Distributed Evaluation of Graph Queries using Recursive Relational Algebra

Sarah Chlyah, Pierre Genevès, Nabil Layaïda

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 for reducing data transfers between nodes of the cluster, as well as other platform-specific performance tuning techniques [8].

To facilitate large-scale graph querying, it is important 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. A possible approach 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. 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.

Several works have addressed the problem of query optimization in the presence of recursion, in particular with extensions of Relational Algebra [9]–[11]; and with Datalog-based approaches [12] such as BigDatalog [13]. Recently, μ-RA [11] proposed logical optimization rules for recursion not supported by earlier approaches. In particular, these rules include the merging and reversal of recursions that cannot be done neither with Magic sets nor with Demand Transformations that constitute the core of optimizations in Datalog-based systems [11]. However, μ-RA is limited to the centralized setting.

In this paper, we present Dist-μ-RA a new method and its implementation for the optimized distributed evaluation of recursive relational algebra terms. Specifically, our contribution is threefold:

1) a new method for the optimization of distributed evaluation of queries written in recursive relational algebra. Since it uses a general recursive relational algebra, it can be of interest for a large number of mainstream RDBMS implementations; and it can also provide the support for distributed evaluation of recursive graph query languages. For example Dist-μ-RA provides a frontend where the programmer can formulate queries known as UCRPQs [14]–[17][1]).
   This method provides a systematic parallelisation technique by means of physical plan generation and selection. These plans automatically repartition data in order to reduce data transfer between cluster nodes and communication costs during recursive computations.

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

[1]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. A translation of UCRPQs into the recursive relational algebra is given in [11].

2) a classification of graph queries by the means of six query classes. Each class characterizes queries with a particular feature: for example a recursion with a filter, or concatenated recursions. A fundamental idea of this classification is that a query may belong to one or more classes. Whenever a query belongs to several classes this means that it requires the optimization techniques of all the corresponding classes, together with a technique capable of combining them. Therefore, the more classes a query belongs two, the harder is its optimization.
3) a prototype implementation [18] of the system on top of Apache Spark. Specifically, Dist-μ-RA can use plain Apache Spark or Apache Spark with PostgreSQL as a DBMS backend. Dist-μ-RA is experimentally evaluated using queries covering the different classes, and using datasets (both real and synthetic) of various sizes. Experimental results show that Dist-μ-RA is more efficient than state-of-the-art systems such as BigDatalog [13] in all query classes except one where their performance is similar. This indicates that Dist-μ-RA represents a better alternative for the distributed evaluation of recursive graph queries.

The outline of the paper is as follows: we first present a summary of useful preliminary notions 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

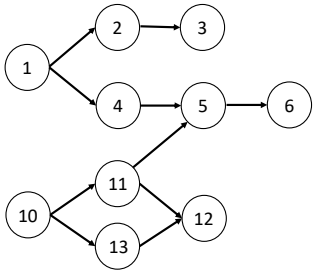
The μ-RA algebra [11] is an extension of the Codd's relational algebra with a recursive operator whose aim is to support recursive terms and transform them when seeking efficient evaluation plans. The syntax of μ-RA is recalled from [11] in Fig. 1. It is composed of database relation variables and operations (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.

φ	::=		term
		X	relation variable
	$\mid$	$\lvert c \to v \rvert$	constant
	$\mid$	$\varphi_1 \cup \varphi_2$	union
	$\mid$	$\varphi_1 \bowtie \varphi_2$	natural join
	$\mid$	$\varphi_1 \triangleright \varphi_2$	antijoin
	$\mid$	$\sigma_{\mathfrak{f}}(\varphi)$	filtering
	$\mid$	$\rho_a^b(\varphi)$	renaming
	$\mid$	$\widetilde{\pi}_a(\varphi)$	anti-projection (column dropping)
	$\mid$	$\mu(X = \Psi)$	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 { $src \to 1$, $dst \to 2$ } 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:



src	dst
1	2
1	4
2	3
4	5
5	6
10	11
10	13
11	5
11	12
13	12

E

src	dst
1	2
1	4
10	11
10	13

S

Figure 2. Graph example.

Example 1. *The term $\widetilde{\pi}_c(\rho_{dst}^c(S) \bowtie \rho_{src}^c(E))$ 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 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 ($\widetilde{\pi}_c$) so as to keep only the two columns `src`, `dst` in the result relation.*

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

The subterm $\varphi = \widetilde{\pi}_c(\rho_{dst}^c(X) \bowtie \rho_{src}^c(E))$ 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:

$$X_0 = \emptyset$$
$$X_1 = \Big\{\{src \to 1, dst \to 2\}, \{src \to 1, dst \to 4\}, \{src \to 10, dst \to 11\}, \{src \to 10, dst \to 13\}\Big\}$$
$$X_2 = X_1 \cup \Big\{\{src \to 1, dst \to 3\}, \{src \to 1, dst \to 5\}, \{src \to 10, dst \to 5\}, \{src \to 10, dst \to 12\}\Big\}$$
$$X_3 = X_2 \cup \Big\{\{src \to 1, dst \to 6\}, \{src \to 10, dst \to 6\}\Big\}$$
$$X_4 = X_3 \quad \textit{(fixpoint reached)}$$

At step 1 it is empty, at step 2 it is a relation of two columns `src` and `dst` 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

$$\varphi(S) = \widetilde{\pi}_c(\rho^c_{dst}(S) \bowtie \rho^c_{src}(E)) = \Big\{\{src \rightarrow 1, dst \rightarrow 3\}, \{src \rightarrow 1, dst \rightarrow 5\}, \{src \rightarrow 10, dst \rightarrow 5\}, \{src \rightarrow 10, dst \rightarrow 12\}\Big\}.$$

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 occurence 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 $\mu(X = \Psi) = \Psi^{\infty}(\emptyset)$.

