proposition 1.

C. Evaluation of the fixpoint

A fixpoint term can be evaluated with the algorithm:

Algorithm 1

```

1   X = R
2   new = R
3   while new ≠ ∅:
4     new = φ(new) \ X
5     X = X ∪ new
6   return X

```

The fixpoint is obtained by evaluating φ repeatedly starting from $X = R$ until the fixpoint is reached. In this algorithm, we apply φ on the new results only (obtained by making a set difference between the current result and the previous one) instead of the entire result set. This is possible thanks to the property of φ stated in proposition 1, which implies that $\varphi(X_i) \cup \varphi(X_{i+1}) = \varphi(X_i) \cup \varphi(X_{i+1} \setminus X_i)$.

Evaluating recursive computations on the new iteration results is well known in the context of Datalog [8] and transitive closure evaluation [17] with the semi-naive (or differential) approach (see Section VI on related works for more discussion on these aspects).

III. ARCHITECTURE OF THE DIST- μ -RA SYSTEM

The Dist- μ -RA system takes a query as input parameter, translates it into μ -RA, optimizes it, and then performs the evaluation in a distributed fashion on top of Spark. For this purpose the Dist- μ -RA system is composed of several components, as illustrated in Fig. 3.

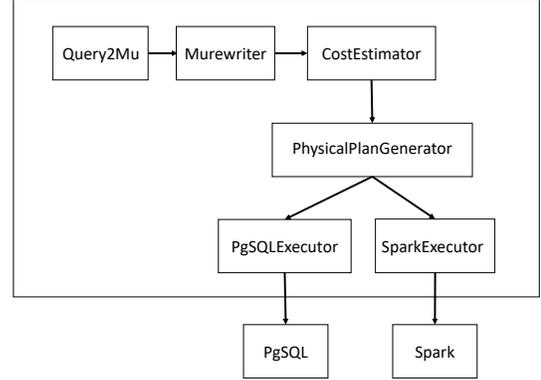


Figure 3. Architecture of the Dist- μ -RA system.

The Query2Mu component translates UCRPQs into μ -RA terms. For sure, Dist- μ -RA also accepts more general μ -RA-terms that are not expressible as UCRPQs¹, as long as they satisfy the triple condition F_{cond} mentioned in Section II.

From a given input μ -RA term, the MuRewriter explores the space of semantically equivalent logical plans by applying a number of rewrite rules. In addition to the rewrite rules already known in classical relational algebra, MuRewriter applies a set of rules specific to the fixpoint operator. These rules and the conditions under which they are applicable are formally defined in [11].

The evaluation cost of these terms are estimated by the CostEstimator component, based on data cardinality estimations proposed in [18], and that outputs a best estimated logical plan for recursion.

From a given recursive logical plan, the PhysicalPlanGenerator generates a physical plan for distributed execution. It can also generate a centralized plan for PostgreSQL, for the dual purpose of (i) efficiently executing certain subqueries in a centralized manner and for (ii) performance comparison with the centralized case, in particular with [11].

The PhysicalPlanGenerator is an original contribution of this paper and it is described in more detail in Section IV.

We describe below steps that sample queries go through for evaluation in the Dist- μ -RA system. Consider for instance the

¹See the practical experiments section for some examples such as the “same generation” query.

following UCRPQ composed of a conjunction of two RPQs:

$$?a, ?b, ?c \leftarrow ?a \text{ wasBornIn/IsLocatedIn+ Japan,} \\ ?b \text{ isConnectedTo+ ?c}$$

The first RPQ computes the people $?a$ that are born in a place that is located directly or indirectly in Japan. The query is first translated into μ -RA by `Query2Mu` so that `MuRewriter` can generate semantically equivalent plans. We describe below the rewrite rules specific to fixpoint terms leveraged from [11] that can apply in `MuRewriter`, and we give the intuition of their effect on performance:

- *Pushing filters into fixpoints*: with this rule, the query $?x \text{ isLocatedIn+ Japan}$ is evaluated as a fixpoint starting from $?x$ such as $?x \text{ isLocatedIn Japan}$, which avoids the computation of the whole `isLocated+` relation followed by the filter `Japan`.
- *Pushing joins into fixpoints*: let us consider the query $?x \text{ isMarriedTo/knowns+ ?y}$. Instead of computing the relation `knowns+` and joining it with `isMarriedTo`, this rule rewrites the fixpoint such that it starts from $?x$ and $?y$ that verify $?x \text{ isMarriedTo/knowns ?y}$. The application of this rule is beneficial in this case because the size of the `isMarriedTo/knowns` relation is usually smaller than the size of the `knowns` relation.
- *Merging fixpoints*: when evaluating $?x \text{ isLocatedIn+/dealsWith+ ?y}$, instead of computing both fixpoints separately then joining them, this rule generates a single fixpoint that starts with `isLocated/dealsWith` then recursively appends either `isLocatedIn` to the left or `dealsWith` to the right.
- *Pushing antiprojections into fixpoints*: this rule gets rid of unused columns during the fixpoint computations. For instance, the query $?y \leftarrow ?x \text{ isLocatedIn+ ?y}$ (this query asks for $?y$ only) is evaluated by starting only from the destinations $?y$ of the `isLocatedIn` relation and by recursively getting the new destinations, thus avoiding to keep the pairs of nodes $?x$ and $?y$ then discarding $?x$ at the end.
- *Reversing a fixpoint*: the fixpoint corresponding to the relation $a+$ can either be computed from left to right by starting from a and by recursively appending a to the right of the previously found results, or from right to left by starting from a and appending a to the left. Reversing a fixpoint consists in rewriting from the first form to the other or vice versa. This rule is necessary to account for all possible filters and joins that can be pushed in a fixpoint. For instance, a filter that is located at the left side of $a+$ can only be pushed if the fixpoint is evaluated from left to right.

After these transformations, the best (estimated) recursive logical plan selected by `CostEstimator` is given as input parameter to `PhysicalPlanGenerator` that is in charge

of generating the best physical plans for distributed execution, and which we now further describe in the next section.

IV. DISTRIBUTED EVALUATION

Non-recursive μ -RA expressions are directly translated into Spark [3] operations using the `Dataset API`. Datasets are used to represent relational data in Spark. The optimization of these expressions is delegated to Spark’s Catalyst internal optimizer [19] before execution. We now describe how fixpoint terms are evaluated in a distributed manner, first by explaining the principles and then how physical plans are generated.

A. Principles of the distributed evaluation of the fixpoint

In this section we present two ways of distributing the fixpoint computation on a Spark cluster:

1) *Global Loop on the Driver* (\mathcal{P}_{gld}): In order to execute the fixpoint in a distributed setting, a simple way is to distribute the computations performed at each iteration of the Algorithm 1. In Spark, the so-called *driver*² performs the loop and, at each iteration, instructions at lines 4 and 5 are executed as `Dataset` operations that are distributed among the workers. We call this execution plan \mathcal{P}_{gld} . On Spark, \cup is executed as `Dataset union` followed by a `distinct` operation. This means that in \mathcal{P}_{gld} , at least one data transfer (shuffle) per iteration is made to perform the union.

2) *Parallel Local loops on the Workers* (\mathcal{P}_{plw}): Another way to distribute the fixpoint is based on the following observation: **Proposition 3.** *Under the conditions F_{cond} , we have:*

$$\mu(X = R_1 \cup R_2 \cup \varphi) = \mu(X = R_1 \cup \varphi) \cup \mu(X = R_2 \cup \varphi)$$

which is a consequence of proposition 1. Thanks to this property 3, the fixpoint can be executed by distributing the constant part R among the workers, then each worker i executes a smaller fixpoint $\mu(X = R_i \cup \varphi)$ locally starting from its own constant part R_i . We call this execution plan \mathcal{P}_{plw} . As opposed to \mathcal{P}_{gld} , \mathcal{P}_{plw} incurs only one data shuffle at the end to make the distinct union between the local fixpoints.

In Example 2, if we split the start edges S among two workers by giving the first (1, 2) and (10, 11) and the second (1, 4) and (10, 13), after executing \mathcal{P}_{plw} , the first worker will find the paths (1, 3), (10, 5), (10, 6) and (10, 12) and the second will find (1, 5), (1, 6), and (10, 12).

