# DatalogRA : Datalog with Recursive Aggregation in the Spark RDD Model

Marek Rogala
Institute of Informatics
University of Warsaw, Poland
marrogala@gmail.com

Jan Hidders
Vrije University Brussel,
Belgium
jan.hidders@vub.ac.be

Jacek Sroka
Institute of Informatics
University of Warsaw, Poland
sroka@mimuw.edu.pl

## ABSTRACT

Distributed computations on graphs are becoming increasingly important with the emergence of large graphs such as social networks and the Web that contain huge amounts of useful information. Computations can be easily distributed with the use of specialised frameworks like Hadoop with MapReduce, Giraph/Pregel or GraphLab. Yet, declarative, query-like, but at the same time efficient solutions are lacking. Programmers are needed to code all computations by hand and manually optimise each individual program.

This paper presents an implementation of a tool which extends a distributed computations platform, Apache Spark, with the capability of executing queries written in a variant of a declarative query language, Datalog, especially extended to better support graph algorithms.

This approach makes it possible to express graph algorithms in a declarative query language, accessible to a broader group of users than typical programming languages, and execute them on an existing infrastructure for distributed computations.

## Keywords

Spark, Datalog, RDD, Resilient Distributed Datasets, Socialite, graph processing, declarative

## 1. INTRODUCTION

Data analysis tasks on real-world web-scale datasets often involve analysing properties of the graphs represented by those datasets. This includes computing graph queries such as *Shortest Path*, *PageRank*, *Mutual Neighbours* (finding mutual neighbours of two nodes), *Connected Components*, *Triangles*, *Clustering Coefficients* and *Betweenness Centrality*. Much effort is being put into developing frameworks that allow users to compute these efficiently, specifically on top of frameworks for parallelised execution of computations on clusters of computational nodes such as Hadoop/MapReduce, extensions of MapReduce for iterations such as Haloop [1], vertex-oriented systems like Pregel [4] and Giraph, or in-memory systems like Spark [12]. These frameworks typically do their computations in several iterations, where each iteration consists of a computation distributed over different nodes. When an algorithm requires many iterations, an important distinction is whether between the iterations the nodes can keep the intermediate result in memory or have to store them on disk. In the first case the computation can become faster but may also require checkpointing to deal with failing nodes. Where MapReduce is assuming the second case, the other frameworks focus more on the first case.

Programming graph analytics with those frameworks requires considerable skill and knowledge of the distributed algorithms, e.g., [7]. For more value-oriented analytics declarative languages have been developed and implemented, such as for example Pig [6], which allow the user to efficiently specify and execute certain analytical tasks without deep algorithmic knowledge. Ideally there should for graph analytical tasks be also such a high-level declarative programming language that has the expressive power to specify the previously mentioned graph queries, but also can be effectively optimised.

A promising approach for such a language is the extension of Datalog with recursive aggregation and certain arithmetic operators, as discussed in earlier work by Shkapsky et al. [10] and Lam et al. [3], which both in turn are based on earlier research on allowing aggregation and multisets in Datalog by Mumick et al. [5]. The resulting language is concise and powerful, and at the same time leverages existing work on optimising its execution [11]. Moreover, the parallelised execution of Datalog has been well studied in the past, for example by Ganguly et al. in [2] and by Zhang et al. in [13].

The extension of Datalog proposed by Lam et al. proposed in [3], called *Socialite*, is provided with a parallelised implementation in [8]. It applies certain forms of parallelised Datalog execution optimisation such as range-based and hash-based sharding, and message batching. Moreover, it performs optimisations specific for its type of aggregation semantics such as Delta Stepping.

In this work we take essentially the same extension of Datalog and provide an alternative implementation on top of the Spark framework. This is an open and mature framework that is aimed at iterative computations and based on the notion of Resilient Distributed Dataset (RDD) which offers an abstract model for distributed data. It offers extensions such as Spark SQL, which lets one efficiently query structured data, and GraphX which unifies Extract-Transform-Load processes, exploratory analysis, and iterative graph

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*GRADES 2016, June 24 2016, Redwood Shores, CA, USA*

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4780-8/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2960414.2960417

computation within a single system. Next to providing a platform that easily integrates with other types of data analytic processing, we believe it offers an interesting environment to investigating and comparing different types of optimisations, more so then the implementation in [8] which is built from scratch and implements for example its own form of checkpointing.