For instance, $\mu(X = R \triangleright X)$ is not positive, $\mu(X = X \bowtie X)$ is not linear, and $\mu(X = \mu(Y = \varphi(X)))$ is mutually recursive. Whereas $\mu(X = R \cup X \bowtie \mu(Y = \varphi(Y)))$ satisfies F_{cond}.

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 Example 2, S is the constant part and $\widetilde{\pi}_c(\rho^c_{dst}(X) \bowtie \rho^c_{src}(E))$ is the variable part. In the rest of the paper, we only consider fixpoint terms satisfiying 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 [12] and transitive closure evaluation [19] 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.

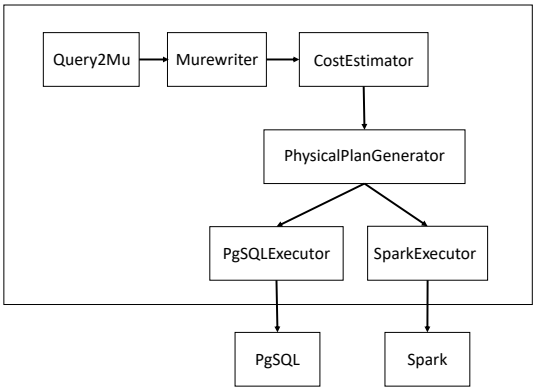


Figure 3. Architecture of the Dist-μ-RA system.

The `Query2Mu` component translates UCRPQs into μ-RA terms. UCRPQ syntax is given in [11] and we give an example below. For sure, Dist-μ-RA also accepts more general μ-RA-terms that are not expressible as UCRPQs[2], 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].

[2]See the practical experiments section for some examples such as the "same generation" query.