Data distribution for \mathcal{P}_{plw} : there are cases where the final data shuffle incurred by \mathcal{P}_{plw} can also be avoided by properly repartitioning input data among workers. We first give the intuition, then we give the proof below.

We look for a column col (or a set of columns) in X that is left unchanged by φ . In other words, a tuple in R having a value v

²The *driver* is the process that creates tasks and send them to be executed in parallel by *worker* nodes.

at column col will only generate tuples having the same value at this column throughout the iterations of the fixpoint. So if we put all tuples in R having v at column col in one worker, no other worker will generate a tuple with this value at that column.

For this, we use the stabilizer technique defined in Definition 10 of [11] and used to push filters in fixpoint expressions. It consists of computing “the set of columns which are not altered during the fixpoint iteration”. For instance, ‘src’ is a stable column in the fixpoint expression of example 2 meaning that tuples in the fixpoint having ‘src’ = 1 can only be produced from tuples in S having ‘src’ = 1, which implies that filtering tuples having ‘src’ = 1 before or after the fixpoint computation lead to the same results. However, this is not true for the column ‘dst’ which is not stable.

To summarize, when the constant part of the fixpoint is repartitioned by the stable column (or columns) prior to the fixpoint execution, we know that there will be no duplicate across the workers (so we can avoid calling distinct at the end of the computation).

For instance, repartitioning the constant part S in example 2 by src will result in the paths (1, 3), (1, 5), and (1, 6) being found in one worker and the paths (10, 5), (10, 6) and (10, 12) in the second, thus avoiding the the duplicate (10, 12) between the two workers.

Proof. Let c be a stable column of $\mu(X = R \cup \varphi)$, which means that $\forall e \in \mu(X = R \cup \varphi) \exists r \in R e(c) = r(c)$ [11]. In μ -RA, an element r in R is a mapping (tuple), which means that it is a function that takes a column name and returns the value that r has at that column.

Let us consider a partitioning R_1, \dots, R_n of R by the column c which verifies the following

$$\forall i \neq j \in \{1..n\} \forall a \in R_i \forall b \in R_j a(c) \neq b(c)$$

This statement means that there are no two elements of R at different partitions that share the same value at column c . We next show that this statement is also true for the fixpoint term.

Let $i \neq j \in \{1..n\}$ and let $x \in \mu(X = R_i \cup \varphi)$ and $y \in \mu(X = R_j \cup \varphi)$. Since c is stable, we have $\exists a \in R_i x(c) = a(c)$ and $\exists b \in R_j y(c) = b(c)$. So $x(c) \neq y(c)$.

In conclusion, the sets $\mu(X = R_i \cup \varphi)$ where $i \in \{1..n\}$ are disjoint. \square

B. Physical plan generation

We first present the different physical plans for the μ -RA operators, and then we explain how they are selected.

a) The fixpoint operator: As mentioned earlier, there are two different ways of evaluating the fixpoint operator in a distributed manner.

\mathcal{P}_{plw} has two implementations (physical plan alternatives):

- \mathcal{P}_{plw}^{pg} : The local fixpoints are executed on PostgreSQL. The fixpoint operator is performed as a Spark mapPartition operation where each worker performs a part of the fixpoint computation on PostgreSQL. A PostgreSQL instance runs on each worker. The part of data assigned to each worker is represented as a view in the PostgreSQL instance running on this worker, and the μ -RA expression (that computes the fixpoint) is translated to a PostgreSQL query that is executed using this view as the constant part of the fixpoint. Each PostgreSQL executor returns its result as an iterator which is then processed by Spark.
- \mathcal{P}_{plw}^s : The fixpoint computation is implemented using a loop in the driver that uses Spark operations to compute the recursive part of the fixpoint. These Spark operations are written in such a way each worker performs its own fixpoint independently, so without data being sent across the cluster. For example, to perform a join, the Spark broadcast join is used. To perform the union (or set-difference), a special union (set-difference) operation is used that computes the union (set-difference) partition-wise. These special union and set-difference operations are implemented as part of the SetRDD API. SetRDD is a special RDD³ where each partition is a set. This SetRDD is used to store the value of the recursive variable X at each iteration. This means that each partition of X holds the intermediate results of the local fixpoint performed by the worker to which this partition has been assigned. This SetRDD technique is inspired from BigDatalog [20].

\mathcal{P}_{gld} is implemented using a global loop in the driver. At each step of the recursion a distinct operation is performed. Intermediate data is stored in a Spark Dataset.

Fig 4 illustrates the dataflow in each physical plan \mathcal{P}_{plw}^s , \mathcal{P}_{plw}^{pg} and \mathcal{P}_{gld} . In \mathcal{P}_{plw}^s , workers use only their own partitions of the SetRDDs R (constant part) and X (recursive part) and communicate their results to the master. Similarly in \mathcal{P}_{plw}^{pg} , workers only communicate with their local PostgreSQL instance and send their final result to the master. Whereas in \mathcal{P}_{gld} , the Datasets X and R get shuffled at each iteration to perform union, set-difference and joins.

b) Other operators: An operator in μ -RA has a local version implemented using PostgreSQL and a distributed version implemented using the Spark Dataset API. Some operators may have more than one distributed execution plan. For instance, for the join operator, we choose which argument (if any) to broadcast in order to guide Spark on whether to use broadcast join or another type of join.

c) Plan selection: while generating the physical plan for the fixpoint operator, we check whether there is a stable column (Sec. IV-A2). If so, we first shuffle data according to this

³RDD is an abstraction that Spark provides to represent a distributed collection of data. An RDD is split among partitions which are assigned to workers.

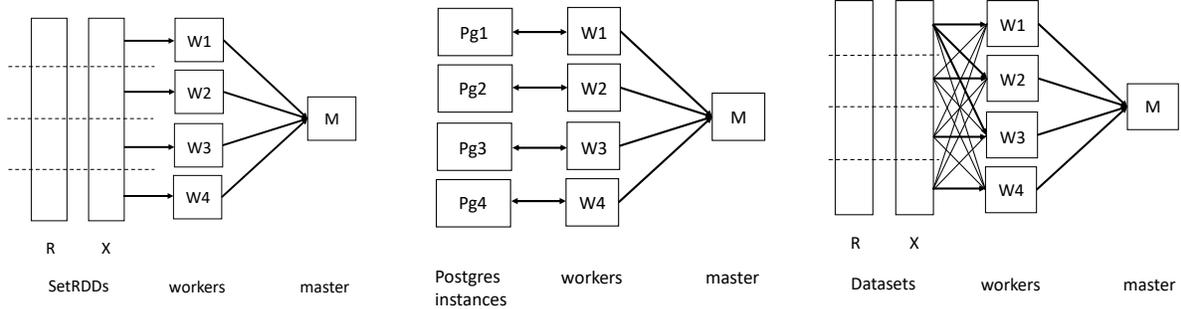


Figure 4. Fixpoint physical plans dataflow. From left to right: \mathcal{P}_{plw}^s , \mathcal{P}_{plw}^{pg} , \mathcal{P}_{gld}

column and execute the plan \mathcal{P}_{plw} . If not, the \mathcal{P}_{gld} plan is selected.

When \mathcal{P}_{plw}^s is chosen, joins are executed as broadcast joins (all relations in the variable part of the fixpoint apart from the recursive relation are broadcasted), antiprojections are executed without the need of applying the distinct operation. If \mathcal{P}_{plw}^{pg} is chosen, then the operators in the fixpoint expression are executed using the local PostgreSQL instances.

V. EXPERIMENTS

We evaluate the performance of a prototype implementation of the Dist- μ -RA system on top of the Spark platform. We extensively compared the performance against other state-of-the-art systems on various datasets and queries. We report below on these experiments.

A. Experimental setup

Experiments have been conducted on a Spark cluster composed of 4 machines (hence using 4 workers, one on each machine, and the driver on one of them). Each machine has 40 GB of RAM, 2 Intel Xeon E5-2630 v4 CPUs (2.20 GHz, 20 cores each) and 66 TB of 7200 RPM hard disk drives, running Spark 2.4.5 and Hadoop 2.8.4 inside Debian-based Docker containers.

B. Datasets

We use real and synthetic datasets of different sizes and topological properties, as summarized in Table I. We consider the following real graphs:

- Yago⁴: A knowledge graph extracted from Wikipidia and other sources [21].
- The Facebook, Reddit, DBLP and Live Journal real word datasets from the Snap network dataset collection [22].

