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
Email: *sarah.chlyah@inria.fr, †pierre.geneves@inria.fr, ‡nabil.layaida@inria.fr,

***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*—component, formatting, style, styling, insert**

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]. The work in [36] introduces a cost model for [33] that allows for estimating the best logical plan among a set of equivalent μ-RA plans. However, both these works are 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 twofold:

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

[1]UCRPQs, discussed in more details in Sec. II, 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].

order to reduce data transfer between cluster nodes and communication costs during recursive computations.

2) 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. To evaluate Dist-μ-RA experimentally, a classification of graph queries by the means of seven query classes has been defined. Each class characterizes queries with a particular feature: for example a recursion with a filter, or concatenated recursions. Dist-μ-RA is evaluated using queries that cover 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 most query classes.

The outline of the paper is as follows: we first present preliminary notions in Section I. Section II describes the architecture of Dist-μ-RA. In Section III, we show how μ-RA terms are distributed and how physical plans are generated. Finally, we report on experimental evaluation in Section IV and related works in Section V before concluding.

I. Preliminaries

A. μ-RA syntax

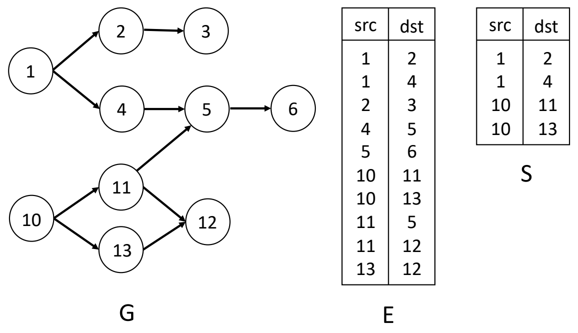
The μ-RA algebra [11] is an extension of the Codd's relational algebra with a recursive operator whose aim is to support recursive terms and transform them when seeking efficient evaluation plans. The syntax of μ-RA is recalled from [11] in Fig. 1. It is composed of database relation variables and operations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



E:

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

S:

src	dst
1	2
1	4
10	11
10	13

Figure 2. Graph example.

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 $\widetilde{\pi}_c(\rho_{dst}^c(S) \bowtie \rho_{src}^c(E))$ 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 a 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.

The evaluation of recursive terms has been studied notably in the context of Datalog [12] and with transitive closure evaluation [19] with the semi-naive (or differential) method. In this approach, a fixpoint term is typically evaluated with the algorithm 1.

Algorithm 1

```
X = R
new = R
while new $\neq \emptyset$:
    new = $\varphi$(new)\ X
    X = X $\cup$ new
return X
```

The final result is obtained by evaluating φ repeatedly starting from $X = R$ until no more results can be produced (the fixpoint has been reached). In this algorithm, φ is applied 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. Notice that this is possible thanks to the property of φ stated in proposition 1, which implies that $\varphi(X_i) \cup \varphi(X_{i+1}) = \varphi(X_i) \cup \varphi(X_{i+1} \setminus X_i)$.

II. Dist-μ-RA architecture

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. Specifically, the Dist-μ-RA system is composed of several components, as illustrated in Fig. 3.

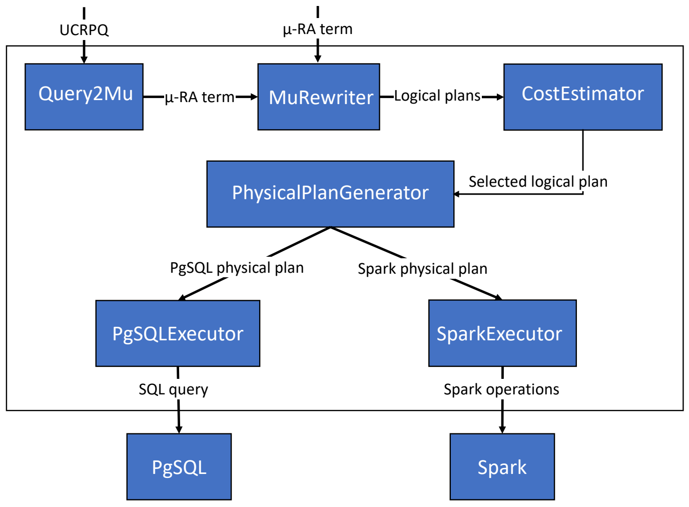


Figure 3. Architecture of the Dist-μ-RA system.

The `Query2Mu` component translates recursive graph queries written in Union of Conjunctive Regular Path Queries (UCRPQ) into μ-RA terms. The UCRPQ syntax is given in [11] and we give an example below. Dist-μ-RA supports 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 I.

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 costs of these terms are estimated by the `CostEstimator` component proposed in [20]. This component is an implementation of a classical Selinger style cost

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

estimator [21] based on cardinality estimation. Based on these estimations, a best logical recursive plan is selected.

From a given recursive logical plan, the `PhysicalPlanGenerator` generates a physical plan for distributed execution (see Section III). Two distributed execution setups are used. In the first setup (using `PgSQLExecutor`), each Spark worker runs a PostgreSQL instance to perform a part of the evaluation locally. The second setup (using `SparkExecutor`) relies only on Spark. In all cases, the query evaluation is performed on top of Spark.

a) Example: We first describe the transformations that a query (UCRPQ or μ-RA term) undergoes before being considered for distributed evaluation when given as input parameter to the `PhysicalPlanGenerator` (described in Section III).

Consider for instance the following UCRPQ composed of a conjunction of two Regular Path Queries (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 plan for distributed execution.

III. Distributed evaluation

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. Fixpoint distributed evaluation principles

The first principle uses a global loop on the Spark *driver*[3]. The second principle uses parallel local loops on the Spark workers, and corresponds to our contribution.