The evaluation cost of these terms are estimated by the `CostEstimator` component, based on data cardinality estimations proposed in [20], 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 following UCRPQ composed of a conjunction of two RPQs:

```
?a,?b,?c ←?a wasBornIn/IsLocatedIn+ Japan,
           ?b 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 isLocatedIn+ Japan` is evaluated as a fixpoint starting from `?x` such as `?x isLocatedIn Japan`, which avoids the computation of the whole `isLocatedIn+` relation followed by the filter `Japan`.
- *Pushing joins into fixpoints*: let us consider the query `?x isMarriedTo/knows+ ?y`. Instead of computing the relation `knows+` and joining it with `isMarriedTo`, this rule rewrites the fixpoint such that it starts from `?x` and `?y` that verify `?x isMarriedTo/knows ?y`. The application of this rule is beneficial in this case because the size of the `isMarriedTo/knows` relation is usually smaller than the size of the knows relation.
- *Merging fixpoints*: when evaluating `?x 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 ← ?x 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. `Dataset`s are used to represent relational data in Spark. The optimization of these expressions is delegated to Spark's Catalyst internal optimizer [21] 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. Fig. 4 (left side) illustrates this execution in Spark.The so-called *driver*[3] 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 $\mathcal{P}_{\texttt{gld}}$. On Spark, $\cup$ is executed as `Dataset` union followed by a `distinct` operation. This means that in $\mathcal{P}_{\texttt{gld}}$, 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. This proposition means that a fixpoint whose constant part is a union of two datasets can be obtained by making the union of two fixpoints, each with one of these datasets as a constant part. Thanks to this property 3, the fixpoint can be executed by distributing

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

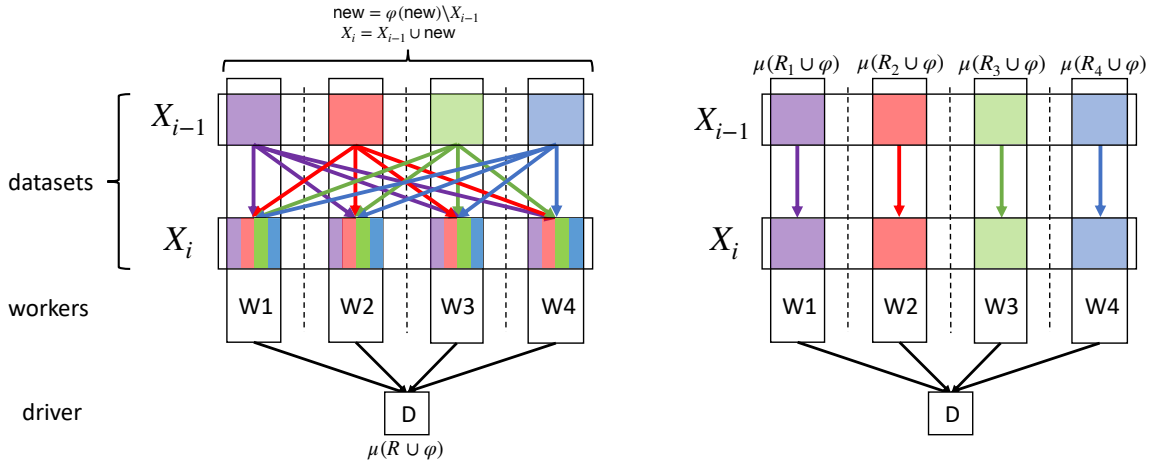


Figure 4. $\mathcal{P}_{\texttt{gld}}$ (left) and $\mathcal{P}_{\texttt{plw}}$ (right) execution on the cluster. Colored arrows show data transfers that occur at each iteration of the fixpoint.

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 $\mathcal{P}_{\texttt{plw}}$. Execution is illustrated in the right side of Fig. 4. As opposed to $\mathcal{P}_{\texttt{gld}}$, $\mathcal{P}_{\texttt{plw}}$ 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 $\mathcal{P}_{\texttt{plw}}$, 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 $\mathcal{P}_{\texttt{plw}}$ can also be avoided by properly repartitioning input data among workers. We first give the inuition, 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 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, ..., 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. □

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.

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

- $\mathcal{P}^{\texttt{pg}}_{\texttt{plw}}$: 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.
- $\mathcal{P}^{\texttt{s}}_{\texttt{plw}}$: 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 `RDD`[4] 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 [13].

$\mathcal{P}_{\texttt{gld}}$ 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`.

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

When $\mathcal{P}^{\texttt{s}}_{\texttt{plw}}$ 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 $\mathcal{P}^{\texttt{pg}}_{\texttt{plw}}$ is chosen, then the operators in the fixpoint expression are executed using the local PostgreSQL instances.

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

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

Dataset	Edges	Nodes
Yago [22]	62,643,951	42,832,856
Facebook [23]	88,234	4,039
Reddit [23]	858,490	55,863
DBLP [23]	1,049,866	317,080
Live Journal [23]	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 use real and synthetic datasets of different sizes and topological properties, as summarized in Table I. We consider the following real graphs:

- Yago[5]: A knowledge graph extracted from Wikipidia and other sources [22].
- The Facebook, Reddit, DBLP and Live Journal real word datasets from the Snap network dataset collection [23].

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.