Next to an alternative implementation, we present in this paper also a more refined definition of the syntax and semantics of the language we implement, and specifically we give a precise definition of which programs with recursive aggregation are considered to be well-defined. Moreover, we show that the performance of the system is competitive with comparable systems.

The remainder of this paper is organised as follows. In Section 2 we recall Datalog and semi-naive evaluation. In Section 3 we introduce the syntax and semantics of DatalogRA. In Section 4 we discuss the implementation of DatalogRA in Spark, and finally in Section 5 we describe some experiments to evaluation the relative performance of the implementation.

## 2. PLAIN DATALOG AND ITS EVALUATION

Datalog programs express queries over *relational databases* which we will define here as a finite sets of *facts* where facts are of the form $\text{R}(v_1, \dots, v_n)$ where R is a *relation name* and $(v_1, \dots, v_n)$ a vector of *domain values*. We will assume that the set of domain values is always finite. An example of such a relational database would be $\{\text{A}(1, 2), \text{A}(2, 3), \text{B}(3, 1)\}$, assuming that the set of relation names is $\{\text{A}, \text{B}\}$ and the set of domain values is $\{1, 2, 3\}$.

Basic Datalog programs consist of a set of *rules* where a rule is an expression of the form:

$$\text{R}(\bar{x}) \text{ :- } \text{S}_1(\bar{y}_1), \dots, \text{S}_n(\bar{y}_n).$$

where $n \geq 1$, $\text{R}, \text{S}_1, \dots, \text{S}_n$ are relation names and $\bar{x}, \bar{y}_1, \dots \bar{y}_n$ are tuples of *variables* and *constants* (i.e., domain values). We call $\text{R}(\bar{x})$ the *head* of the rule, while $\text{S}_1(x_1), \dots, \text{S}_n(x_n)$ is called the *body* of the rule and each element of the body is called a *subgoal* of the rule. Every rule in a Datalog program must be *safe*, which means that each variable appearing in the head appears in at least one of the subgoals.

The meaning of a rule can be described as a first-order logic formula of the form:

$$\forall_{\bar{z}} : \text{R}(\bar{x}) \Leftarrow \text{S}_1(\bar{y}_1) \wedge \dots \wedge \text{S}_n(\bar{y}_n)$$

where $\bar{z}$ is a vector containing exactly all variables in the vectors $\bar{x}, \bar{y}_1, \dots, \bar{y}_n$. We say that a rule holds for a relational database if the corresponding formula holds for (the first-order model represented by) the database. The result of applying a Datalog program $P$ to a relational database $D$, denoted as $P(D)$, is then defined as the minimal superset of $D$ for which all rules in $P$ hold. It can be shown that such a superset always exists and is indeed unique.

A more operational way of defining the semantics of a program $P$ is based on fixed points of a consequence function. We define the *result of a rule* $r$ for a database $D$, as

$$r(D) = \{\text{R}(\bar{x}) \mid \text{S}_1(\bar{y}_1) \in D, \dots, \text{S}_n(\bar{y}_n) \in D\}.$$

Observe that this is a well-defined set if $r$ is a safe rule. Intuitively it describes all the facts that are implied directly by the rule $r$ given the database $D$.

Then we define the *immediate consequence of a program* $P$ for database $D$, as $T_P(D) = D \cup \bigcup_{r \in P} r(D)$. It is not hard to see that for every database $D$ it holds that all rules in $P$ hold for $D$ iff $D$ is a fixed point of $T_P$. It can also be observed that $T_P$ is *monotonic*, i.e., if $D_1 \subseteq D_2$ then $T_P(D_1) \subseteq T_P(D_2)$. Moreover, since we assume that the set of domain values is finite, there are only a finite number of possible databases. It therefore follows that we can compute the minimal fixed point of $T_P$ that is a superset of $D$ by repeatedly applying $T_P$ to $D$ until the result no longer changes, i.e., by determining the smallest natural number $i$ such that $T_P^{i+1}(D) = T_P^i(D)$.