1) Global Loop on the Driver ($\mathcal{P}_{gld}$): $\mathcal{P}_{\texttt{gld}}$ corresponds to the natural way a Spark programmer would implement the fixpoint operation: it distributes the computations performed at each iteration of Algorithm 1. This execution is illustrated in Fig. 4 (left side). Colored arrows show data transfers that occur at each iteration of the fixpoint. The driver performs the loop and, at each iteration, instructions at lines 4 and 5 are executed as `Dataset` [4] 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 ($\mathcal{P}_{plw}$): This evaluation principle uses the following observation to distribute the fixpoint:

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,

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

[4]Distributed collection data structure used to store relational data in Spark [22]

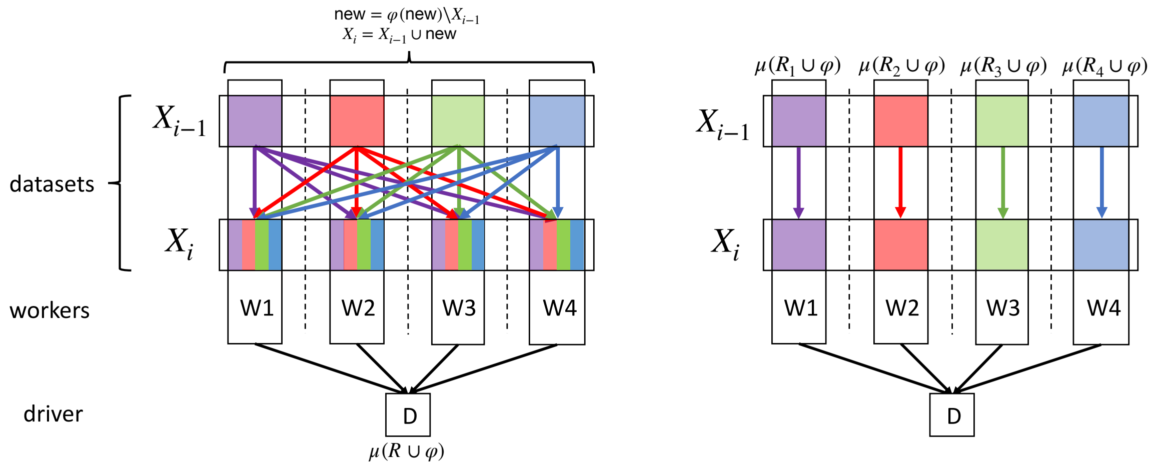


Figure 4. Execution on the cluster of $\mathcal{P}_{\texttt{gld}}$ (left) and $\mathcal{P}_{\texttt{plw}}$ (right).

each with one of these datasets as a constant part. Thanks to this proposition 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 $\mathcal{P}_{\texttt{plw}}$. Execution is illustrated in the right side of Fig. 4. As opposed to $\mathcal{P}_{\texttt{gld}}$, $\mathcal{P}_{\texttt{plw}}$ performs only one data shuffle at the end to make the union ($\cup$) 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 $\mathcal{P}_{\texttt{plw}}$: There are cases where the final data shuffle induced by $\mathcal{P}_{\texttt{plw}}$ can also be avoided by appropriately repartitioning input data among workers. We first give the intuition behind this idea, followed by the proof.

We look for a column *col* (or a set of columns) in X 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 for certain 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 a 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 and selection

We present the different physical plans automatically generated by the Dist-μ-RA system for μ-RA terms, and explain how they are selected. Dist-μ-RA generates a physical plan for $\mathcal{P}_{\texttt{gld}}$, which is used only as a baseline in performance comparisons.

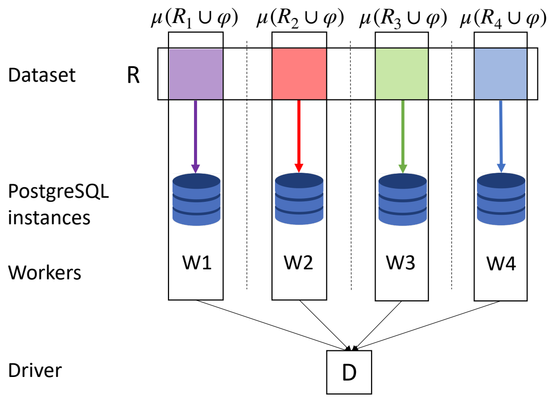


Figure 5. Execution on the cluster of $\mathcal{P}^{\mathtt{pg}}_{\mathtt{plw}}$.

We propose two alternative physical plans which are variants of $\mathcal{P}_{\mathtt{plw}}$:

- $\mathcal{P}^{\mathtt{pg}}_{\mathtt{plw}}$: Fig. 5 illustrates the execution of this physical plan. The local fixpoints are executed on PostgreSQL. The fixpoint operator is performed as a Spark `mapPartition()` operation where each worker performs a portion 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. 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. The local PostgreSQL plans are selected for the operators in the fixpoint expression. Each PostgreSQL executor returns its results as an iterator which is then processed by Spark.

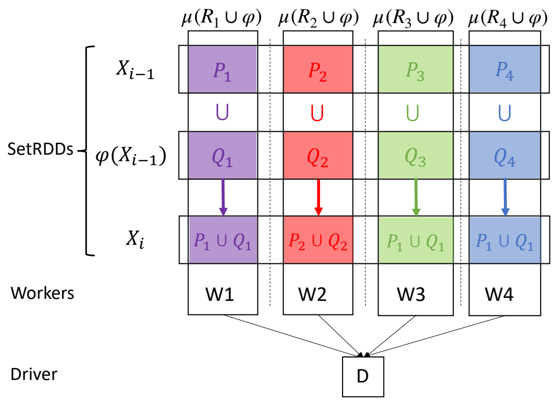


Figure 6. Execution on the cluster of $\mathcal{P}^{\mathtt{s}}_{\mathtt{plw}}$.

- $\mathcal{P}^{\mathtt{s}}_{\mathtt{plw}}$: Fig. 6 illustrates the execution of this physical plan. 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 that each worker performs its own fixpoint independently (i.e. without data exchanged between workers). 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. 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` [13] is a special `RDD`[5] 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.

As a consequence, for each of the non-recursive μ-RA operators, there are two kinds of physical plans: local plans implemented using PostgreSQL and distributed plans implemented using the Spark `Dataset` API. `Datasets` are used to represent relational data in Spark. The optimization of these expressions is then delegated to Spark's Catalyst internal optimizer [23] before execution. Some operators 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.