⁴We use a cleaned version of the real world dataset Yago 2s, that we have preprocessed in order to remove duplicate RDF [23] triples (of the form <source, label, target>) and keep only triples with existing and valid identifiers. After preprocessing, we obtain a table of Yago facts with 83 predicates and 62,643,951 rows (graph edges).

Dataset	Edges	Nodes
Yago [21]	62,643,951	42,832,856
Facebook [22]	88,234	4,039
Reddit [22]	858,490	55,863
DBLP [22]	1,049,866	317,080
Live Journal [22]	68,993,773	4,847,571

Dataset	Edges	Nodes	TC size
rnd_10k_0.001	50,119	10,000	5,718,306
rnd_20k_0.001	199,871	20,000	81,732,096
rnd_30k_0.001	450,904	30,000	255,097,974
rnd_10k_0.005	249,791	10,000	39,113,982
rnd_50k_0.001	1,250,922	50,000	906,630,823
tree_10	9,999	10,000	84,615
tree_150	149,999	150,000	1,775,161
uniprot_1M	1,000,443	1,017,828	–
uniprot_5M	5,001,427	5,081,402	–
uniprot_10M	10,001,920	10,153,411	–

Table I
REAL AND SYNTHETIC GRAPHS.

We consider the synthetic graphs defined as follows:

- rnd_n_p: random graphs generated with the Erdos Renyi algorithm, where n is the number of nodes in the graph and p the probability that two nodes are connected.
- uniprot_n: a benchmark graph of n nodes generated using the gMark benchmark tool [24]. It models the Uniprot database of proteins [25].
- tree_n: a random tree of n nodes generated recursively as follows: tree_1 is a tree of 1 node, and then tree_ i + 1 is a tree of i + 1 nodes where the i^{th} + 1 node is connected as a child of a randomly selected node in tree_ i .
- Other graphs derived from rnd_p_n by adding a set of predefined labels randomly. They are used to compute terms that require labeled graphs like the $a^n b^n$ query and the concatenated closure queries (Sec. V-D).

C. Systems

We compare Dist- μ -RA with the following systems:

- BigDatalog [20]: a large-scale distributed Datalog engine built on top of Spark.

- GraphX [7]: a Spark library for graph computations. It exposes the Pregel API for recursive computations. In order to compare our system with GraphX we need to convert UCRPQs to GraphX programs⁵. Specifically, we compute a regular graph query by making each node send a message to its adequate neighbors in such a way that the query pattern is traversed recursively from left to right. This means that for a query that starts by selection ($?x \leftarrow A \text{ pattern } ?x$), only the node A sends a message at the start of the computation.
- Myria [26]: A distributed big data management system that supports Datalog and exposes MyriaL, a query language similar to Datalog. We were not able to run this system on our cluster because the software stack has evolved and Myria is not maintained anymore. However, we were able to test it on a local configuration of four workers on a single machine.
- Centralized μ -RA [11]: Implementation of μ -RA on PostgreSQL that runs locally on a single machine.

D. Queries

Knowledge graphs like Uniprot and Yago are queried using UCRPQs. We also provide additional experiments using more general μ -RA terms not expressible by UCRPQs because they are not regular (e.g. same generation example).

a) Yago queries: Fig. 5 lists the UCRPQs evaluated on the Yago dataset. Queries Q_1 to Q_7 are taken from [27], Q_8, Q_9 from [28], and Q_{10}, Q_{11} from [29]. We have added the rest of the queries to illustrate larger transitive closures.

?x	\leftarrow	?x isMarriedTo/livesIn/IsL+/dw+ Argentina	Q_1
?x	\leftarrow	?x hasChild/livesIn/IsL+/dw+ Japan	Q_2
?x	\leftarrow	?x influences/livesIn/IsL+/dw+ Sweden	Q_3
?x	\leftarrow	?x livesIn/IsL+/dw+ United_States	Q_4
?x	\leftarrow	?x hasSuccessor/livesIn/IsL+/dw+ India	Q_5
?x	\leftarrow	?x hasPredecessor/livesIn/IsL+/dw+ Germany	Q_6
?x	\leftarrow	?x haa/livesIn/IsL+/dw+ Netherlands	Q_7
?x	\leftarrow	?x IsL+/dw+ United_States	Q_8
?x	\leftarrow	?x (actedIn/-actedIn)+ Kevin_Bacon	Q_9
?area	\leftarrow	wce -type/(IsL+/dw/dw) ?area	Q_{10}
?person	\leftarrow	?person isMarriedTo+owns/IsL+/owns/IsL+ USA	Q_{11}
?a,?b	\leftarrow	?a IsL+/dw ?b	Q_{12}
?a,?b	\leftarrow	?a IsL+/dw+ ?b	Q_{13}
?a,?b,?c	\leftarrow	?a wasBornIn/IsL+ ?b, ?b isConnectedTo+ ?c	Q_{14}
?a,?b,?c	\leftarrow	?a (IsL isConnectedTo)+ ?b, ?a wasBornIn ?c	Q_{15}
?a,?b,?c	\leftarrow	?a wasBornIn/IsL+ Japan, ?b isConnectedTo+ ?c	Q_{16}
?a	\leftarrow	?a IsL+/(isConnectedTo dw)+ Japan	Q_{17}
?a,?c	\leftarrow	?a IsL+ Japan, ?a isConnectedTo+ ?c	Q_{18}
?a	\leftarrow	?a IsL+/IsL Japan	Q_{19}
?a	\leftarrow	?a IsL+/isConnectedTo+/dw+ Japan	Q_{20}
?a,?b	\leftarrow	?a (isL dw) rdfs:subClassOf isConnectedTo)+ ?b	Q_{21}
?a	\leftarrow	?a (isConnectedTo/-isConnectedTo)+ Shannon_Airport	Q_{22}
?a	\leftarrow	?a (wasBornIn/isL/-wasBornIn)+ John_Lawrence_Toole	Q_{23}
?a	\leftarrow	Jay_Kapraff (livesIn/isL/-livesIn)+?x	Q_{24}
?a,?b	\leftarrow	?a (actedIn/-actedIn)+hasChild+ ?b	Q_{25}

Figure 5. Queries for the YAGO dataset⁶.

⁵In the GraphX framework, a recursive computation is composed of “supersteps” where, in each superstep, graph nodes send messages to their neighbor nodes, then a merge function aggregates messages per recipient and each recipient receives its aggregated messages in order to process them. A computation is stopped when no new message is sent.

⁶ “isL” stands for “IsLocatedIn”, “dw” for “dealsWith”, “haa” for “hasAcademicAdvisor”, and “wce” for “wikicat_Capitals_in_Europe”.

b) Concatenated closures: are queries of the form $a_1+a_2+...+a_n+$ where $2 \leq n \leq 10$.

c) μ -RA queries: the tested μ -RA terms are the following:

- $a^n b^n$: computes 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 . It is expressed by the following μ -RA term:

$$\mu(X = \tilde{\pi}_m(\rho_{trg}^m(\sigma_{pred=a}(R)) \bowtie \rho_{src}^m(\sigma_{pred=b}(R))) \cup \tilde{\pi}_m(\tilde{\pi}_n(\rho_{trg}^m(\sigma_{pred=a}(R)) \bowtie \rho_{trg}^n(\rho_{src}^m(X)) \bowtie \rho_{src}^n(\sigma_{pred=b}(R))))))$$

- Same Generation: computes the pairs of nodes that are of the same generation in a graph. The graph edges R represent the parent relation. We use the following term to express it:

$$\mu(X = \tilde{\pi}_m(\rho_{src}^m(R) \bowtie \rho_{src}^m(R)) \cup \tilde{\pi}_m(\tilde{\pi}_n(\rho_{src}^m(R) \bowtie \rho_{trg}^n(\rho_{src}^m(X)) \bowtie \rho_{src}^n(R))))$$

- TC: computes the transitive closure of a graph:

$$\mu(X = R \cup \tilde{\pi}_m(\rho_{trg}^m(X) \bowtie \rho_{src}^m(R)))$$

- Reach: computes the reachable nodes in a graph from a source node N (chosen randomly from each test file).

$$\sigma_{src=N}(\tilde{\pi}_{src}(\mu(X = R \cup \tilde{\pi}_m(\rho_{trg}^m(X) \bowtie \rho_{src}^m(R))))))$$

d) Uniprot queries: In order to test the comparative behavior of Dist- μ -RA on the Uniprot benchmark graphs, we have made UCRPQ queries of various shapes (see Fig. 6). Each tested query (fig. 7) has at least one of those shapes. For instance, the query $?x, ?y \leftarrow ?x \text{ interacts}/(\text{encodedOn}/-\text{encodedOn})+ ?y$ has the shape $?x, ?y \leftarrow ?x \text{ a/b}+ ?y$.