There are many ways to implement Datalog, but we will focus here on bottom-up evaluation techniques where we more or less iteratively compute the applications of $T_P$, and in particular on *semi-naive evaluation* of Datalog. The basic idea of semi-naive evaluation is that when computing the result of a rule we take into account which relations where extended in the preceding iteration and what the newly added tuples where. For example, if we have a rule $\text{R}(x, y) \text{ :- } \text{S}(x, y, z), \text{R}(z, 2), \text{R}(y, z)$ and we know that $\text{R}'$ contains the recently added tuples, then we can compute the tuples added only by this rule by computing the union of the query $\{(x, y) \mid \text{S}(x, y, z) \wedge \text{R}'(z, 2), \text{R}(y, z)\}$ and the query $\{(x, y) \mid \text{S}(x, y, z) \wedge \text{R}(z, 2) \wedge \text{R}'(y, z)\}$. Observe that the resulting R after adding the tuples computed by these queries will be the same as for the evaluation of the rule under the conventional bottom-up evaluation. When adding these records, it can be checked which ones were not already in R, to compute the relation $\text{R}'$ for the next step. In general the evaluation of a rule is split into a union of $m$ queries, where each query is defined by the rule as usual but with the $m^{th}$ subgoal replaced by the relation that contains the newly added tuples in the preceding iteration. Note that this approach indeed often prevents recomputing a large part of the result in R, although it can stil happen that the same tuple is computed more than once.

## 3. DatalogRA

DatalogRA is founded on SociaLite [3, 8], which in turn is a graph query language based on Datalog which provides several optimisations and extensions to Datalog for expressing and executing basic graph algorithms. The key extension in SociaLite over Datalog are *recursive aggregate functions*. This extension allows for correctly solving many graph problems, e.g., finding shortest paths from single source without finding all possible paths first and avoiding infinite execution when a cycle exists in the graph.

Like Socialite, DatalogRA allows aggregation to be combined with recursion under some conditions. A DatalogRA program is essentially a Datalog program where optionally for a relation an *aggregation function* is defined for the last column. Figure 1 presents a DatalogRA program for computing the shortest paths from node 1 to all nodes in a graph defined by the EDGE relation.

As in Socialite, the rules are preceded by a declaration section that declares all the involved relations, their names and types and names of their columns. The name is only added as a comment to illustrate the intended meaning. In addition it is also indicated if the last column is aggregated or not, and if so what aggregation function is used for it. This is different from Socialite where the aggregation functions are indicated within the rules. However, recursive aggregation only makes sense when it is applied to whole relations,

so we propose defining it in relation declarations as a more intuitive approach.

$\textsc{Edge}(\text{int } src, \text{int } sink, \text{int } len)$
$\textsc{Path}(\text{int } target, \text{int } dist \text{ aggregate } \textsc{Min})$

$$\textsc{Path}(t, d) \quad :- \quad t = 1, d = 0.$$
$$\textsc{Path}(t, d) \quad :- \quad \textsc{Path}(s, d_1), \textsc{Edge}(s, t, d_2), d = d_1 + d_2.$$

**Figure 1: DatalogRA query for computing the shortest paths from node 1 to other nodes.**

As a consequence, a program $P$ in DatalogRA does not only consist of a set of rules but also an indication of in which relations the last column is being aggregated and what the aggregation operator is. The restriction of aggregation to the last column is only there to keep the current implementation simple and has no deeper fundamental reason nor does it essentially change the expressive power. In addition to aggregation, DatalogRA also allows stratified negation and arithmetic equations in the usual way.

The semantics of a DatalogRA program $P$ without negation can be described as the minimal fixed point of an immediate consequence operator which is the composition $\Gamma_P \circ \hat{T}_P$ where $\hat{T}_P$ computes the direct consequences of rules in $P$ while ignoring aggregation, and $\Gamma_P$ is a function that aggregates as specified in $P$.