As mentioned earlier, datasets in the variable part of the fixpoint are broadcasted to the workers in $\mathcal{P}^{\mathtt{s}}_{\mathtt{plw}}$, whereas in $\mathcal{P}^{\mathtt{pg}}_{\mathtt{plw}}$, they are are stored as tables in Postgres that are queried by the parallel tasks. To select between the two alternatives $\mathcal{P}^{\mathtt{s}}_{\mathtt{plw}}$ and $\mathcal{P}^{\mathtt{pg}}_{\mathtt{plw}}$, we rely on the following criteria: when the size of the datasets in the variable part of the fixpoint exceeds the memory available for a task[6], we select $\mathcal{P}^{\mathtt{pg}}_{\mathtt{plw}}$ and $\mathcal{P}^{\mathtt{s}}_{\mathtt{plw}}$ otherwise.

IV. Experiments

We evaluate the performance of a prototype implementation of the Dist-μ-RA system on top of the Spark platform [3]. We extensively compared its 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.

[5]`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.

[6]A Spark task is unit of computation executed on the worker on a single partition. Tasks are executed in parallel on partitioned data.

B. Datasets

Real Dataset	Description	Edges	Nodes
Yago [24]	YAGO semantic knowledge base	62,643,951	42,832,856
Epinions [25]	Epinions product ratings (2005)	13,668,320	996,744
Wikitree [26]	Online genealogy dataset	9,192,212	1,382,751
Coauth-M [25]	MAG Geology coauthor simplices	5,120,762	1,256,385
Gottron [27]	Wikipedia words	2,941,903	273,961
AcTree [28]	Academic family tree data	1,561,494	777,220
Wikitree_0 [26]	Wikitree filtered on relation ID 0	1,556,453	1,019,438
Reddit [29]	Hyperlinks between subreddits	858,490	55,863
TW-Cannes [30]	Cannes Multiplex social network	991,855	438,539
Higgs-RW [25]	Twitter, Higgs boson (2012)	733,647	425,008
Wikidata_c [31]	Wikidata child relation	280,405	333,572
Wikidata_p [31]	Wikidata father & mother relations	280,740	334,430
Facebook [29]	Social circles from Facebook	88,234	4,039
Ragusan [25]	Ragusan nobility genealogy	51,938	13,690
Isle-of-Man [25]	Isle of Man genealogy	36,666	10,474
Fr-Royalty [32]	French royalty genealogy tree	12,358	2,127

Synthetic Dataset	Edges	Nodes
uniprot_10M	10,001,920	10,000,000
uniprot_5M	5,001,427	5,000,000
uniprot_1M	1,000,443	1,000,000
uniprot_100k	66,181	100,000
rnd_100k_0.001	5,003,893	100,000
rnd_10k_0.001	249,791	10,000
rnd_7k_0.001	24,630	7,000
rnd_5k_0.001	12,660	5,000
tree_$10k$	9,999	10,000
tree_$7k$	6,999	7,000
tree_$5k$	4,999	5,000

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[7]: A knowledge graph extracted mainly from Wikipidia [24].
- datasets from the Colorado index of complex networks [25] and from the Snap network dataset collection [29].

In addition, we consider the following synthetic graphs:

- uniprot_n: a benchmark graph of n nodes generated using the gMark benchmark tool [34]. It models the Uniprot database of proteins [35].
- 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.
- 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.

[7]We use a cleaned version of the real world dataset Yago 2s, that we have preprocessed in order to remove duplicate RDF [33] 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).

C. Systems

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

- BigDatalog [13] available at [36]: 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[8]. Specifically, we compute a regular graph query by making each node send a message to its 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 pattern ?x`), only the node `A` sends a message at the start of the computation.

D. Queries

Queries may contain various forms of recursion. To ensure that tested queries cover different forms of recursion, we rely on a classification of queries in seven classes. Each class regroups queries with a particular recursive feature: $\mathcal{C}_1 - \mathcal{C}_6$ describe UCRPQ queries on knowledge graphs (like Uniprot and Yago). We also provide additional experiments using more general queries not expressible as UCRPQs in the class $\mathcal{C}_7$. The classification is the following:

- $\mathcal{C}_1$ corresponds to queries containing a single transitive closure (TC), e.g. `?x, ?y` $\leftarrow$ `?x a+ ?y`
- $\mathcal{C}_2$: queries with a filter to the right of a TC, e.g. `?x` $\leftarrow$ `?x a+ C`
- $\mathcal{C}_3$: queries with a filter to the left a TC, e.g. `?x` $\leftarrow$ `C a+ ?x`
- $\mathcal{C}_4$: queries which contain a concatenation of a non recursive term to the right of a TC, e.g. `?x, ?y` $\leftarrow$ `?x a+/b ?y`
- $\mathcal{C}_5$: queries which contain a concatenation of a non recursive term to the left of a TC, e.g. `?x, ?y` $\leftarrow$ `?x b/a+ ?y`
- $\mathcal{C}_6$: queries which contain a concatenation of TCs, e.g. `?x, ?y` $\leftarrow$ `?x a+/b+ ?y`
- $\mathcal{C}_7$: queries with non regular recursion, e.g. $a^n b^n$.

Each class requires specific optimizations. For instance, the optimization of queries of classes $\mathcal{C}_2$ and $\mathcal{C}_3$ requires pushing filters in fixpoint terms (in two different directions). Queries of classes $\mathcal{C}_4$ and $\mathcal{C}_5$ require an optimization that pushes joins in fixpoint terms. $\mathcal{C}_2$ and $\mathcal{C}_4$ require reversing fixpoint terms before applying other optimizations (rewritings). Queries of $\mathcal{C}_6$ can be optimized by merging fixpoints or by pushing joins in fixpoint terms.

[8]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.

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 to, 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 transitive closure `b+` and also belongs to $\mathcal{C}_5$ because there is a concatenation to the left of `b+`.