[5]We use a cleaned version of the real world dataset Yago 2s, that we have preprocessed in order to remove duplicate RDF [24] 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).

- uniprot_n: a benchmark graph of n nodes generated using the gMark benchmark tool [25]. It models the Uniprot database of proteins [26].
- 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 $i^{\text{th}}+1$ 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 [13] available at [27]: 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 programs[6]. 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 ← A pattern ?x)`, only the node `A` sends a message at the start of the computation.
- Myria [28]: 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. For this reason, comparison against Myria is performed on this local configurations on smaller test files and on a subset of test programs.
- 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).

Queries found in practice may contain various forms of recursion. To ensure that tested queries cover all forms of recursion, we rely on a classification of recursive queries in six classes: $\mathcal{C}_1 - \mathcal{C}_6$. Each class regroups queries with a particular recursive feature as follows, where we give for each class an example in UCRPQ:

- $\mathcal{C}_1$: Single recursion, e.g. `?x, ?y ← ?x a+ ?y`
- $\mathcal{C}_2$: Filter to the right of a recursion, e.g. `?x ← ?x a+ C`
- $\mathcal{C}_3$: Filter to the left a recursion, e.g. `?x ← C a+ ?x`
- $\mathcal{C}_4$: Concatenation of a non recursive term to the right of a recursion, e.g. `?x, ?y ← ?x a+/b ?y`
- $\mathcal{C}_5$: Concatenation of a non recursive term to the left of a recursion, e.g. `?x, ?y ← ?x b/a+ ?y`
- $\mathcal{C}_6$: Concatenation of recursions, e.g. `?x, ?y ← ?x a+/b+ ?y`

Each class requires specific optimizations. For instance, the optimization of queries of classes $\mathcal{C}_2$ and $\mathcal{C}_3$ requires pushing a filter in the fixpoint (in two different directions). Queries of classes $\mathcal{C}_4$ and $\mathcal{C}_5$ can be subject of a rewrite by pushing a join in the fixpoint. $\mathcal{C}_2$ and $\mathcal{C}_4$ require reversing the fixpoint before applying rewritings. Queries of $\mathcal{C}_6$ can be rewritten by merging fixpoints or by pushing a join in a fixpoint.

A query may belong to one or more classes. Whenever a query belongs to several classes this means that it requires the optimization techniques of all the corresponding classes, together with a technique capable of combining them. Therefore, the more classes a query belongs two, the harder is its optimization. For example, the query `?x ← C a/b+ ?x` belongs to $\mathcal{C}_3$ because there is a filter to the left of the recursion `b+` and also belongs to $\mathcal{C}_5$ because there is a concatenation to the left of the recursion `b+`.

To ensure that a wide range of recursive queries is covered in the experiments (see Figures 5 and 6), there is, for each class $\mathcal{C}_i$, at least a query that belongs to $\mathcal{C}_i$ alone. The rest of the tested queries belong to different combinations of these classes.

a) Yago queries: Fig. 5 lists the UCRPQs evaluated on the Yago dataset along with their classes. Queries $\mathcal{Q}_1$ to $\mathcal{Q}_7$ are taken from [29], $\mathcal{Q}_8, \mathcal{Q}_9$ from [30], and $\mathcal{Q}_{10}, \mathcal{Q}_{11}$ from [31]. We have added the rest of the queries to illustrate larger transitive closures.

$\mathcal{Q}_{id}$	*Query*	$\mathcal{C}_1$	$\mathcal{C}_2$	$\mathcal{C}_3$	$\mathcal{C}_4$	$\mathcal{C}_5$	$\mathcal{C}_6$
$\mathcal{Q}_1$	`?x ← ?x isMarriedTo/livesIn/IsL+/dw+ Argentina`	×				×	×
$\mathcal{Q}_2$	`?x ← ?x hasChild/livesIn/IsL+/dw+ Japan`	×				×	×
$\mathcal{Q}_3$	`?x ← ?x influences/livesIn/IsL+/dw+ Sweden`	×				×	×
$\mathcal{Q}_4$	`?x ← ?x livesIn/IsL+/dw+ United_States`	×				×	×
$\mathcal{Q}_5$	`?x ← ?x hasSuccessor/livesIn/IsL+/dw+ India`	×				×	×
$\mathcal{Q}_6$	`?x ← ?x hasPredecessor/livesIn/IsL+/dw+ Germany`	×				×	×
$\mathcal{Q}_7$	`?x ← ?x haa/livesIn/IsL+/dw+ Netherlands`	×				×	×
$\mathcal{Q}_8$	`?x ← ?x IsL+/dw+ United_States`	×					×
$\mathcal{Q}_9$	`?x ← ?x (actedIn/-actedIn)+ Kevin_Bacon`	×					
$\mathcal{Q}_{10}$	`?area ← wce -type/(IsL+/dw|dw) ?area`			×	×	×	
$\mathcal{Q}_{11}$	`?person ← ?person isMarriedTo+/owns/IsL+|owns/IsL+ USA`	×		×	×	×	