We first define $\hat{T}_P$ which is similar to $T_P$ except that it returns a bag where facts are duplicated if they are derived in multiple ways, similar to how multiplicity is defined in [5] in terms of number derivation trees. For this purpose we define the notion of *result bag of a rule* $r$ for a database $D$, denoted as $\hat{r}(D)$, as the bag over the set $r(D)$ where the multiplicity of each element $\text{R}(\bar{x})$ is equal to the number of valuations $f$, i.e., functions from the variables in $\bar{y}_1, \ldots, \bar{y}_n$ to domain values, such that $\text{S}_1(f(\bar{y}_1)) \in D, \ldots, \text{S}_n(f(\bar{y}_n)) \in D$. Based on this we define the *bag of immediate consequences of a program* $P$ for database $D$, as $\hat{T}_P(D) = D \uplus \biguplus_{r \in P} \hat{r}(D)$ where $\uplus$ is the additive bag union.

The aggregation operator $\Gamma_P$ maps bags of facts to sets of facts in the following way. If a relation $R$ is aggregated in $P$ then for each vector $\bar{x}$ such that there is at least one fact of the form $R(\bar{x}, y)$ in the input, it replaces these facts with $R(\bar{x}, G(\bar{Y}))$ where $G$ is the aggregation function indicated in $P$ for the relation and $\bar{Y}$ is the bag of domain values where the multiplicity of an element $y$ is the multiplicity of $R(\bar{x}, y)$ in the input. If a relation $R$ is *not* aggregated in $P$ then it removes duplicate facts for this relation, i.e., for each vector $\bar{x}$ such that there is at least one fact of the form $R(\bar{x})$ in the input, it replaces all these facts with a single fact $R(\bar{x})$. Note that the final result of $\Gamma_P$ is always without duplicates for both aggregated and non-aggregated relations and therefore indeed a set.

Having defined the semantics of a program $P$ applied to database $D$ as the first fixed point of $\Gamma_P \circ \hat{T}_P$ the next question is if this always exists and if it is minimal in some sense. Since the set of possible databases is finite, because we assume a finite set of domain values, it is sufficient to show that there is some partial ordering over databases for which $\Gamma_P \circ \hat{T}_P$ is monotonic. However, programs with aggregation will no longer be monotonic with respect to the subset ordering. Fortunately, as is discussed in [3] it is sometimes possible to define an alternative ordering for which the program is monotonic on the basis of the chosen aggregation operators. We explain this in what follows.

We consider aggregation operators, say $G$, that are based on a binary operator, say $\oplus_G$, that is commutative and associative. The result of applying $G$ to a non-empty bag $\{\!\{a_1, \ldots, a_n\}\!\}$ is defined as $a_1 \oplus_G \ldots \oplus_G a_n$. Based on this we can define a pre-order $\sqsubseteq_G$ over the set of domain values for the aggregation such that $a \sqsubseteq_G b$ iff $a = b$ or there is a $c$ such that $a \oplus_G c = b$. For example, for the $\textsc{Max}$ operator that ordering is $\leq$, for $\textsc{Min}$ it is $\geq$ and for $\textsc{Sum}$ over nonnegative integers it is also $\leq$. Note that not for all aggregation operators this pre-order is a partial order. For example, for $\textsc{Sum}$ over all integers it holds that both $-1 \sqsubseteq_{\textsc{Sum}} 1$ and $1 \sqsubseteq_{\textsc{Sum}} -1$ since $-1 \oplus_{\textsc{Sum}} 2 = 1$ and $1 \oplus_{\textsc{Sum}} -2 = -1$. We will consider only those aggregation operators where the associated ordering is a partial order.

This allows us now to define a partial order over facts and databases, given a program $P$. For facts we define $\sqsubseteq_P$ for comparing facts concerning relation $R$ such that (1) if relation $R$ has aggregation operator $G$ in $P$ then $R(\bar{x}, y) \sqsubseteq_P R(\bar{x}', y')$ iff $\bar{x} = \bar{x}'$ and $y \sqsubseteq_G y'$ and (2) if $R$ has no aggregation operator in $P$ then $R(\bar{x}) \sqsubseteq_P R(\bar{x}')$ iff $\bar{x} = \bar{x}'$. We then define for databases the ordering such that $D_1 \sqsubseteq_P D_2$ holds iff for all $R(\bar{x}) \in D_1$ there is a fact $R(\bar{x}') \in D_2$ such that $R(\bar{x}) \sqsubseteq_P R(\bar{x}')$. It can be verified that this defines a partial order. It is now for this ordering that we will require that programs are monotonic, i.e, that $\Gamma_P \circ \hat{T}_P$ is monotonic under $\sqsubseteq_P$, which will ensure that there is a well-defined notion of minimal fixed point.