To cover a variety of queries in the experiments (see Figures 7 and 8), there is, for each class $\mathcal{C}_i$, at least one query that belongs to $\mathcal{C}_i$ alone. In addition, we also consider queries that belong to $\mathcal{C}_i$ and to a combination of other classes. This allows to test how the different combinations of optimizations are supported by the tested systems.

a) Yago queries: Fig. 7 lists the UCRPQs evaluated on the Yago dataset along with their classes. Queries $\mathcal{Q}_3$ and $\mathcal{Q}_4$ are taken from [37], $\mathcal{Q}_5$ from [38], and $\mathcal{Q}_6, \mathcal{Q}_7$ from [39]. We have added queries $\mathcal{Q}_8 - \mathcal{Q}_{25}$ that include 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,?y`	`← ?x,?y <- ?x hasChild+ ?y`	×					
$\mathcal{Q}_2$	`?x,?y`	`← ?x,?y <- ?x isConnectedTo+ ?y`	×					
$\mathcal{Q}_3$	`?x`	`← ?x isMarriedTo/livesIn/IsL+/dw+ Argentina`		×			×	×
$\mathcal{Q}_4$	`?x`	`← ?x livesIn/IsL+/dw+ United_States`		×			×	×
$\mathcal{Q}_5$	`?x`	`← ?x (actedIn/-actedIn)+ Kevin_Bacon`		×				
$\mathcal{Q}_6$	`?area`	`← wce -type/(IsL+/dw|dw) ?area`			×	×	×	
$\mathcal{Q}_7$	`?person`	`← ?person isMarriedTo+/owns/IsL+|owns/IsL+ USA`		×		×	×	
$\mathcal{Q}_8$	`?x,?y`	`← ?x IsL+/dw+ ?y`						×
$\mathcal{Q}_9$	`?x,?y`	`← ?x (IsL|dw|rdfs:subClassOf|isConnectedTo)+ ?y`	×					
$\mathcal{Q}_{10}$	`?x`	`← ?x (isConnectedTo/-isConnectedTo)+ S_Airport`		×				
$\mathcal{Q}_{11}$	`?person`	`← ?person (wasBornIn/IsL/-wasBornIn)+ JLT`		×				
$\mathcal{Q}_{12}$	`?x`	`← Jay_Kappraff (livesIn/IsL/-livesIn)+ ?x`			×			
$\mathcal{Q}_{13}$	`?x,?y`	`← ?x (actedIn/-actedIn)+/hasChild+ ?y`						×
$\mathcal{Q}_{14}$	`?x,?y`	`← ?x (wasBornIn/IsL/-wasBornIn)+/isMarriedTo ?y`				×		
$\mathcal{Q}_{15}$	`?x,?y`	`← ?x (actedIn/-actedIn)+/influences ?y`				×		
$\mathcal{Q}_{16}$	`?x`	`← Marie_Curie (hWP/-hWP)+ ?x`			×			
$\mathcal{Q}_{17}$	`?x`	`← London -wasBornIn/(playsFor/-playsFor)+ ?x`			×		×	
$\mathcal{Q}_{18}$	`?x`	`← London (-wasBornIn/hWP/-hWP/wasBornIn)+ ?x`			×			
$\mathcal{Q}_{19}$	`?x,?y`	`← ?x -actedIn/(-created/influences/created)+ ?y`					×	
$\mathcal{Q}_{20}$	`?x,?y`	`← ?x -isLeaderOf/(livesIn/-livesIn)+ ?y`					×	
$\mathcal{Q}_{21}$	`?x,?y`	`← ?x (-created/created)+/directed ?y`				×		
$\mathcal{Q}_{22}$	`?x`	`← Lionel_Messi (playsFor/-playsFor)+/isAff ?y`			×	×		
$\mathcal{Q}_{23}$	`?x`	`← SH (haa|influences)+/(isMarriedTo|hasChild)+ ?x`			×			×
$\mathcal{Q}_{24}$	`?x,?y`	`← ?x isConnectedTo+/IsL+/dw+/owns+ ?y`						×
$\mathcal{Q}_{25}$	`?x,?y`	`← ?x haa/hasChild/(hWP/-hWP)+ ?y`					×	

Figure 7. Queries for the YAGO dataset[9].

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

c) Non regular queries: We also consider queries that contain non-regular forms of recursion. These queries are exclusively expressible as μ-RA terms, not as UCRPQs. All of these queries belong to $\mathcal{C}_7$:

- $a^n b^n$ queries: they return 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. They are expressed with the following μ-RA term:

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

- Same Generation (SG) queries: they return the pairs of nodes that are of the same generation in a graph. We use the following term to express them:

$$\begin{aligned}T_{SG} =& \mu(X = \widetilde{\pi}_m(\rho_{src}^{m}(R) \bowtie \rho_{src}^{m}(R))\\ &\cup\, \widetilde{\pi}_m(\widetilde{\pi}_n(\rho_{src}^{m}(R) \bowtie \rho_{trg}^{n}(\rho_{src}^{m}(X)) \bowtie \rho_{src}^{n}(R))))\end{aligned}$$

- Filtered SG queries: they compute the pairs of nodes that are of the same generation for a particular predicate p in a graph.

$$\sigma_{pred=p}(T_{SG})$$

- Joined SG: they return the pairs of nodes that are of the same generation for a particular set of predicates P in a graph. P is a one column ($pred$) relation that gets joined with the T_{SG} term on the column $pred$:

$$P \bowtie T_{SG}$$

d) Uniprot queries: For the synthetic Uniprot datasets, we use the UCRPQ queries shown in Fig. 8.