$\mathcal{Q}_{12}$	`?a,?b ← ?a IsL+/dw ?b`				×		
$\mathcal{Q}_{13}$	`?a,?b ← ?a IsL+/dw+ ?b`						×
$\mathcal{Q}_{14}$	`?a,?b,?c ← ?a wasBornIn/IsL+ ?b, ?b isConnectedTo+ ?c`					×	×
$\mathcal{Q}_{15}$	`?a,?b,?c ← ?a (IsL|isConnectedTo)+ ?b, ?a wasBornIn ?c`					×	
$\mathcal{Q}_{16}$	`?a,?b,?c ← ?a wasBornIn/IsL+ Japan, ?b isConnectedTo+ ?c`	×				×	
$\mathcal{Q}_{17}$	`?a ← ?a IsL+/(isConnectedTo|dw)+ Japan`	×					×
$\mathcal{Q}_{18}$	`?a,?c ← ?a IsL+ Japan, ?a isConnectedTo+ ?c`	×					×
$\mathcal{Q}_{19}$	`?a ← ?a IsL+/IsL Japan`	×			×		
$\mathcal{Q}_{20}$	`?a ← ?a IsL+/isConnectedTo+/dw+ Japan`	×					×
$\mathcal{Q}_{21}$	`?a,?b ← ?a (isL|dw| rdfs:subClassOf|isConnectedTo)+ ?b`	×					
$\mathcal{Q}_{22}$	`?a ← ?a (isConnectedTo/-isConnectedTo)+ SA`	×					
$\mathcal{Q}_{23}$	`?a ← ?a (wasBornIn/isL/-wasBornIn)+ JLT`	×					
$\mathcal{Q}_{24}$	`?a ← Jay_Kappraff (livesIn/isL/-livesIn)+?x`			×			
$\mathcal{Q}_{25}$	`?a,?b ← ?a (actedIn/-actedIn)+/hasChild+ ?b`						×

Figure 5. Queries for the YAGO dataset[7].

[6]In 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.

[7]"`isL`" stands for "`IsLocatedIn`", "`dw`" for "`dealsWith`", "`haa`" for "`hasAcademicAdvisor`", "`SA`" for "`Shannon_Airport`", "`JLT`" for "`John_Lawrence_Toole`", and "`wce`" for "`wikicat_Capitals_in_Europe`".

b) Concatenated closures: are queries of the form a_1+/a_2+/.../a_n+ where $2 \leq n \leq 10$. They all belong to $\mathcal{C}_6$.

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:

$$\mu(X = \widetilde{\pi}_m(\rho^m_{trg}(\sigma_{pred=a}(R)) \bowtie \rho^m_{src}(\sigma_{pred=b}(R))) \cup \widetilde{\pi}_m(\widetilde{\pi}_n(\rho^m_{trg}(\sigma_{pred=a}(R)) \bowtie \rho^n_{trg}(\rho^m_{src}(X)) \bowtie \rho^n_{src}(\sigma_{pred=b}(R)))))$$

- 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:

$$\mu(X = \widetilde{\pi}_m(\rho^m_{src}(R) \bowtie \rho^m_{src}(R)) \cup \widetilde{\pi}_m(\widetilde{\pi}_n(\rho^m_{src}(R) \bowtie \rho^n_{trg}(\rho^m_{src}(X)) \bowtie \rho^n_{src}(R))))$$

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

$$\widetilde{\pi}_{src}(\mu(X = \sigma_{src=N}(R) \cup \widetilde{\pi}_m(\rho^m_{trg}(X) \bowtie \rho^m_{src}(R))))$$

All of these queries belong to $\mathcal{C}_1$.

d) Uniprot queries: In order to test the comparative behavior of Dist-μ-RA on the Uniprot benchmark graphs

Each tested query (fig. 6) has at least one of those shapes. For instance, the query `?x,?y ← ?x interacts/(encodedOn/-encodedOn)+ ?y` has the shape `?x,?y ← ?x a/b+ ?y`.

$\mathcal{Q}_{id}$	Query	$\mathcal{C}_1$	$\mathcal{C}_2$	$\mathcal{C}_3$	$\mathcal{C}_4$	$\mathcal{C}_5$	$\mathcal{C}_6$
$\mathcal{Q}_{26}$	?x,?y ← ?x -hKw/(ref/-ref)+ ?y					×	
$\mathcal{Q}_{27}$	?x,?y ← ?x -hKw/(enc/-enc)+ ?y					×	
$\mathcal{Q}_{28}$	?x,?y ← ?x -hKw/(occ/-occ)+ ?y					×	
$\mathcal{Q}_{29}$	?x,?y ← ?x int/(enc/-enc)+ ?y					×	
$\mathcal{Q}_{30}$	?x,?y ← ?x int/(occ/-occ)+ ?y					×	
$\mathcal{Q}_{31}$	?x,?y ← ?x int+/(occ/-occ)+ ?y						×
$\mathcal{Q}_{32}$	?x,?y ← ?x int+/(enc/-enc)+ ?y						×
$\mathcal{Q}_{33}$	?x,?y ← ?x int+/(occ/-occ)+/(hKw/-hKw)+ ?y						×
$\mathcal{Q}_{34}$	?x,?y ← ?x -hKw/int/ref/(auth/-auth)+ ?y					×	
$\mathcal{Q}_{35}$	?x,?y ← ?x (enc/-enc)+/hKw ?y				×		
$\mathcal{Q}_{36}$	?x ← ?x (enc/-enc)+ C		×				
$\mathcal{Q}_{37}$	?x,?y,?z,?t ← ?x (enc/-enc)+ ?y, ?x int+ ?z, ?x ref ?t					×	×
$\mathcal{Q}_{38}$	?x,?y ← ?x (int|(enc/-enc))+ ?y, C (occ/-occ)+ ?y			×			×
$\mathcal{Q}_{39}$	?x ← ?x int+/ref ?y, C (auth/-auth)+ ?y			×	×		
$\mathcal{Q}_{40}$	?x ← ?x int+/ref ?y, C -pub/(auth/-auth)+ ?y			×	×	×	
$\mathcal{Q}_{41}$	?x ← C -pub/(auth/-auth)+ ?x			×		×	
$\mathcal{Q}_{42}$	?x,?y ← ?x -occ/int+/occ ?y				×	×	
$\mathcal{Q}_{43}$	?x,?y ← ?x (-ref/ref)+ ?y	×					
$\mathcal{Q}_{44}$	?x,?y ← ?x int/ref/(-ref/ref)+ ?y					×	
$\mathcal{Q}_{45}$	?x ← C (ref/-ref)+ ?x			×			
$\mathcal{Q}_{46}$	?x,?y ← ?x (-ref/ref)+/(auth|pub) ?y				×		
$\mathcal{Q}_{47}$	?x ← ?x (enc/-enc | occ/-occ)+ C		×				
$\mathcal{Q}_{48}$	?x ← C int/(enc/-enc|occ/-occ)+ ?x			×		×	
$\mathcal{Q}_{49}$	?x ← C (enc/-enc)+ ?x			×			
$\mathcal{Q}_{50}$	?x ← C (occ/-occ)+ ?x			×			

Figure 6. Uniprot queries[8].

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:

[8]"int" stands for "interacts", "enc" for "encodes", "occ" for "occurs", "hKw" for "hasKeyword", "ref" for "reference", "auth" for "authoredBy", and "pub" for "publishes".

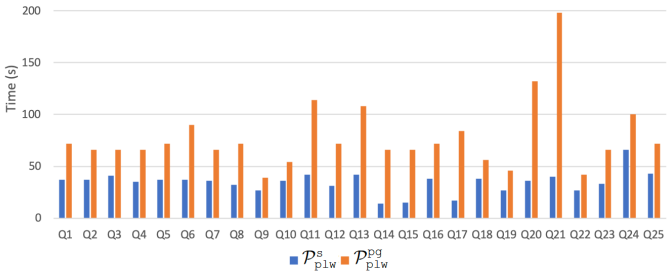