?x,?y	\leftarrow	?x a+ ?y
?x,?y	\leftarrow	?x a/b+ ?y
?x,?y	\leftarrow	?x a+/b ?y
?x,?y	\leftarrow	?x a+/b+ ?y
?x,?y	\leftarrow	?x a+/b/c+ ?y
?x,?y	\leftarrow	?x a/b+/c ?y
?x,?y	\leftarrow	?x a/b/c+ ?y
?x,?y	\leftarrow	?x a/b/c/d+ ?y
?x	\leftarrow	?x a+ C
?x	\leftarrow	C a+ ?x
?x	\leftarrow	C a/b+ ?x
?x,?y,?z,?t	\leftarrow	?x a+ ?y, ?x b+ ?z, ?x c ?t
?x,?y	\leftarrow	?x (a b)+ ?y, C c+ ?y
?x	\leftarrow	?x a+/b ?y, C d+ ?y
?x	\leftarrow	?x a+/b ?y, C c/d+ ?y
?x,?y	\leftarrow	?x a+/(b c) ?y
?x	\leftarrow	?x (a b)+ C
?x	\leftarrow	C a/(b c)+ ?x

Figure 6. Uniprot query shapes.

E. Results

We report and comment on experimental results. Absence of a time in a figure means that the query evaluation has failed (the system has crashed).

1) Evaluation of UCRPQs on Yago:

⁷ “int” stands for “interacts”, “enc” for “encodes”, “occ” for “occurs”, “hKw” for “hasKeyword”, “ref” for “reference”, “auth” for “authoredBy”, and “pub” for “publishes”.

2x, ?y	↑	2x -hKw/(ref/-ref)+ ?y	Q ₁
2x, ?y	↑	2x -hKw/(enc/-enc)+ ?y	Q ₂
2x, ?y	↑	2x -hKw/(occ/-occ)+ ?y	Q ₃
2x, ?y	↑	2x int/(enc/-enc)+ ?y	Q ₄
2x, ?y	↑	2x int/(occ/-occ)+ ?y	Q ₅
2x, ?y	↑	2x int+/(occ/-occ)+ ?y	Q ₆
2x, ?y	↑	2x int+/(enc/-enc)+ ?y	Q ₇
2x, ?y	↑	2x int+/(occ/-occ)+/(hKw/-hKw)+ ?y	Q ₈
2x, ?y	↑	2x -hKw/int/ref/(auth/-auth)+ ?y	Q ₉
2x, ?y	↑	2x (enc/-enc)+hKw ?y	Q ₁₀
2x	↑	2x (enc/-enc)+ C	Q ₁₁
2x, ?y, ?z, ?t	↑	2x (enc/-enc)+ ?y, ?x int+ ?z, ?x ref ?t	Q ₁₂
2x, ?y	↑	2x (int (enc/-enc))+ ?y, C (occ/-occ)+ ?y	Q ₁₃
2x	↑	2x int+ref ?y, C (auth/-auth)+ ?y	Q ₁₄
2x	↑	2x int+ref ?y, C -pub/(auth/-auth)+ ?y	Q ₁₅
2x	↑	2x (enc/-enc)+hKw ?x	Q ₁₆
2x, ?y	↑	2x -occ/int+occ ?y	Q ₁₇
2x, ?y	↑	2x (-ref/ref)+ ?y	Q ₁₈
2x, ?y	↑	2x int/ref/(-ref/ref)+ ?y	Q ₁₉
2x	↑	2x (ref/-ref)+ ?x	Q ₂₀
2x, ?y	↑	2x (-ref/ref)+/(auth pub) ?y	Q ₂₁
2x	↑	2x (enc/-enc occ/-occ)+ C	Q ₂₂
2x	↑	2x int/(enc/-enc occ/-occ)+ ?x	Q ₂₃
2x	↑	2x (enc/-enc)+ ?x	Q ₂₄
2x	↑	2x (occ/-occ)+ ?x	Q ₂₅

Figure 7. Uniproton queries⁷.

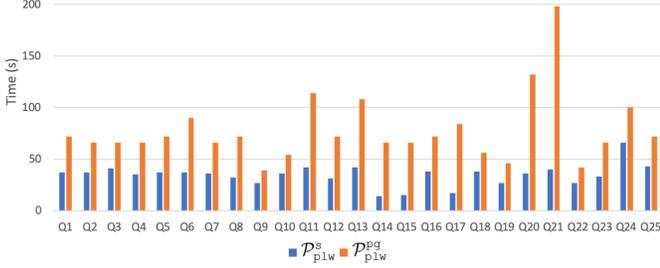


Figure 8. \mathcal{P}_{plw} implementations running times on Yago

a) *Comparison of \mathcal{P}_{plw} implementations* : We compare the two implementations of the plan \mathcal{P}_{plw} described in Sec. IV-A on the Yago dataset. Results in Fig. 8 show that the implementation using SetRdd is faster. Therefore, we use this implementation as a basis of comparisons in the rest of the experiments.

b) *Comparison with other systems*: Fig. 9 shows the execution times. The comparison between centralized μ -RA and Dist- μ -RA shows that 17 out of the 25 tested queries (these are typically queries which handle more intermediate data) execute faster when distributed. In μ -RA with \mathcal{P}_{gld} , all fixpoints in the queries are executed with the \mathcal{P}_{gld} plan. Comparison with Dist- μ -RA shows the benefit of the \mathcal{P}_{plw} plan for executing fixpoints.

Dist- μ -RA and BigDatalog times are very similar except for queries that handle big intermediate results and that have been optimized by the μ -RA compiler. For instance, in Q_9 , the TC (actedIn/-actedIn)+ contains 30 million records approximately. However, the Dist- μ -RA program does not need to compute this entire TC because the compiler pushes the filter "Kevin Bacon" in the constant part of the fixpoint. BigDatalog is built on a datalog engine that is capable of pushing filters, but this optimisation has apparently not been performed on the Datalog program that computes Q_9 . A similar observation can be made for Q_{25} where two fixpoints are merged in Dist- μ -RA. This shows that for queries handling large amounts of data, the plan chosen for executing the query has a considerable impact on execution time.

GraphX is much slower overall except for queries like Q_{10} and Q_{24} where filtering is performed at the beginning of the query (as mentioned in Sec. V-C). We believe that this lack of performance is due to the fact that, in the GraphX Pregel model, each node has to keep track of its ancestors that satisfy a given regular path query (or a part of it) and transmit this information to their successors in order to get the pairs of nodes satisfying the whole query. So while GraphX has been proven to be efficient for a number of graph algorithms [7], it can be less suitable for this kind of queries.

2) *Concatenated closures*: We evaluate concatenated closure queries on the graph obtained from rnd_100k_0.001 by randomly assigning 10 different labels to its edges. Results are shown in Fig. 10. Dist- μ -RA is faster on all queries. For queries with more concatenations, the time difference between Dist- μ -RA and other systems gets larger. BigDatalog fails from the fifth query and GraphX fails on all queries.

The plans that were selected in Dist- μ -RA for the execution of these queries apply a mixture of the rewritings that "push joins" and "merge fixpoints" (see Sec. III).

3) *μ -RA queries*: Execution times are shown in Fig. 11. Dist- μ -RA and BigDatalog have comparable execution times. We compare Myria and Dist- μ -RA on the Same Generation query (Fig. 12): Dist- μ -RA is significantly faster on all queries.

4) *Evaluation of UCRPQs on Uniproton*: the execution times (on the cluster) are shown in Fig 13. Dist- μ -RA answers all of the queries and is faster. The comparison with Myria (on a single machine) is shown in Fig 14.