$\mathcal{Q}_{id}$	*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`	`← C (occ/-occ)+ ?x`			×			
$\mathcal{Q}_{29}$	`?x,?y`	`← ?x int+/(occ/-occ)+/(hKw/-hKw)+ ?y`						×
$\mathcal{Q}_{30}$	`?x`	`← ?x (enc/-enc | occ/-occ)+ C`		×				
$\mathcal{Q}_{31}$	`?x,?y`	`← ?x int+/(occ/-occ)+ ?y`						×
$\mathcal{Q}_{32}$	`?x,?y`	`← ?x int+/(enc/-enc)+ ?y`						×
$\mathcal{Q}_{33}$	`?x,?y`	`← ?x int/(enc/-enc)+ ?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,?y`	`← ?x int/(occ/-occ)+ ?y`					×	
$\mathcal{Q}_{48}$	`?x`	`← C int/(enc/-enc|occ/-occ)+ ?x`			×		×	
$\mathcal{Q}_{49}$	`?x`	`← C (enc/-enc)+ ?x`			×			
$\mathcal{Q}_{50}$	`?x,?y`	`← ?x -hKw/(occ/-occ)+ ?y`					×	

Figure 8. Uniprot queries[10].

E. Results

We report on experimental results and analyse them. We measure the time spent in evaluating queries by the different systems, in seconds. For each set of experiments, we define a timeout value. Whenever the time spent in evaluating a query reaches this timeout value, we consider that the query evaluation did not terminate within reasonable time. On charts, the timeout value corresponds to the maximum value on the y-axis. Some systems crashed in some query evaluations. In charts, this is denoted by the presence of a red cross on a time bar. The observed crashes are out of memory and

[9] "`isL`" stands for "`IsLocatedIn`", "`dw`" for "`dealsWith`", "`haa`" for "`hasAcademicAdvisor`", "`JLT`" for "`John_Lawrence_Toole`", "`hWP`" for "`hasWonPrize`", "`SH`" for "`Stephen_Hawking`", "`isAff`" for "`isAffiliatedTo`", "`S_Airport`" for "`Shannon_Airport`", and "`wce`" for "`wikicat_Capitals_in_Europe`".

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

timeout failures. They are due to the amount of data processed and transferred over the network that exceeds the capacity of the machines. This amount of data is linked to the size of intermediate results produced by the query evaluation. The other cases correspond to query evaluations where the system answered correctly. The plotted times represent the average running times over three executions.

1) Dist-μ-RA recursive plans evaluation: Fig. 9 presents the time spent in evaluating each of the Dist-μ-RA plans (Sec. III) for UCRPQs on the Yago dataset. We observe that the $\mathcal{P}_{\texttt{plw}}$ plans are faster than $\mathcal{P}_{\texttt{gld}}$. This illustrates the interest of the communication cost reduction performed by $\mathcal{P}_{\texttt{plw}}$. As explained in Sec. III, $\mathcal{P}_{\texttt{gld}}$ requires communications between the workers at each step of the recursion while $\mathcal{P}_{\texttt{plw}}$ does not. Furthermore, we observe that $\mathcal{P}^{\texttt{s}}_{\texttt{plw}}$ performs better on most cases when compared to $\mathcal{P}^{\texttt{pg}}_{\texttt{plw}}$. This is due to the cost of data marshalling and exchange between PostgreSQL and Spark. However, when the size of the datasets in the variable part of the fixpoints is large, $\mathcal{P}^{\texttt{pg}}_{\texttt{plw}}$ becomes faster (e.g. queries $\mathcal{Q}_{22}$ and $\mathcal{Q}_{25}$). In that case, the performance gains are due to local optimizations of this variable part performed by PostgreSQL.

2) UCRPQs on Yago: comparison with other systems:

In Fig. 10 we present the performance results of Dist-μ-RA, BigDatalog and GraphX on the queries $\mathcal{Q}_1 - \mathcal{Q}_{25}$ on the Yago dataset.

First, these results show that Dist-μ-RA is much faster than GraphX overall. 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. The only case where GraphX matches the performance of Dist-μ-RA is for $\mathcal{Q}_{17}$ where filtering is performed at the beginning of the query (see Sec. IV-C).

Second, these results show that Dist-μ-RA provides much faster performance than Bigdatalog for all the classes $\mathcal{C}_2$-$\mathcal{C}_6$, and comparable performance for class $\mathcal{C}_1$.

One explanation for the difference in performance of $\mathcal{Q}_5$ of class $\mathcal{C}_2$ is that it requires reversing a fixpoint term first before pushing the filter "Kevin Bacon". This fixpoint reversal is not supported by Datalog's Magic Sets optimization technique (see Sec. V for more details). Another example is $\mathcal{Q}_{24}$ of class $\mathcal{C}_6$ where Dist-μ-RA merges fixpoint terms (which BigDatalog is unable to do). Overall, the optimizations in Dist-μ-RA are more effective. We noticed that this is particularly true when the size of intermediate results is large ($\mathcal{Q}_5$ and $\mathcal{Q}_{10} - \mathcal{Q}_{25}$).

3) Concatenated closures: We now evaluate concatenated closure queries (which belong to $\mathcal{C}_6$) on the graph obtained from rnd_100k_0.001. The graph edges are randomly labeled from a set of 10 different labels. Results are shown in Fig. 11. Dist-μ-RA is faster on all queries. The time difference between Dist-μ-RA and BigDatalog for a query with n concatenations (a_1+/.../a_n+) becomes larger when n increases. BigDatalog fails for queries where $n \geq 5$ and GraphX crashes on all queries. The plans that are 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. II). These results also indicate that optimizations introduced by these rewritings provide significant performance gains for class $\mathcal{C}_6$.

4) Non regular queries: The execution times for these queries of class $\mathcal{C}_7$ are given in Fig. 12. On the basic SG and $a^n b^n$ queries, Dist-μ-RA and BigDatalog have comparable execution times. Dist-μ-RA is faster on Filtered SG and Joined SG queries.

5) UCRPQs on Uniprot: The results reported in Fig 13 show that Dist-μ-RA is the only system that answers all of the queries. Furthermore, Dist-μ-RA is faster on all queries belonging to $\mathcal{C}_{2-6}$ (except $\mathcal{Q}_{42}$ where the size of the transitive closure is small).

a) Further scalability banchmarks on Uniprot: We report on further scalability benchmarks where Dist-μ-RA and BigDatalog execution times are compared for each Uniprot query on generated uniprot_n graphs with varying sizes of 1M, 5M and 10M edges. Results are shown in Fig. 14. Results indicate that BigDatalog fails in 44 cases out of 75 query evaluations. Dist-μ-RA answers all of them and scales better.