Figure 7. $\mathcal{P}_{plw}$ implementations running times on Yago

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. 7 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 execute faster when distributed. The 17 queries have all in common a large size of intermediate results. Queries handling large amounts of data tend to execute faster using Dist-μ-RA.

In Dist-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 impact of communication cost reduction performed by $\mathcal{P}_{plw}$. As explained in Sec. IV, $\mathcal{P}_{gld}$ requires communications between the workers at each step of the recursion while $\mathcal{P}_{plw}$ does not.

We notice that all the queries where Dist-μ-RA is significantly faster than BigDatalog ($\mathcal{Q}_9$ and $\mathcal{Q}_{22}$ to $\mathcal{Q}_{25}$) have a transitive closure of size exceeding 20M records and belong to classes other than $\mathcal{C}_1$. $\mathcal{Q}_9$ for instance belongs to $\mathcal{C}_2$, so it requires reversing the fixpoint first before pushing the filter "Kevin Bacon". This fixpoint reversal is not supported by the Magic Sets optimization technique of Datalog (see Sec. VI for more details). A similar observation can be made for $\mathcal{Q}_{25}$ of class $\mathcal{C}_6$ and where two fixpoints are merged in Dist-μ-RA. The other queries either have transitive closures of sizes less than 20M or belong to $\mathcal{C}_1$ like $\mathcal{Q}_{21}$. For these queries, Dist-μ-RA and BigDatalog times are very close. This confirms that Dist-μ-RA performs more optimizations and shows that their impact on performance is significant when the size of the data processed by the query is large.

GraphX is much slower overall except for queries like $\mathcal{Q}_{10}$ and $\mathcal{Q}_{24}$ 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

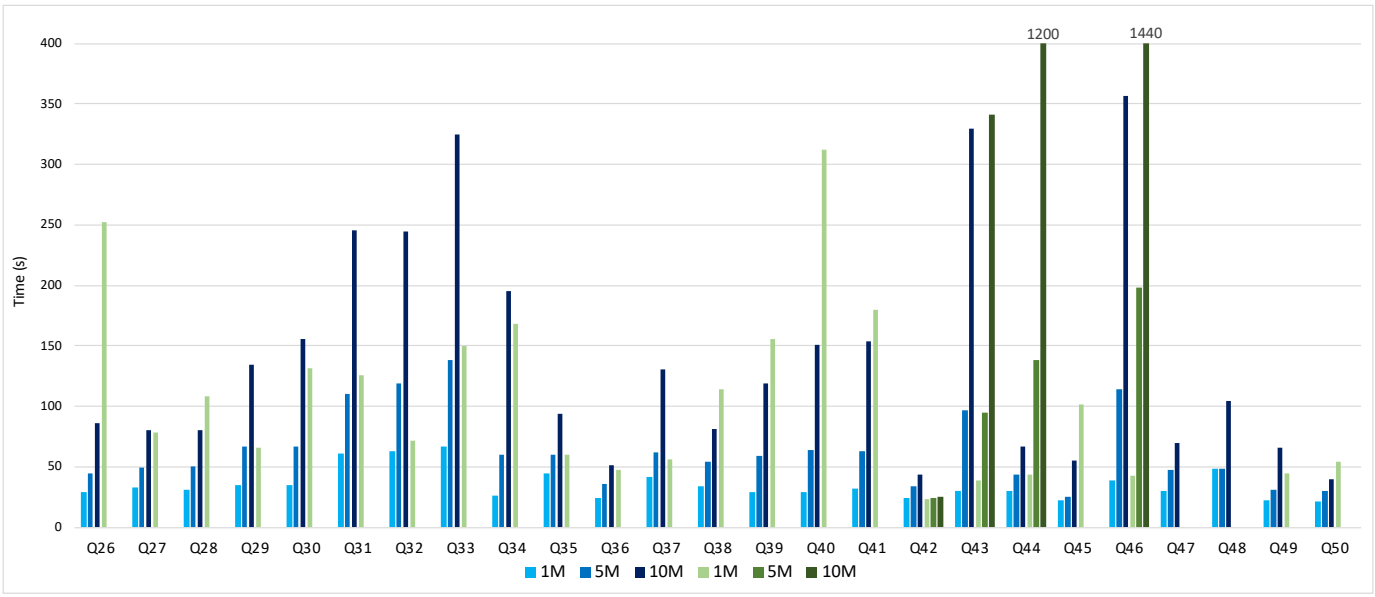


Figure 8. Dist-μ-RA running times on Uniprot of different sizes

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 (which all belong to $\mathcal{C}_6$). The time difference between Dist-μ-RA and other systems for a query with n concatenations (a_1+/.../a_n+) gets larger when n increases. BigDatalog fails for queries where $n \geq 5$ and Centralized μ-RA times out for all queries, while GraphX fails (crashed) 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). The observed results show significant improvements brought by these rewritings on the performance of class $\mathcal{C}_6$ queries.

3) μ-RA queries: Execution times are shown in Fig. 11. Dist-μ-RA and BigDatalog have comparable execution times. This shows that for a variety of queries in $\mathcal{C}_1$, Dist-μ-RA and Bigdatalog have a similar performance. The comparison between Myria and Dist-μ-RA is done on the same generation query. It can be observed in Fig. 12 that the relative perfomance of Dist-μ-RA and Myria increases with the dataset size in favor of Dist-μ-RA. Myria even crashes for rnd_10k_0.001.

4) Evaluation of UCRPQs on Uniprot: the execution times (on the cluster) are shown in Fig 13. Dist-μ-RA answers all of the queries and is faster on all queries belonging to $\mathcal{C}_{2-6}$ (apart from $\mathcal{Q}_{42}$ where the size of the transitive closure is small). This confirms the results observed for the Yago dataset (Sec. V-E1). The comparison with Myria (on a single machine and on a smaller Uniprot file) is shown in Fig 14. We observe that all queries where Myria is slower than Dist-μ-RA (or where it fails) have a transitive closure of size exceeding 500,000 records (and none of the other queries contain a transitive closure greater than this size). Execution times on all other queries are very similar except for $\mathcal{Q}_{42}$. Note that Dist-μ-RA spends approximatey 10s to set the Spark context (and this time is included in the measured Dist-μ-RA execution time).

We perform further scalability tests by measuring Dist-μ-RA and BigDatalog execution times for each Uniprot query on the generated uniprot_n graphs with varying sizes of 1M, 5M and 10M edges. Fig. 8 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. $\mathcal{Q}_{40}$ is one of the queries with the smallest result size (14K for uniprot_10M) and $\mathcal{Q}_{46}$ is one with the largest (around 1.5B for uniprot_10M, which is 150 times the size of the graph).

F. Summary