The requirement for monotonicity for $\Gamma_P \circ \hat{T}_P$ can be simplified if we restrict ourselves to aggregation operators $G$ for which $\oplus_G$ is idempotent, i.e., $a \oplus_G a = a$ for all a in the domain of $\oplus_G$, such as holds for example for $\textsc{Min}$ and $\textsc{Max}$. In that case the aggregation operator is essentially operating on sets, rather then bags, and so it holds that $\Gamma_P \circ \hat{T}_P = \Gamma_P \circ T_P$. Since $\Gamma_P$ is always monotonic under $\sqsubseteq_P$, it is then sufficient to require that $T_P$ is monotonic under $\sqsubseteq_P$. So, for example, for the program in Figure 1 we only need to verify that the rules, while ignoring the aggregation, are monotonic under $\leq$, which they indeed are. Note that this is only a sufficient condition; the exact characterisation and complexity of deciding the monotonicity property for a program is a subject for further research.

For a program with only idempotent aggregation operators, i.e., where $\oplus_G$ is idempotent, it is possible to straightforwardly implement a semi-naive evaluation strategy. We only need to maintain after each iteration the current content of a relation as produced by applying $\Gamma_P \circ T_P$. After each iteration we determine the "new facts" where a new fact is defined as a fact $R(\bar{x})$ for which there was not already a larger fact, i.e., a fact $R(\bar{x}')$ such that $R(\bar{x}) \sqsubseteq_P R(\bar{x}')$. As usual in semi-naive evaluation we can then compute which other new facts are implied by these new facts and add them to the database. This can be done by determining this for $T_P$ in the usual way, and then adding the additional facts to the database and reapplying $\Gamma_P$ (which can remove facts). Because the aggregation operator is idempotent, this will for monotonic $P$ produce the same result as applying $\Gamma_P \circ \hat{T}_P$.

The restrictions mentioned above on the usage and type of aggregation operators only apply when the aggregation is used in a recursive manner. We can make this more precise as follows. Every program $P$ defines a dependency graph of the relation names where there is an edge from $R$ to $R'$ if $R'$

is mentioned in the body of one of the rules in $P$ where $R$ is in the head. We then require that for every cycle in this graph it holds that all the relations in the cycle either have no associated aggregation operator or one that is restricted as above, which in the current implementation are only the MIN and MAX operators. In the same way we also allow negation in a restricted fashion such that the program is stratified: for every cycle it holds that none of the relations in the cycle depends negatively on a relation in the cycle, i.e., there is not a rule that mentions one relation in the head and the other in the tail such that it is there immediately preceded by a negation.

These restrictions on using arbitrary aggregation and negation allow us to stratify a program, i.e., partition the relations into a sequence of sets of relations (the strata) such that within each stratum the relations depend only on relations within that stratum or a stratum earlier in the sequence. As usual this allows us to give a well-defined semantics to the total program, assuming that the rules associated with each stratum have a well-defined semantics.

## 4. IMPLEMENTATION IN SPARK

The key concept in Spark is *Resilient Distributed Dataset* (RDD), which is an abstraction of distributed memory and lets the programmer perform distributed computations. They are stored in a way that is transparent to the user and assures fault tolerance. RDDs provide a *coarse-grained* interface, i.e., operations that apply to the whole dataset, such as *map*, *filter* and *join*. This allows for achieving fault-tolerance by storing only the history of operations that were used to build a dataset, called its *lineage*, instead of replicating the data to be able to recover it. An additional advantage is that the RDDs do not need to be materialised, unless it is actually necessary. Since parallel computations generally apply some transformation to multiple elements of a dataset, they can in most cases be expressed easily with coarse-grained operation on datasets.

The main component DatalogRA provides is the *Database* class which represents a set of relations which can be created from regular RDDs. Database objects are equipped with a method *datalog*, which performs a Datalog query on this database. The result of the query is a new Database, from which individual relations can be extracted as RDDs. This allows for the construction of data processing pipelines intermixing several Spark components, e.g., adding pre- and post-processing around DatalogRA computation. Figure 2 shows an example of a DatalogRA query in a Spark program using DatalogRA which is implemented in the form of a Datalog API for Spark.