Notice that, for comprehensive benchmarking, 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 records for uniprot_10M) and $\mathcal{Q}_{46}$ is one with the largest (around 1.5B records for uniprot_10M, which is 150 times the size of the graph).

F. Summary

Overall, for all query classes, Dist-μ-RA is significantly more efficient compared to GraphX. For query classes $\mathcal{C}_{2-6}$ and some queries in $\mathcal{C}_7$, Dist-μ-RA is more efficient than BigDatalog, especially for large intermediate query result sizes. For query class $\mathcal{C}_1$ and some queries in $\mathcal{C}_7$, Dist-μ-RA and BigDatalog have a comparable performance. Our empirical findings tend to indicate that for these cases the various optimizations techniques of Dist-μ-RA and Bigdatalog have limited impact.

V. 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 both data and computations. We examine and compare to the closest related works along these two aspects below.

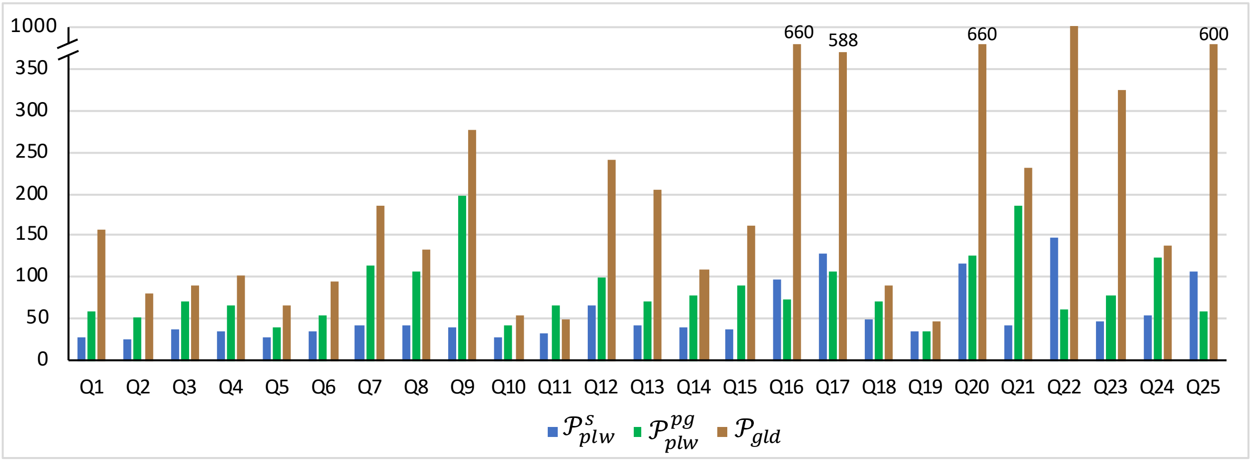


Figure 9. Running times of $\mathcal{P}_{\texttt{plw}}$ and $\mathcal{P}_{\texttt{gld}}$ plans on Yago.

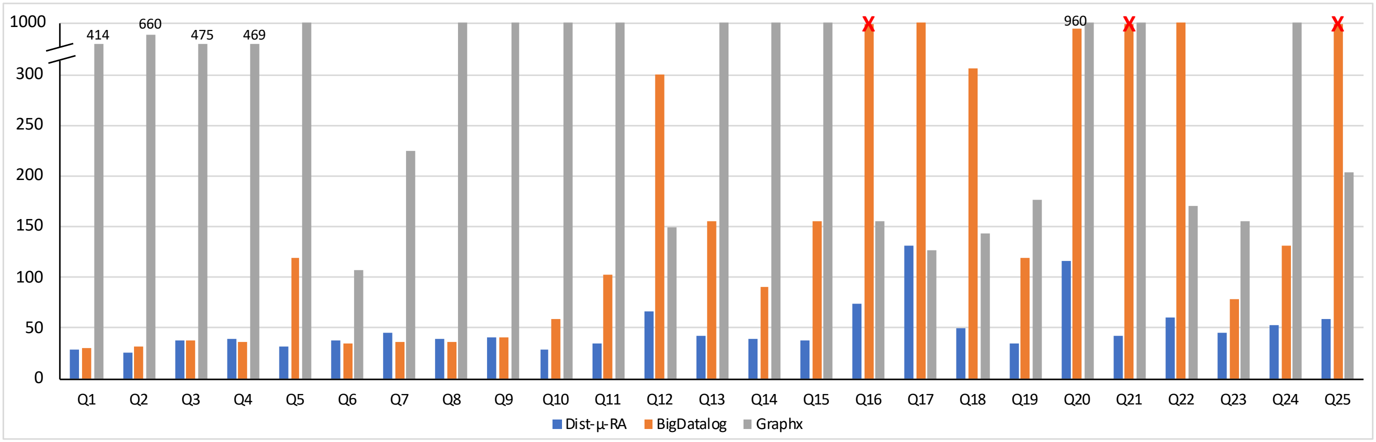


Figure 10. Running times on Yago. A timeout is set at 1,000 s.

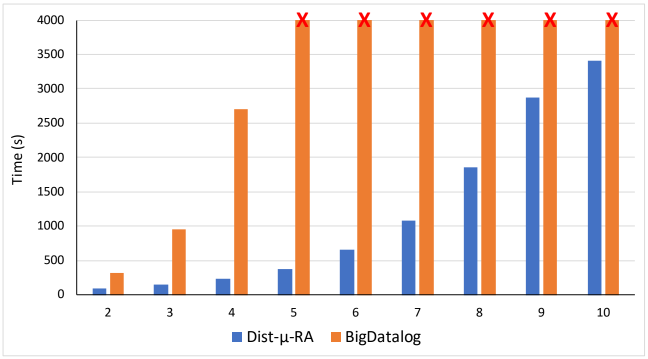


Figure 11. Evaluation times for concatenated closure queries.