In summary, when it comes to queries involving large amounts of data, in particular as intermediate results, Dist-μ-RA is more efficient than centralized μ-RA. In addition, Dist-μ-RA is significantly more efficient when compared to GraphX and Myria. Compared to BigDatalog, experimental results indicate that Dist-μ-RA performs better in most cases. Dist-μ-RA evaluates queries belonging to $\mathcal{C}_{2-6}$ in a significantly faster way (especially when intermediate results sizes are large). Dist-μ-RA and BigDatalog have comparable performance for queries in $\mathcal{C}_1$. Overall, experimental results indicate Dist-μ-RA as the best system for evaluating the diversity of recursive queries (the various query classes) on large graphs.

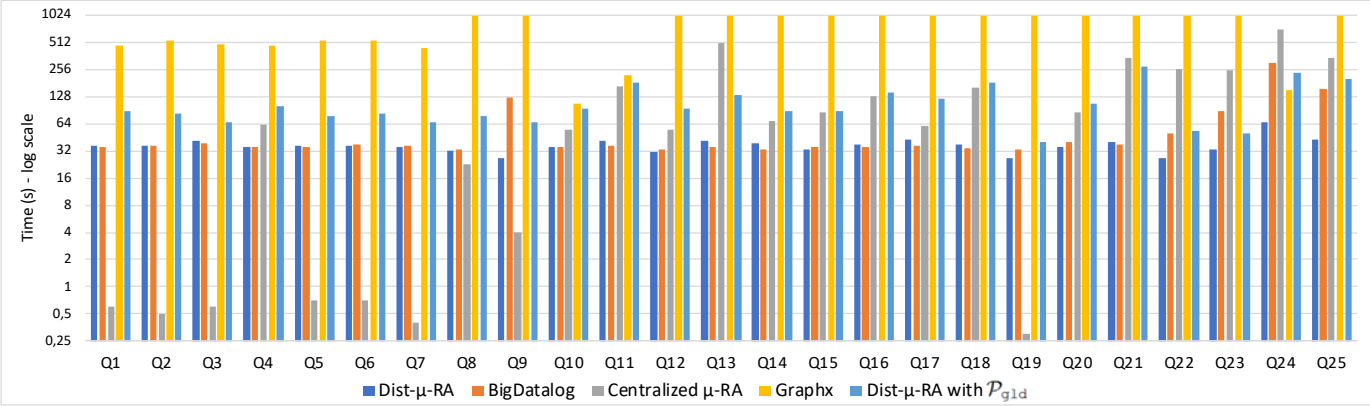


Figure 9. Running times on Yago. A timeout is set at 1,000 s.

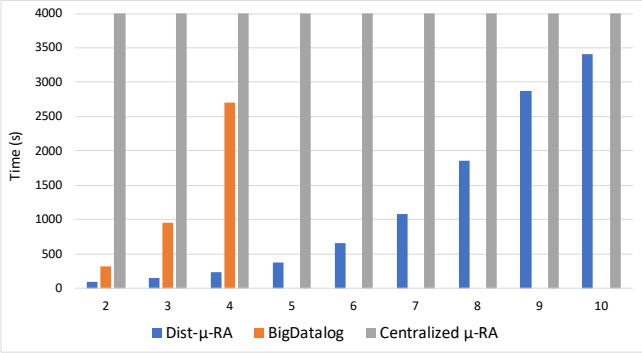


Figure 10. Evaluation times for concatenated closure queries.

VI. Related Works

In order to evaluate expressive queries (such as UCRPQs) over graphs, 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.

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, they are either less expressive than μ-RA or are more expressive but support less optimizations than μ-RA. See [11] and references thereof for more details. α-extended RA [9] introduces an operation α that expresses transitive closure, which means it has the same expressivity as UCRPQs. LFP-RA [10] introduces a least fixpoint operation and has the same expressivity as Datalog with stratified negation. Various fragments of LFP-RA have also been studied, notably the fragment that is restricted to linear recursion and is as expressive as linear Datalog. Finally, the WHILE [12] language is at least as expressive as LFP-RA. It is shown in [11] that μ-RA with the restrictions F_{cond} (Sec. II-B) is as expressive as linear Datalog. So μ-RA is more expressive than α-extended RA, as expressive as the linear version of LFP-RA, and less expressive than the other formalisms.

In Datalog, Magic Sets [32], [33], recently improved by Demand Transformation [34], 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 [35]. 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. 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 MapMeduce and Dryad are known to be inefficient for iterative applications [4].

Spark [3] and Flink [4] were introduced to improve upon these systems and became prevalent for large scale and data-parallel computations. Work in [36] proposes a technique that improves Spark task scheduling and thus performance

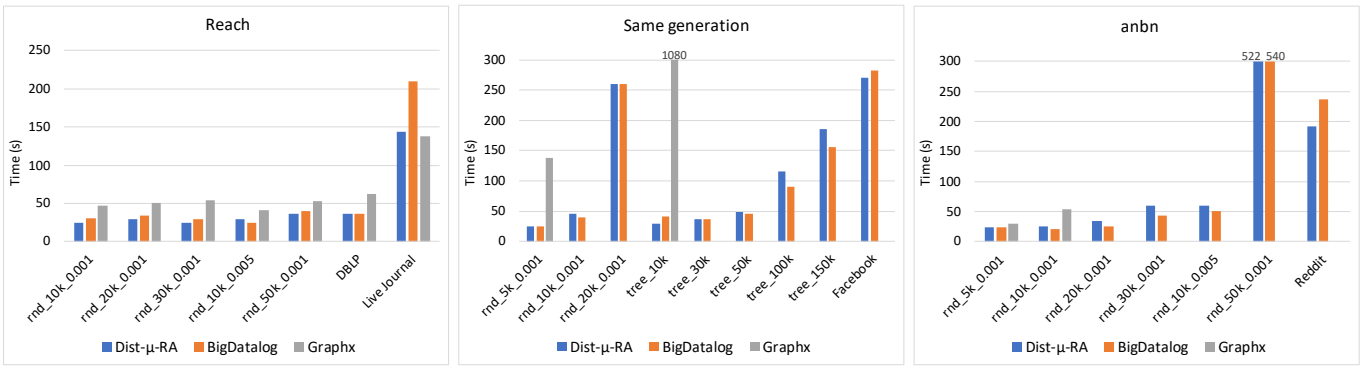


Figure 11. μ-RA queries running times

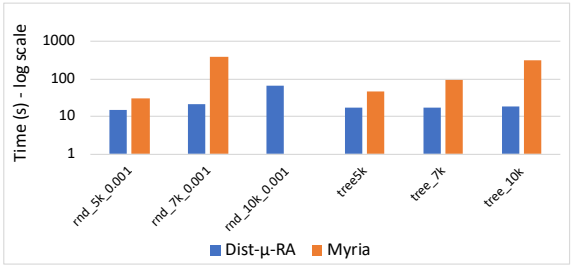


Figure 12. Comparison with Myria on Same Generation.

for iterative applications. This system-level optimization is transparent for Spark applications and thus Dist-μ-RA can directly benefit from it.

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 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.

Is 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 occuring 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 $\mathcal{P}_{\texttt{plw}}$ plan in Dist-μ-RA.

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. TitanDB [43] is a distributed graph database that supports the Gremlin query language. Gremlin provides primitives for expressing graph traversals. It is able to express UCRPQ queries with its own syntax. However, no optimization technique comparable to the ones in Dist-μ-RA have been proposed in the literature.

SociaLite [44] is an extension of Datalog for social network graph analysis. Its distributed implementation runs queries on a cluster of multi-core machines in which workers communicate using message passing. SociaLite does not offer a distribution plan equivalent to $\mathcal{P}_{\texttt{plw}}$ where recursion can be executed without communication between workers at every step.

Myria [28] 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. 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 $\mathcal{P}_{\texttt{plw}}$.

RaSQL [45] proposes an extension of SQL with some aggregate 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. The RASQL implementation is not available for testing.

BigDatalog [13] is a recursive Datalog engine that runs on

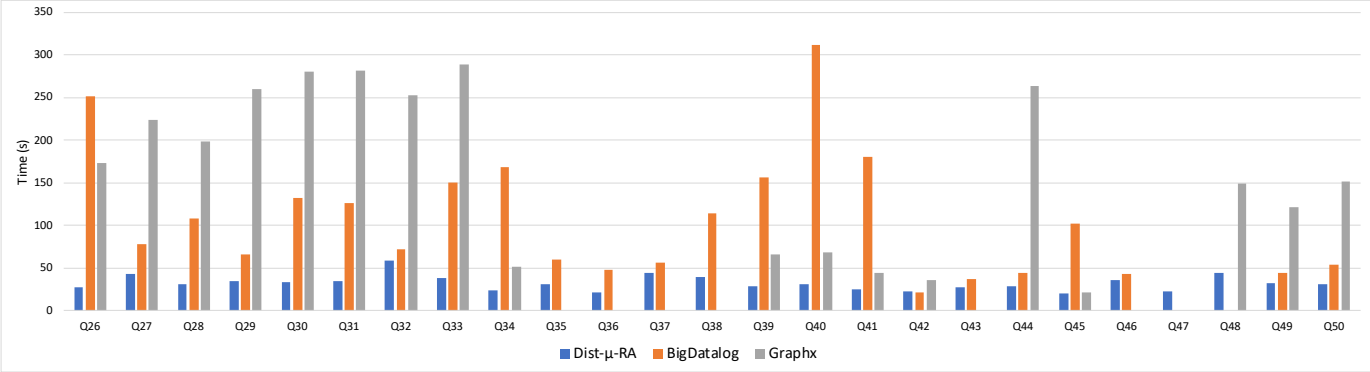


Figure 13. Running times on uniprot_1M.

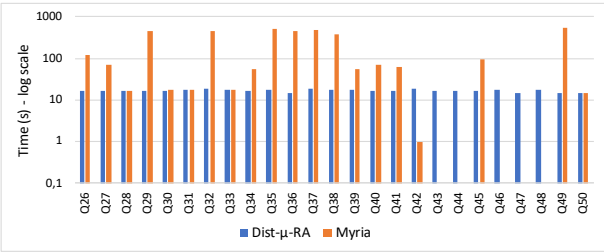