We perform further scalability tests by measuring Dist- μ -RA and BigDatalog execution times for each Uniproton query on the generated uniproton_n graphs with varying sizes of 1M, 5M and 10M edges. Fig. 15 shows the execution times for Dist- μ -RA in comparison to BigDatalog. BigDatalog fails in 44 cases out of a total of 75 query evaluations. Dist- μ -RA answers all of them and scales better.

For more comprehensive testing, queries and graph sizes have been selected so as to cover a wide range of result sizes. Q_{15} is one of the queries with the smallest result size (14,000 results for uniproton_10M) and Q_{21} is one with the largest (around 1.5 billion results for uniproton_10M, which is 150 times the size of the graph).

VI. RELATED WORKS

The Dist- μ -RA system evaluates expressive queries (such as UCRPQs) over graphs. To achieve this goal, it is essential for a system to be able to: (i) support recursion and the optimization of recursive terms, and (ii) provide distribution of data and computation. We examine and compare to the closest related works along these aspects below.

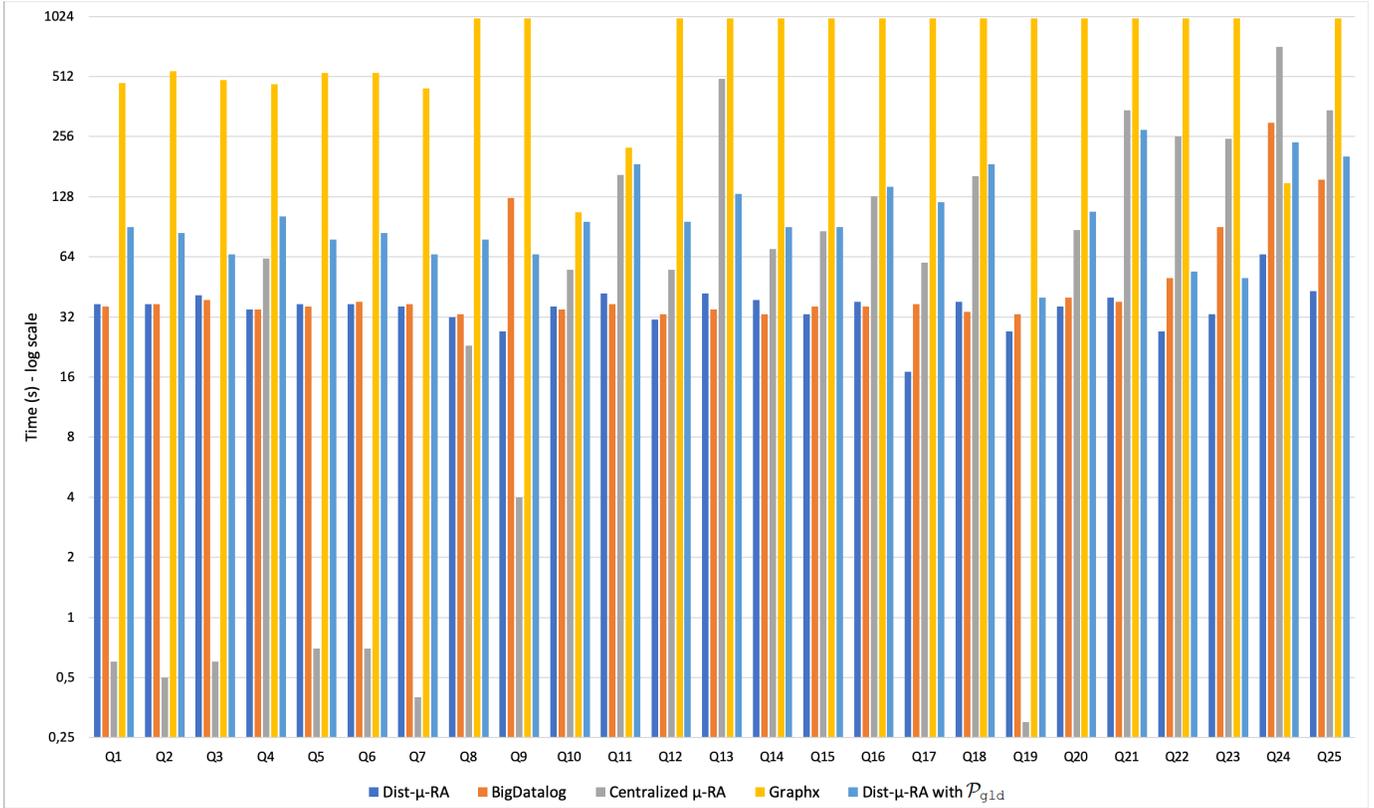


Figure 9. Running times on Yago. A timeout is set at 1,000 s.

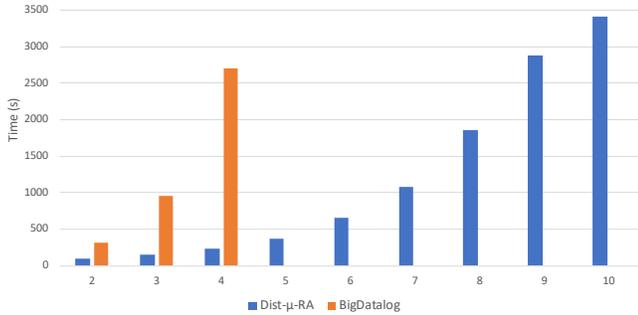


Figure 10. Evaluation times for concatenated closure queries.

A. Logical recursion optimization

The ability to express recursive queries has been studied in various formalisms including RA, Datalog, and automata as an ad-hoc technique to evaluate regular queries like UCRPQs on graphs.

The system we present in this paper is based on the μ -RA algebra, an extension of RA with a fixpoint operator introduced in [11] which offers a good balance between expressivity, possible optimizations, and yields state-of-the-art performance for evaluating UCRPQs in a centralized setting with PostgreSQL [11]. Earlier works tried to extend RA with

recursion, see [11] and references thereof for more details.

In Datalog, Magic Sets [30], [31], recently improved by Demand Transformation [32], is a well-known optimization technique for recursive Datalog programs. The optimizations provided by Magic Sets or Demand Transformations are equivalent to pushing selections and projections in μ -RA. However, there is no equivalent to merging fixpoints. Furthermore, depending on the way the Datalog program is written, some optimizations may or may not be applied. For instance a left-linear DL program (e.g. $P(x, y) \leftarrow P(x, z), R(z, y)$) cannot push filters that are applied on the right side (on y in the example). The optimization framework has then to be coupled with a technique for reversing DL programs as proposed in [33]. Since Datalog engines use heuristics to combine optimization techniques, optimizations are not always performed as observed in [11]. In all cases, they lack the ability to merge fixpoints.

Other approaches to evaluate UCRPQs are based on automata. Each RPQ of an UCRPQ can be translated to an automaton. One problem of this approach is that the choice of the join order is limited. For instance, if we consider the RPQ $a/b+/c/d+$, an automaton is not able to evaluate $b+/c$ first, no matter how it evaluates the query, from left to right or from right to left.