When a Datalog query is performed on the Database object, it needs to be translated into a sequence of Spark transformations which eventually produce a new Database object containing the result of the query. In this section, we show how this translation is made.

Each Relation object has a name and an RDD of Facts, which in turn are represented as arrays. All Facts in a relation are required to have the same arity, which is ensured when the Relation object is created. In the current implementation we only allow facts over integers, but this can be easily extended.

To execute a DatalogRA program we first analyse it to produce an AST representation of the program. All correctness requirements are verified at this stage. Then, the

```
1  val edgesRdd = ... // Read from HDFS or computed using Spark
2
3  val database = Database(Relation.ternary("Edge", edgesRdd))
4  val resultDatabase = database.datalog("""
5      declare Path(int v, int dist aggregate Min).
6      Path(x, d) :- s == 1, Edge(s, x, d).
7      Path(x, d) :- Path(y, da), Edge(y, x, db), d = da + db.
8  """)
9  val resultPathsRdd = resultDatabase("Path")
10
11 ... // Save or use resultPathsRdd as any RDD.
```

**Figure 2: Example of Datalog query for computing single source shortest paths embedded in a Spark program.**

program's rules are stratified, i.e., divided into a sequence of strata, which can be evaluated one by one. Finally, each stratum is evaluated iteratively until a fix-point is reached. In each step of the evaluation, all rules in the stratum are applied to the current state of the database. In this section, we describe how a single step is performed.

A single rule consists of a head and a sequence of subgoals. The task is to evaluate this rule given an RDD representing the current state of the database, i.e., to compute an RDD of all facts that can be inferred from the current state. This is done in two steps: first, all valuations satisfying the rule body are computed, and then each such valuation is converted to a fact based on the head of the rule. To represent mappings of variables to their values during query evaluation we use *Valuation* objects, which are not available to the end user. To find all valuations satisfying a rule body, we process all its subgoals one by one from left to right. To deal with arithmetical goals, where we have input and output positions, we sort the subgoals such that all inputs of a goal are available before it is evaluated. We start with an RDD of valuations containing an empty valuation, i.e., a valuation containing no variable assignments. Next, the subgoals are applied sequentially. Each subgoal is evaluated on the current RDD of valuations. This returns a new RDD of valuations all of which satisfy this subgoal. This new RDD is then used for the next subgoal, until all subgoals are processed.

How a subgoal is evaluated depends on its type, which is either *relational subgoal*, *arithmetic comparison* or *assignment subgoal*. In case of relational subgoals, the relation is first converted into valuations of variables in the subgoal using a *map* transformation. The resulting RDD is then joined with the starting valuations on the matching variables using the *join* transformation on RDDs and converted to a new RDD of valuations using another *map* transformation. Arithmetic comparison subgoals are translated into a *filter* transformation on the valuations RDD. Finally, assignment subgoals are translated into *map* transformations on the valuations RDD, which adds the newly computed value to each valuation.

Each rule head consists of a relation name and a sequence of variable names, e.g., $Path(v, d)$. Language constraints ensure that each valuation satisfying the rule body contains values for all variables appearing in the head. A map transformation is used to convert an RDD of valuations into an RDD of facts.

The procedure described above finds all facts that can

be inferred using a single rule. This can be done for each rule within a stratum. The next step in one iteration of evaluation is to merge the results obtained from the rules with the database from the previous step.

For relations without aggregation, this is achieved by performing a *union* transformation on the RDDs containing new facts for a given relation and the corresponding relation in the current database. Then, a *distinct* transformation is applied to remove duplicated facts.

For aggregated relations, this is slightly different. After performing a *union* of the new facts and the current relation contents, the facts are grouped by the qualifying parameters and the aggregated value is computed. This is done by the following three transformations:

1. *map* to split the facts into qualifying parameters and the aggregated values,

2. *reduceByKey* to group facts by qualifying parameters and apply the aggregation function to the set of aggregated values in each group,

3. *map* to merge the qualifying parameters and the computed value in each group back into facts.