Figure 14. Myria and Dist-μ-RA running times on uniprot_100k.

Spark. It uses the Datalog GPS technique [46] that analyses Datalog rules to identify decomposable Datalog programs and determine how to distribute data and computations. These ideas are tied to Datalog and are not applicable to the relational algebra. The present work proposes a new method specifically designed for recursive relational algebraic terms. It uses the μ-RA filter pushing technique to automatically repartition data. Compared to BigDatalog, Dist-μ-RA is superior because it supports optimizations that BigDatalog is unable to provide. For instance, as mentioned earlier, BigDatalog cannot perform Dist-μ-RA's optimization that merges fixpoints. In fact, it is not possible to do so in the Datalog framework. This is known as an intrinsic limitation of Magic Sets and Demand Transformations at the core of Datalog optimizations. Another example is that BigDatalog cannot push all filters and joins that can be pushed, without requiring the support of techniques to reverse fixpoints. This is notably hard and BigDatalog does not implement such techniques. In comparison, Dist-μ-RA supports more logical optimizations of terms (and regardless of their initial form) and is thus capable of generating logical query plans that are beyond reach for BigDatalog. In practice, Dist-μ-RA provides superior performance on a wider range of query classes, as reported by the experiments in Section V. The only technique that Dist-μ-RA reuses from BigDatalog is a Spark optimized implementation of RDDs: SetRDDs which is a specialization of the Spark `RDD` for sets.

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*, 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] "Spark tuning." [Online]. Available: https://spark.apache.org/docs/latest/tuning.html

[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] 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.

[13] 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. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 1135–1149. [Online]. Available: https://doi.org/10.1145/2882903.2915229

[14] 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

[15] 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

[16] 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

[17] 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

[18] "Dist-μ-RA implementation." [Online]. Available: https://tyrex.inria.fr/distmura

[19] 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.

[20] 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'Aquin, 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

[21] 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

[22] 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/

[23] J. Leskovec, "Snap: Stanford large network dataset collection," november 2019. [Online]. Available: https://snap.stanford.edu/data/

[24] 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

[25] 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

[26] 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.

[27] "Bigdatalog repository." [Online]. Available: https://github.com/ashkapsky/BigDatalog

[28] 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

[29] 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.

[30] N. Yakovets, P. Godfrey, and J. Gryz, "Waveguide: Evaluating sparql property path queries." in *EDBT*, 2015, pp. 525–528.

[31] 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

[32] 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

[33] 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

[34] 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.

[35] 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

[36] P. Katsogridakis, S. Papagiannaki, and P. Pratikakis, "Execution of recursive queries in apache spark," in *Euro-Par*, 2017.

[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] "Titan distributed graph database." [Online]. Available: http://titan.thinkaurelius.com/

[44] 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

[45] 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

[46] J. Seib and G. Lausen, "Parallelizing datalog programs by generalized pivoting," in *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '91. New York, NY, USA: Association for Computing Machinery, 1991, p. 241–251. [Online]. Available: https://doi.org/10.1145/113413.113435