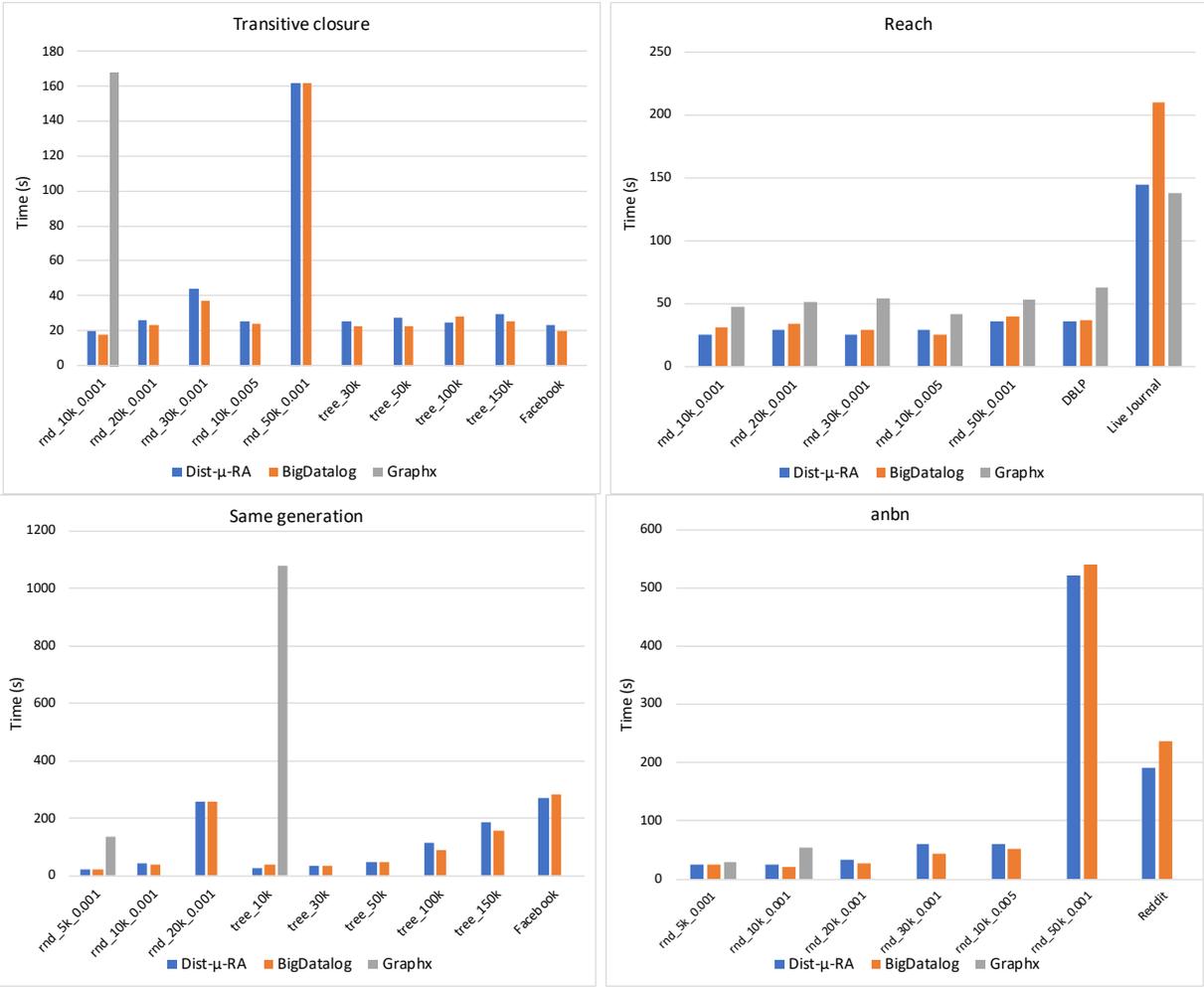


Figure 11. μ -RA queries running times

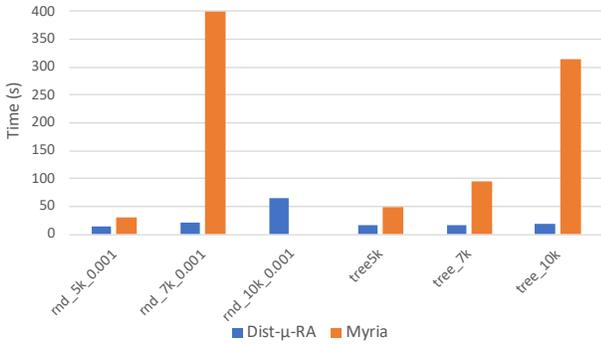


Figure 12. Comparison with Myria on Same Generation.

Waveguide [28] is a system for the evaluation and optimization of RPQs. Since it focuses on the optimization of individual RPQs only, this system misses some global optimizations that can be applied on a conjunction of RPQs. For instance, selections that are in one RPQ cannot be pushed into another. Wireframe [34] and Tasweet [27] are two works that extend/improve on Waveguide but still suffer from this same

limitation.

In summary, Dist- μ -RA relies on μ -RA which enables optimizations at the logical stage (before distribution) that other systems are unable to achieve.

B. Optimizations in distributed data management frameworks

The seminal systems MapReduce and Dryad are known to be inefficient for iterative applications [4]. Systems like Twister [35] and Haloop [36] were proposed to address the shortcomings of MapReduce for iterative computations by reducing the access to disk and reusing in-memory data across iterations. Later, Spark [3] and Flink [4] were introduced to improve upon these systems and became prevalent for large scale and data-parallel computations. Systems specifically designed for large-scale graph processing include Google's Pregel [5], Giraph [6] an open-source system based on the Pregel model, GraphLab [37] and Powergraph [38]. GraphX [7] is a Spark library for graph processing that offers a Pregel API to perform recursive computations. Pregel is based on the Bulk Synchronous Parallel model. A Pregel program is composed

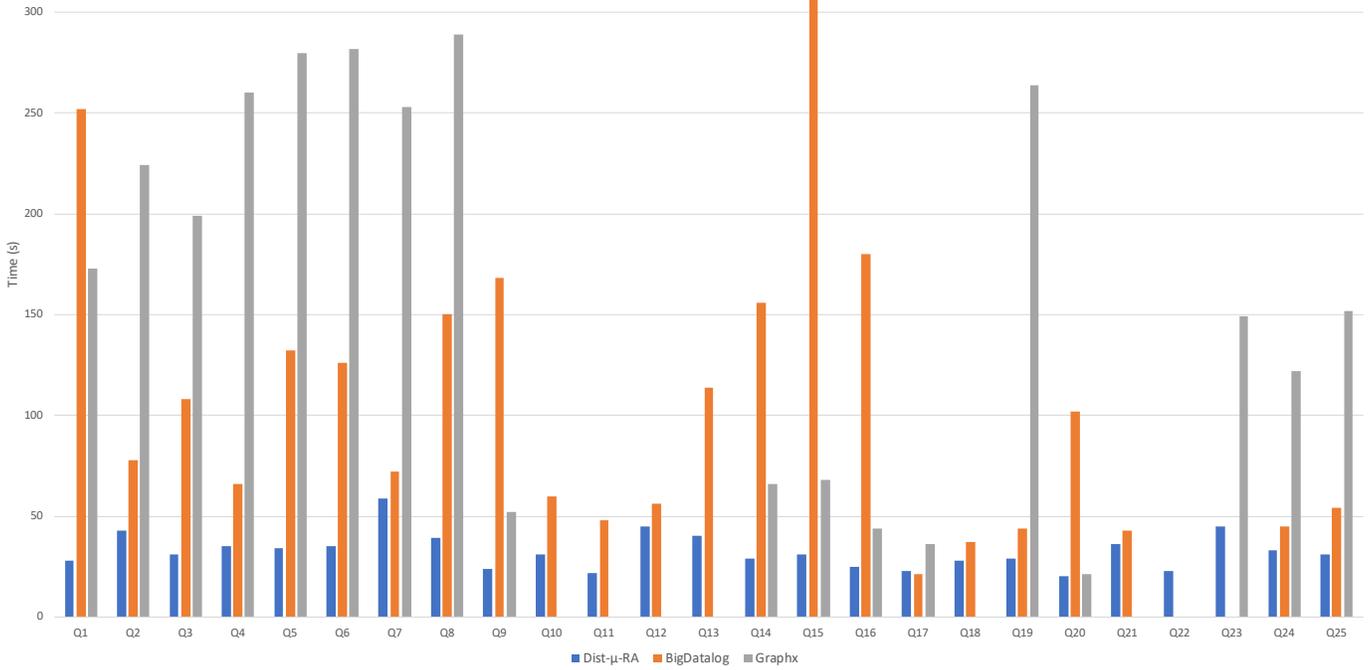


Figure 13. Running times on uniprot_1M.