The main built-in optimisation is the semi-naive evaluation. In all steps starting from the second one, the delta database is used, which contains only the new facts inferred in the previous step. We evaluate each rule body several times, each time using the delta database in place of a subsequent relational subgoal, and the full database for the other subgoals. The delta database for the next step could be naively computed as the difference between full databases from the current step and the previous step. Instead, it is more efficiently computed by marking the new facts when the facts inferred in a step are merged into the current database.

Other optimisations include detecting non-recursive strata and caching intermediate results. Non-recursive strata require only one iteration and need no check to see if the fixed point has been reached. For caching intermediate results we proceed as follows. If a relation is used in a subgoal, it needs to be converted into an RDD of valuations for joins to be performed. Within a given stratum, however, only the relations defined in this particular stratum can change. Therefore, for each subgoal referring to a relation from another stratum the corresponding valuations are found once and persisted. This makes it possible to avoid repeated work.

A practical implementation of the above procedure requires some RDD-specific issues to be handled:

- The new full database and delta database are marked to be cached so that they are stored in memory and not recomputed each time they are used,

- Results obtained from an iteration are materialised, i.e., actually computed. This allows for the results of the previous iteration to be unpersisted, i.e., removed from cache. This helps reduce memory usage and increases performance,

- If many iterations are performed, the procedure can create RDDs with a very long lineage. Lineage is stored in RDDs so that they can be restored after a failure of a worker. Too long lineage can cause errors, so every several iterations all results are checkpointed to persistent storage, which causes the lineage to be cropped,

- Unlike most RDD transformations, the *join* changes the partitioning of data, i.e., the way the data is distributed between worker nodes, increasing the number of partitions. The number of partitions is reduced when it exceeds a certain limit using the *coalesce* action, as performance is negatively affected when there are too many partitions compared to the size of data.

# 5. EXPERIMENTS AND EVALUATION

The performance of the tested implementation was compared with plain Spark programs solving the same problem. We were unfortunately unable to reliably run the equivalent programs in SociaLite to compare results with it. In addition, we compared the complexity of each solution as measured by the number of lines in each program.

DatalogRA was implemented as a fully working prototype. We evaluated it for several classic graph problems and compared it against plain Spark implementations which were written using Spark core methods and the GraphX extension. For each problem, both solutions were evaluated on Amazon EC2 clusters consisting of $n = 2$, 4, 8 and 16 worker nodes and one master node. Each node was a 2-core 64-bit machine with 7.5 GiB of RAM memory. In all experiments, a social graph of Twitter circles, which has 2.4M edges was used.

DatalogRA is not limited to a specific domain and can be used to express various types of distributed computations, but it is primarily intended for computations on large graphs such as social networks. Therefore, we have selected three common graph problems for performance tests of the prototype implementation: finding and counting triangles, dividing the graph into connected components, computing the shortest paths from a single source to all other vertices. The results of the experiments are shown in Figure 3.

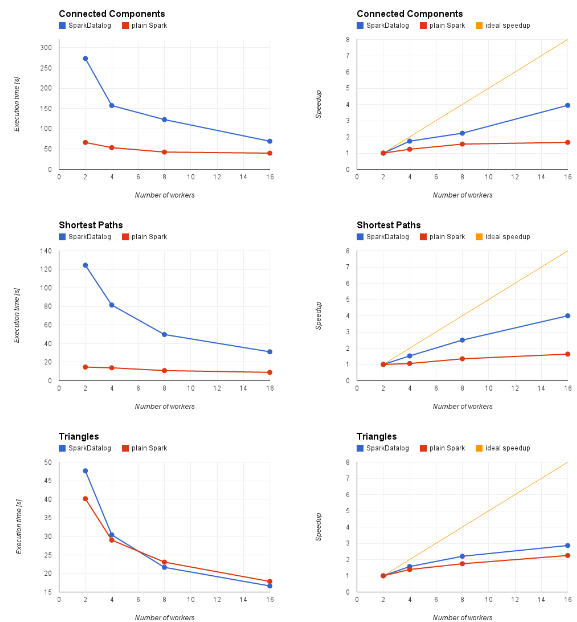


**Figure 3: Results of experiments.**

In the Triangles Count test case the execution time in SparkDatalog is very similar to dedicated Spark. The SparkDatalog versions are slower than dedicated Spark programs in both Shortest Paths and Connected Components, by a factor of approximately 8.5 to 3.5 in Shortest Paths and 4 to 1.7 in Connected Components. The speedups achieved were similar for both versions of each program. This shows that although the implemented solution is slower by some factor, it does parallelise. The speedups were slightly better for the SparkDatalog versions. This is probably because the additional overhead in SparkDatalog could also get parallelised. In Connected Components and Shortest Paths, the difference in execution times is the least with the greatest number of worker nodes.