For (i) we have choosen to build our system using μ-RA [11] since it offers more optimization opportunities without sacrifying expressivity, in particular with respect to approaches based on Datalog and RA [11]. In Datalog, Magic Sets [40], [41], recently improved by Demand Transformation [42], are well-known optimization techniques for recursive programs. The optimizations provided by Magic Sets and Demand Transformations are equivalent to pushing selections and projections in μ-RA. However, there is no Datalog equivalent to merging fixpoints as in [11]. 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 [43]. Since Datalog engines use heuristics to combine optimization techniques, optimizations are not always performed as observed in [11]. In all cases, combinations of Datalog optimizations lack the ability to merge fixpoints. In summary, Dist-μ-RA relies on μ-RA which enables optimizations at the logical level that other systems are unable to achieve. However, μ-RA [11] has been only proposed for the centralized setting.

Concerning the distribution aspect (ii), the seminal systems MapMeduce and Dryad are known to be inefficient for iterative applications [4]. Spark [3] and Flink [4] were introduced to

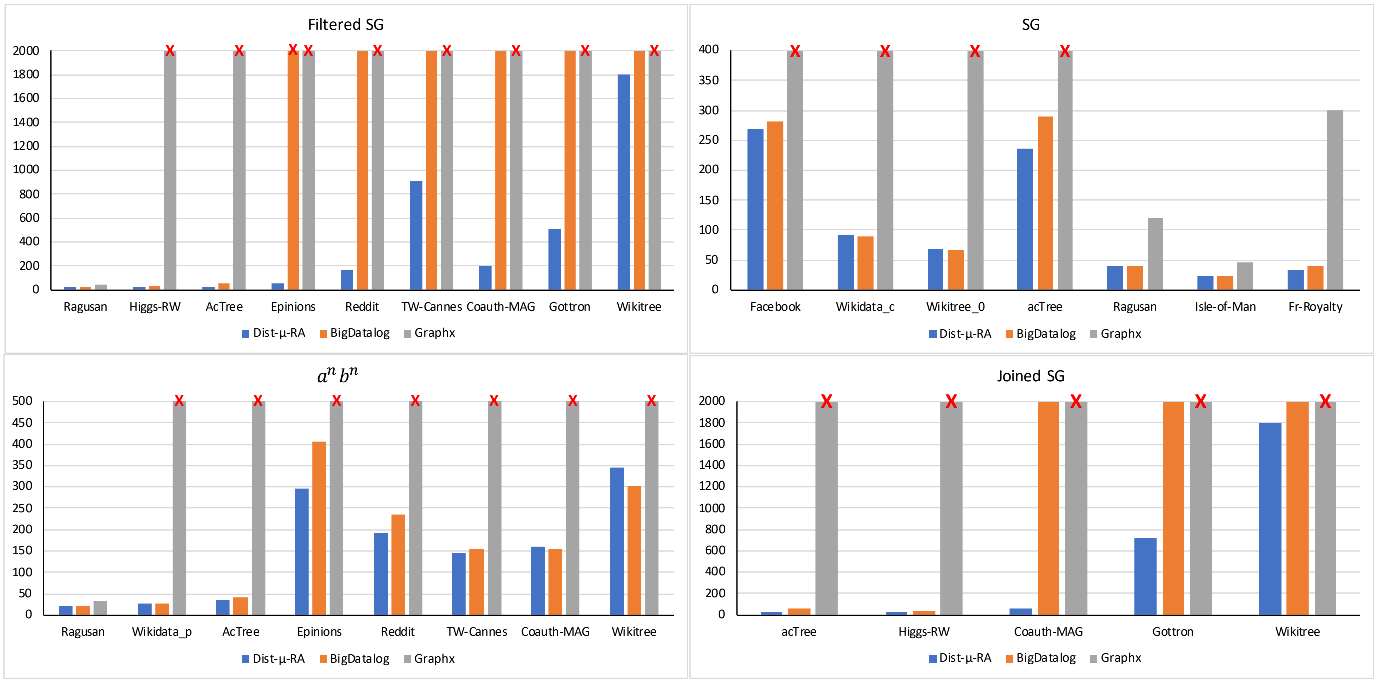


Figure 12. μ-RA queries running times. A timeout is set to 2000s.

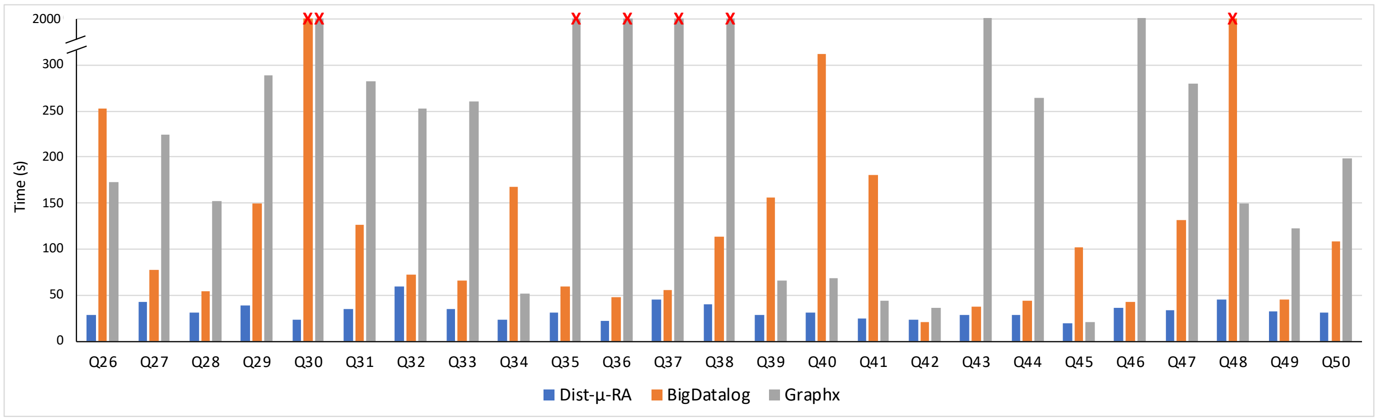


Figure 13. Running times on uniprot_1M. A timeout is set to 2,000 s.