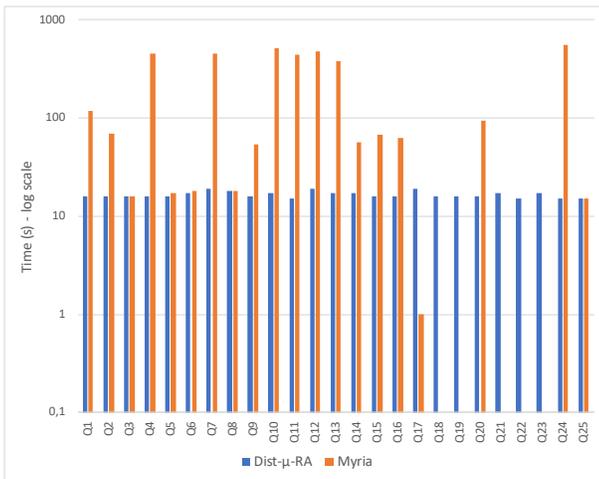


Figure 14. Myria and Dist-μ-RA running times on uniprot_100k.

of supersteps. At each superstep, a vertex receives messages sent by other vertices at the previous iteration and processes them to update its state and send new messages. Computation stops when no new message is sent. It is not straightforward to evaluate UCRPQs in Pregel. An automata like algorithm needs to be written to know which stage of the regular query each processed path has reached. The idea is to traverse the paths in the graph (by sending messages from vertices to their neighbors) while traversing the regular query. [39] proposes a system that implements RPQ queries on GraphX and proposes optimizations to reduce communications between nodes. In all

these systems, selections can be pushed in one direction only. For instance, if the program traverses the regular query from left to right, the execution of the program naturally computes the filters and edge selections occurring before a recursion first, thereby pushing these operations in the recursion. Selections which occur after the recursion cannot be pushed. Additionally, communications between workers of the cluster happen in every superstep of the recursion, which is avoided by the \mathcal{P}_{plw} plan in Dist-μ-RA.

Although systems like Spark and Flink are relatively high level compared to earlier systems, they still require a significant level of programmer expertise to: repartition data, cache data, reorder operators, tune the configuration of the system, etc. in order to achieve better performance. Moreover, these systems are generic and allow to write a wide range of data analytics applications. However, they do not allow for automatic program optimization (like pushing filters), in particular for recursive queries.

Distributed systems with higher-level query language support have been developed. The Spark SQL [40] library enables the user to write SQL queries and process relational data using *Datasets* or *DataFrames*. However, recursion is not supported. DryadLINQ [41] that exposes a declarative query language on top of Dryad (or Pig Latin [42] on top of Hadoop MapReduce) has the same limitation.

SocialLite [43] is an extension of Datalog for social network analysis with graphs. They implement a distributed system that runs queries on a cluster of multi-core machines in which

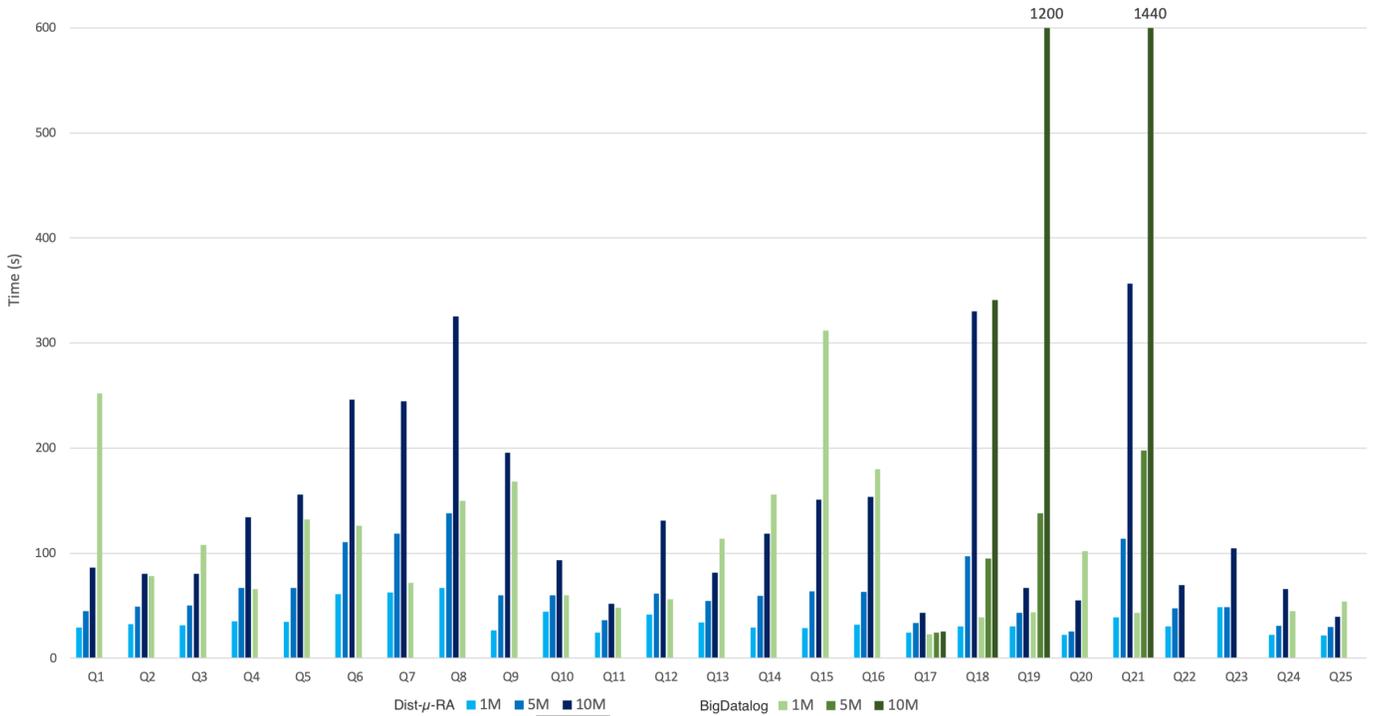


Figure 15. Dist- μ -RA running times on Uniprot of different sizes

workers communicate using message passing. SocialLite does not offer a distribution plan equivalent to \mathcal{P}_{p1w} where recursion can be executed without communication between workers at every step.

Myria [26] is a distributed system that supports a subset of Datalog extended with aggregation. Queries are translated into query plans that are executed on a parallel relational engine. Myria supports incremental evaluation of recursion and provides both a synchronous and an asynchronous mode for evaluating recursive queries. It does not support logical optimizations of the generated query plan involving the recursive operator like pushing joins in fixpoints and merging fixpoints. Myria does not either propose a distribution plan equivalent to \mathcal{P}_{p1w} .

BigDatalog [20] is a recursive Datalog engine that runs on Spark. It proposes *SetRDD*, a specialization of the Spark RDD where each partition is a set and where union and set-difference are performed partition-wise (hence not requiring data transfer between partitions). The present work has been inspired from BigDatalog: it reuses *SetRDDs* to store iteration intermediate results. BigDatalog also proposes a decomposable plan similar to \mathcal{P}_{p1w} . They use the Datalog GPS technique to identify decomposable Datalog programs and determine how to repartition data. The present work brings this technique to the μ -RA framework and use the μ -RA filter pushing technique to automatically repartition data.

RaSQL [44] proposes an extension of SQL with some aggre-

gate operations in recursion. Queries are compiled to Spark SQL to be distributed and executed on Spark. This system does not propose rules to push selections in the fixpoint operator nor to merge fixpoints. RaSQL also proposes a decomposable plan for recursion similar to the one in BigDatalog but has no automated technique to distribute data.

In the experimental section V, the Dist- μ -RA system is experimentally compared to GraphX, BigDatalog (the RaSQL system is not available for testing), and Myria. Experimental results indicate that Dist- μ -RA performs better in most cases and constitutes a good alternative for evaluating UCRPQs.

VII. CONCLUSION

The Dist- μ -RA system combines various techniques for the optimization and distribution of recursive relational queries in a single framework. Regarding the logical/algebraic aspect, it integrates well with the relational algebra. It inherits RA's advantages including the fact that queries are optimized regardless of their initial shape and translation in the algebra. With respect to distribution, different strategies for evaluating recursive algebraic terms in a distributed setting are provided. These strategies are implemented as plans with automated techniques for distributing data in order to reduce communication costs. Experimental results on both real and synthetic graphs show the effectiveness of the proposed approach compared to existing systems.