An important goal for SparkDatalog is to provide programmers and non-programming analysts with the possibility to perform computations by writing declarative queries instead of implementing complicated, lengthy algorithms using the Pregel model or standard RDD transformations. Datalog versions are 1.4 to 3 times shorter than dedicated Spark. Most importantly, they are conceptually simpler since they only require a few declarative rules instead of expressing the problem in the vertex-centric Pregel model.

| | **plain Spark** | **SparkDatalog** |
|---|---|---|
| *Connected Components* | 11 | 6 |
| *Shortest Paths* | 12 | 4 |
| *Triangles* | 7 | 5 |

**Table 1: Number of lines of code in programs, excluding data loading and comments.**

We close with a brief discussion of related work. This work is directly based on that of distributed Socialite [8] but we were unfortunately not able to do a direct performance comparison. On the other hand our system does not have as its main goal to provide a faster implementation but rather to illustrate that it is possible to have a robust implementation on top of a well-known and well-tested framework. Other closely related work is the semi-naive implementation of unextended Datalog on Haloop [9] which optimises the execution of iterations by intelligent caching and specialised *join* and *difference* implementations. In DatalogRA similar optimisations are applied, but require less or no work since they are largely already provided by the Spark framework.

# 6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented DatalogRA, a version of Datalog extended with recursive aggregation, implemented in Spark's RDD model. This language allows end-users to query large graphs in a declarative fashion and in a well-studied language with well-defined semantics.

An advantage of the chosen architecture is the separation of concerns between managing distributed computation in a cluster as offered by Spark and Datalog-specific optimisations which, as we show, can be implemented in the Spark's RDD model. This combination makes it easy to study the effectiveness of further Datalog optimisation techniques. Moreover, the framework distributing the computations is mature and well tested. In addition its popularity guarantees easy integration with other tools from the open-source distributed computation stack. Finally, Spark programs can be readily executed on the main cloud platforms.

# 7. REFERENCES


[1] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, Sept. 2010.

[2] S. Ganguly, A. Silberschatz, and S. Tsur. A framework for the parallel processing of datalog queries. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 143–152, New York, NY, USA, 1990. ACM.

[3] M. S. Lam, S. Guo, and J. Seo. Socialite: Datalog extensions for efficient social network analysis. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 278–289, Washington, DC, USA, 2013. IEEE Computer Society.

[4] 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 SIGMOD 2010*, pages 135–146, New York, NY, USA, 2010. ACM.

[5] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pages 264–277, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[6] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of SIGMOD 2008*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[7] S. Salihoglu and J. Widom. Optimizing graph algorithms on pregel-like systems. *PVLDB*, 7(7):577–588, 2014.

[8] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *Proc. VLDB Endow.*, 6(14):1906–1917, Sept. 2013.

[9] M. Shaw, P. Koutris, B. Howe, and D. Suciu. Optimizing large-scale semi-naïve datalog evaluation in hadoop. In *Proceedings of the Second International Conference on Datalog in Academia and Industry*, Datalog 2.0'12, pages 165–176, Berlin, Heidelberg, 2012. Springer-Verlag.

[10] A. Shkapsky, K. Zeng, and C. Zaniolo. Graph queries in a next-generation datalog system. *Proc. VLDB Endow.*, 6(12):1258–1261, Aug. 2013.

[11] K. T. Tekle, M. Gorbovitski, and Y. A. Liu. Graph queries through datalog optimizations. In *Proceedings of PPDP'12*, pages 25–34, New York, NY, USA, 2010. ACM.

[12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[13] W. Zhang, K. Wang, and S.-C. Chau. Data partition and parallel evaluation of datalog programs. *IEEE Trans. on Knowl. and Data Eng.*, 7(1):163–176, Feb. 1995.