improve upon these systems and became prevalent for large scale and data-parallel computations. Work in [44] proposes a technique that improves Spark task scheduling and thus performance 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 [45] and Powergraph [46]. 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. [47] 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,

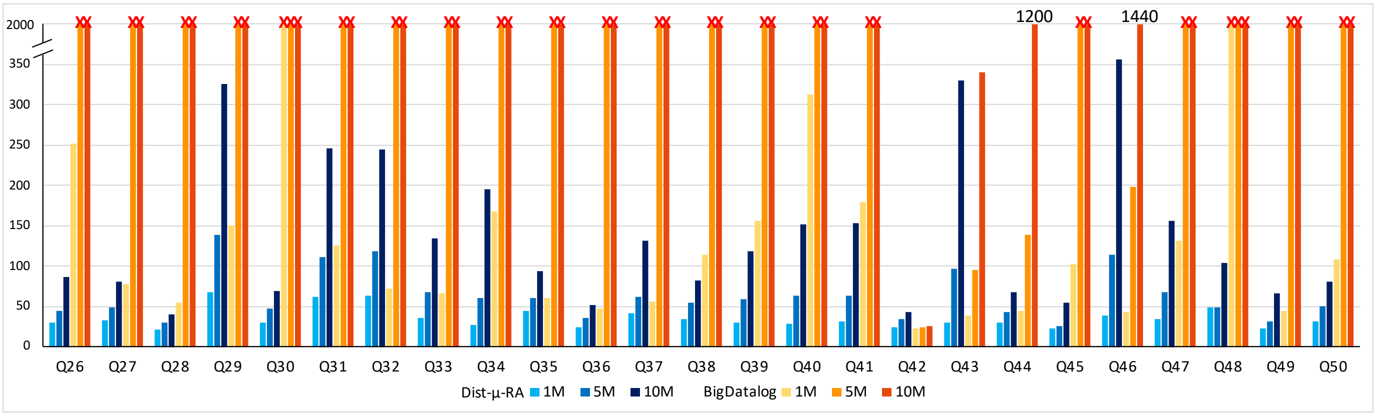


Figure 14. Dist-μ-RA and BigDatalog running times on Uniprot graphs of different sizes

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 happen in every recursion superstep, 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 [22] library enables the user to write SQL queries and process relational data using `Datasets` or `DataFrames`. However, recursion is not supported. DryadLINQ [48] that exposes a declarative query language on top of Dryad (or Pig Latin [49] on top of Hadoop MapReduce) has the same limitation. TitanDB [50] 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, these systems do not provide optimization techniques comparable to the ones we propose.

SociaLite [51] 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 [52] is a distributed system that supports a subset of Datalog extended with aggregation. Queries are translated into query plans executed on a parallel relational engine. Myria supports incremental evaluation of recursion and provides synchronous and asynchronous modes. It does not support advanced logical optimizations of the recursive query plan like pushing joins in fixpoints nor merging fixpoints. Myria does not either propose a distribution plan equivalent to $\mathcal{P}_{\texttt{plw}}$.

RaSQL [53] proposes an extension of SQL with some aggregate operations in recursion. Queries are compiled to Spark SQL to be distributed and executed on Spark. RaSQL does not propose rules to push selections in the fixpoint operator nor to merge fixpoints. RaSQL 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 benchmarking.

BigDatalog [13] is a recursive Datalog engine that runs on Spark. It uses the Datalog GPS technique [54] 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 IV.

VI. Conclusion

We propose a new approach for the evaluation of recursive algebraic terms in a distributed manner. It relies on an technique capable of generating independent parallel loops on the worker nodes in a cluster of machines instead of executing a global loop on the driver node. The advantage of the parallel local loops is a minimization of the amount of data shuffled between worker nodes. This reduces communication costs and significantly improves overall query evaluation time. We applied this approach to recursive graph queries on real and synthetic datasets. Experimental results show that the proposed approach is more efficient than the state-of-the-art.

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." february 2022. [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*. ACM, 2020, pp. 681–697.

[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 system implementation." february 2022. [Online]. Available: https://gitlab.inria.fr/tyrex-public/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] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, P. A. Bernstein, Ed. ACM, 1979, pp. 23–34. [Online]. Available: https://doi.org/10.1145/582095.582099

[22] 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

[23] ——, "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

[24] 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/

[25] "The colorado index of complex networks (icon)," february 2022. [Online]. Available: https://icon.colorado.edu/#!/

[26] M. Fire and Y. Elovici, "Data mining of online genealogy datasets for revealing lifespan patterns in human population," *ACM Trans. Intell. Syst. Technol.*, vol. 6, no. 2, mar 2015. [Online]. Available: https://doi.org/10.1145/2700464

[27] J. Kunegis, "Konect: the koblenz network collection," 05 2013, pp. 1343–1350.

[28] J. Liénard, T. Achakulvisut, D. Acuna, and S. David, "Intellectual synthesis in mentorship determines success in academic careers," *Nature Communications*, vol. 9, 11 2018.

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

[30] M. De Domenico and E. Altmann, "Unraveling the origin of social bursts in collective attention," *Scientific Reports*, vol. 10, p. 4629, 03 2020.

[31] "Wikidata the free knowledge base," february 2022. [Online]. Available: https://www.wikidata.org/wiki/Wikidata:Main_Page

[32] N. Halliwell, F. Gandon, and F. Lecue, "User Scored Evaluation of Non-Unique Explanations for Relational Graph Convolutional Network Link Prediction on Knowledge Graphs," in *International Conference on Knowledge Capture*, Virtual Event, United States, Dec. 2021. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03402766

[33] 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

[34] 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

[35] 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.

[36] "Bigdatalog repository." february 2022. [Online]. Available: https://github.com/ashkapsky/BigDatalog

[37] 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.

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

[39] 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

[40] 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

[41] 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

[42] 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.

[43] 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

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

[45] 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

[46] 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.

[47] 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.

[48] 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.

[49] 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

[50] "Titan distributed graph database." february 2022. [Online]. Available: http://titan.thinkaurelius.com/

[51] 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

[52] 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

[53] 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

[54] 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