REFERENCES

- [1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *6th Symposium on Operating System Design and Implementation (OSDI 2004)*, San Francisco, California, USA, December 6-8, 2004, pp. 137–150. [Online]. Available: <http://www.usenix.org/events/osdi04/tech/dean.html>
- [2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 59–72. [Online]. Available: <https://doi.org/10.1145/1272996.1273005>
- [3] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: a unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016. [Online]. Available: <http://doi.acm.org/10.1145/2934664>
- [4] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink™: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015. [Online]. Available: <http://sites.computer.org/debull/A15dec/p28.pdf>
- [5] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 135–146. [Online]. Available: <https://doi.org/10.1145/1807167.1807184>
- [6] "Apache giraph." november 2019. [Online]. Available: <https://giraph.apache.org>
- [7] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, 2014, pp. 599–613. [Online]. Available: <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez>
- [8] S. Abiteboul, R. Hull, and V. Vianu, Eds., *Foundations of Databases: The Logical Level*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.
- [9] R. Agrawal, "Alpha: an extension of relational algebra to express a class of recursive queries," *IEEE Transactions on Software Engineering*, vol. 14, no. 7, pp. 879–885, Jul. 1988.
- [10] 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*, ser. POPL '79. New York, NY, USA: ACM, 1979, pp. 110–119. [Online]. Available: <http://doi.acm.org/10.1145/567752.567763>
- [11] 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*, 2020 (to appear), <https://hal.inria.fr/hal-01673025v5/document>. [Online]. Available: <https://hal.inria.fr/hal-01673025v5/document>
- [12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 15–28. [Online]. Available: <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia>
- [13] M. P. Consens and A. O. Mendelzon, "Graphlog: A visual formalism for real life recursion," in *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '90. New York, NY, USA: ACM, 1990, pp. 404–416. [Online]. Available: <http://doi.acm.org/10.1145/298514.298591>
- [14] P. Barcelo, D. Figueira, and L. Libkin, "Graph logics with rational relations and the generalized intersection problem," in *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, ser. LICS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 115–124. [Online]. Available: <https://doi.org/10.1109/LICS.2012.23>
- [15] P. Barceló, L. Libkin, A. W. Lin, and P. T. Wood, "Expressive languages for path queries over graph-structured data," *ACM Trans. Database Syst.*, vol. 37, no. 4, pp. 31:1–31:46, Dec. 2012. [Online]. Available: <http://doi.acm.org/10.1145/2389241.2389250>
- [16] L. Libkin, W. Martens, and D. Vrgoč, "Querying graphs with data," *J. ACM*, vol. 63, no. 2, pp. 14:1–14:53, Mar. 2016. [Online]. Available: <http://doi.acm.org/10.1145/2850413>
- [17] Y. E. Ioannidis, "On the computation of the transitive closure of relational operators," in *Proceedings of the 12th International Conference on Very Large Data Bases*, ser. VLDB '86. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, p. 403–411.
- [18] M. Lawal, P. Genevès, and N. Layaïda, "A cost estimation technique for recursive relational algebra," in *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*, M. d'Aquino, S. Dietze, C. Hauff, E. Curry, and P. Cudré-Mauroux, Eds. ACM, 2020, pp. 3297–3300. [Online]. Available: <https://doi.org/10.1145/3340531.3417460>
- [19] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, 2015, pp. 1383–1394. [Online]. Available: <https://doi.org/10.1145/2723372.2742797>
- [20] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo, "Big data analytics with datalog queries on spark," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Kourtrika, and S. Madden, Eds. ACM, 2016, pp. 1135–1149. [Online]. Available: <https://doi.org/10.1145/2882903.2915229>
- [21] M. P. I. for Informatics and T. P. University, "YAGO: A high-quality knowledge base," july 2019. [Online]. Available: <https://www.mpi-inf.mpg.de/yago-naga/yago/>
- [22] J. Leskovec, "Snap: Stanford large network dataset collection," november 2019. [Online]. Available: <https://snap.stanford.edu/data/>
- [23] R. Cyganiak, D. Wood, and M. Lanthaler, "Rdf 1.1 concepts and abstract syntax." february 2014. [Online]. Available: <https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225>
- [24] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat, "gmark: Schema-driven generation of graphs and queries," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 4, pp. 856–869, 2017. [Online]. Available: <https://doi.org/10.1109/TKDE.2016.2633993>
- [25] P. Gane, A. Bateman, M. Mj, C. O'Donovan, M. Magrane, R. Apweiler, E. Alpi, R. Antunes, J. Arganiska, B. Bely, M. Bingley, C. Bonilla, R. Britto, B. Bursteinas, G. Chavali, E. Cibrián-Uhalte, S. Ad, M. De Giorgi, T. Dogan, and J. Zhang, "Uniprot: A hub for protein information," *Nucleic Acids Research*, vol. 43, p. D204–D212, 11 2014.
- [26] J. Wang, M. Balazinska, and D. Halperin, "Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines," *Proc. VLDB Endow.*, vol. 8, no. 12, p. 1542–1553, Aug. 2015. [Online]. Available: <https://doi.org/10.14778/2824032.2824052>
- [27] Z. Abul-Basher, N. Yakovets, P. Godfrey, S. Ghajar-Khosravi, and M. H. Chignell, "TASWEET: Optimizing Disjunctive Path Queries in Graph Databases," in *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*. OpenProceedings.org, 2017, pp. 470–473.
- [28] N. Yakovets, P. Godfrey, and J. Gryz, "Waveguide: Evaluating sparql property path queries," in *EDBT*, 2015, pp. 525–528.

- [29] A. Gubichev, S. J. Bedathur, and S. Seufert, "Sparqling kleene: Fast property paths in RDF-3X," in *First International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES '13. New York, NY, USA: ACM, 2013, pp. 14:1–14:7. [Online]. Available: <http://doi.acm.org/10.1145/2484425.2484443>
- [30] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman, "Magic sets and other strange ways to implement logic programs (extended abstract)," in *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, ser. PODS '86. New York, NY, USA: ACM, 1986, pp. 1–15. [Online]. Available: <http://doi.acm.org/10.1145/6012.15399>
- [31] D. Saccà and C. Zaniolo, "On the implementation of a simple class of logic queries for databases," in *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, ser. PODS '86. New York, NY, USA: ACM, 1986, pp. 16–23. [Online]. Available: <http://doi.acm.org/10.1145/6012.6013>
- [32] K. T. Tekle and Y. A. Liu, "More efficient datalog queries: subsumptive tabling beats magic sets," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 2011, pp. 661–672.
- [33] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman, "Efficient evaluation of right-, left-, and multi-linear rules," in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '89. New York, NY, USA: ACM, 1989, pp. 235–242. [Online]. Available: <http://doi.acm.org/10.1145/67544.66948>
- [34] P. Godfrey, N. Yakovets, Z. Abul-Basher, and M. H. Chignell, "WIREFRAME: two-phase, cost-based optimization for conjunctive regular path queries," in *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017*, ser. CEUR Workshop Proceedings, J. L. Reutter and D. Srivastava, Eds., vol. 1912. CEUR-WS.org, 2017. [Online]. Available: <http://ceur-ws.org/Vol-1912/paper22.pdf>
- [35] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 810–818. [Online]. Available: <https://doi.org/10.1145/1851476.1851593>
- [36] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, no. 1–2, p. 285–296, Sep. 2010. [Online]. Available: <https://doi.org/10.14778/1920841.1920881>
- [37] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, p. 716–727, Apr. 2012. [Online]. Available: <https://doi.org/10.14778/2212351.2212354>
- [38] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. USA: USENIX Association, 2012, p. 17–30.
- [39] X. Wang, S. Wang, Y. Xin, Y. Yang, J. Li, and X. Wang, "Distributed pregel-based provenance-aware regular path query processing on rdf knowledge graphs," *World Wide Web*, vol. 23, 05 2020.
- [40] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1383–1394. [Online]. Available: <https://doi.org/10.1145/2723372.2742797>
- [41] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 1–14.
- [42] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 1099–1110. [Online]. Available: <https://doi.org/10.1145/1376616.1376726>
- [43] J. Seo, J. Park, J. Shin, and M. S. Lam, "Distributed socialite: A datalog-based language for large-scale graph analysis," *Proc. VLDB Endow.*, vol. 6, no. 14, p. 1906–1917, Sep. 2013. [Online]. Available: <https://doi.org/10.14778/2556549.2556572>
- [44] J. Gu, Y. H. Watanabe, W. A. Mazza, A. Shkapsky, M. Yang, L. Ding, and C. Zaniolo, "Rasql: Greater power and performance for big data analytics with recursive-aggregate-sql on spark," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 467–484. [Online]. Available: <https://doi.org/10.1145/3299869.3324959>