Toward Executing Datalog on Big Data Platforms

Andrea Cuteri, Giuseppe Mazzotta* and Francesco Ricca

Department of Mathematics and Computer Science, University of Calabria, Rende, Italy

Abstract

Modern databases must be capable of querying massive volumes of data, as we live in the era of Big Data. At the same time, Datalog, a logic language at the roots of database theory, remains a powerful formalism for expressing complex queries in a declarative manner. In this paper, we move the first steps toward bridging these two worlds in an innovative way. In particular, we introduce an approach that compiles stratified Datalog programs in Spark jobs for execution on Big Data frameworks. Our approach aims at effectively combining the declarative nature of logic rules with the scalability of modern distributed systems.

Keywords

Datalog, Big Data, Spark

1. Introduction

Modern databases are increasingly required to manage and process massive volumes of data generated by diverse applications [1]. The continuous growth of data, both in size and complexity, has pushed traditional systems beyond their limits, necessitating new architectural and computational approaches. To efficiently store, retrieve, and analyze such vast datasets, modern databases often operate over large-scale distributed clusters. This trend has accelerated research into scalable systems capable of handling complex analytical tasks across distributed environments [2]. Consequently, designing efficient, fault-tolerant, and highly performant database solutions has become a critical focus in both academia and industry [1]. Among the prominent systems developed to address these challenges there, is Apache Spark [3], a widely adopted platform for large-scale data analytics. Spark enables efficient in-memory processing across distributed clusters, significantly accelerating the computation of complex analytics tasks. Its flexible architecture supports a variety of workloads, ranging from simple queries to advanced machine learning pipelines. However, implementing analytics in Spark often requires programmers to manually express the evaluation of queries using imperative programming constructs [1].

On the other hand, declarative query languages offer higher-level abstractions that allow users to specify what results they want without detailing how to compute them, leading to improved productivity and reduced potential for programming errors compared to imperative approaches. In the database landscape, Datalog has emerged as a foundational declarative language, deeply influencing both the theoretical underpinnings and practical applications of database systems [4]. Datalog allows the formulation of complex queries using a declarative logic-based formalism [5]. The limitations of traditional Datalog evaluation systems in handling Big Data, due to their inability to leverage modern distributed data management environments [6, 7], pave the way for a new generation of Datalog engines designed on top of scalable platforms like Spark, possibly unlocking unprecedented potential for declarative analytics at massive scale.

In this paper, we make a preliminary contribution toward this goal by presenting *Datalog2Spark*, a system that explores the compilation of Datalog queries into Spark programs for evaluation over computer clusters. In this first prototype, we focus on non-recursive Datalog programs with stratified negation and incorporate some useful aggregation functions. Both extensions were conceived by recognizing that (i) the support of data types is fundamental in database systems, and (ii) it has been

CILC 2025: 40th Italian Conference on Computational Logic, June 25–27, 2025, Alghero, Italy

*Corresponding author.

andrea.cuteri@unical.it (A. Cuteri); giuseppe.mazzotta@unical.it (G. Mazzotta); francesco.ricca@unical.it (F. Ricca)

0009-0000-7629-7347 (A. Cuteri); 0000-0003-0125-0477 (G. Mazzotta); 0000-0001-8218-3178 (F. Ricca)

observed how aggregation is crucial for effective data processing and analytics [6]. For this reason, we first extend the Datalog language with constructs for type definitions, on the lines of previous studies [8]. Furthermore, we support some aggregation functions designed to enhance the expressive power and practical utility of our tool. To assess the viability of our approach, we conducted a preliminary experimental evaluation of our implementation, comparing it with two main-memory engines capable of processing Datalog. The first system, DLV [9], is well known for extending a powerful Datalog engine into a comprehensive logic programming platform for declarative reasoning under the Answer Set Programming (ASP) paradigm. The second system, CLINGO [10], is an ASP solver that can also evaluate Datalog programs, leveraging its support for the broader ASP language. Obtained results confirm the viability of the approach, demonstrating its potential to bring the benefits of declarative programming to Big Data analytics while delivering acceptable performance on single-server environments compared to existing systems, and at the same time enabling seamless scale-out over modern distributed databases, and capability of executing queries with aggregates that cannot be handled by compared systems.

The remaining of this paper is structured as follows. In section 2 we review the basics of Datalog and Apache Spark. Section 3 presents an extension of the Datalog language as well as the novel compilation-based technique implemented by *Datalog2Spark*. Section 4 provides an experimental analysis. Finally Section 6 discusses related works and draws the conclusions.

2. Preliminaries

In this section we provide basic notions about Apache Spark and Datalog which are the core of the contribution of this paper.

2.1. Spark

Apache Spark is a unified engine allowing for parallel data processing on distributed clusters [11]. Such an engine provides a powerful, yet efficient, programming API, which allows distributed applications to combine (possibly) multiple workflows such as streaming data analysis, machine learning, structured data analysis, among others. The Spark programming API, available in different programming languages such as Java, Python, Scala, and R, is built on top of Resilient Distributed Datasets (RDDs) [12]. An RDD is an immutable and distributed in-memory dataset which represents the basic data abstraction in Spark. RDDs have then been refined creating another data abstraction called *Spark Dataset*. A Spark Dataset supports mainly two kinds of operations, namely *transformations* and *actions*. Transformations are all those operations that, once applied on a Dataset, generate a new Dataset obtained by applying the transformations to all the elements of the input Dataset. As examples, we can consider *filtering*, *projection*, *sorting* among many others. Actions, instead, are those operations that consume elements of an input Dataset to produce a specific result. As example consider *counting* or *printing*. One of the most important features of the Spark programming API, and so of Datasets, is that transformations are lazily evaluated. More precisely, Spark keeps track of the sequence of transformations (also known as *lineage*) which define each Dataset. As soon as an action is invoked, Spark identifies the sequence of transformations needed for computing the Dataset on which the action has been invoked, and creates one or more *jobs* for carrying out the computation. A *job* is made of *stages* which are collections of *tasks* that can be executed in parallel. Transformations of each job are transformed into a directed acyclic graph (DAG) which defines dependency relations between transformations. Finally, Spark translates the DAG into an optimized execution plan which is executed in order to compute the result of the invoked action. This feature makes Spark applications very efficient, since Dataset are computed only when they are needed, but at the same time also very fault tolerant, since transformations defining each Dataset are stored and so they can be (partially) re-executed if some errors happen.

Example 1. *Let us consider data reported in the following table:*

rood_id	timestamp	measure
room_1	1	10
room_2	1	12
room_1	2	14
room_1	3	18

Assuming that the table is stored into a CSV file named sensors.csv, then the following spark application reads this CSV file and finds all the available measurements for "room_1".

```
SparkSession session = SparkSession.builder().getOrCreate();

Dataset<Row> sensorsData = session.read()
    .schema("room_id string, timestamp integer, measure integer")
    .csv("sensors.csv");

int room1Measures = sensorsData
    .filter(sensorsData.col("room_id").equalTo("room_1"))
    .show();
```

The first instruction builds the so called spark session which serves as entry point of all Spark functionalities. By means of this session object, it is possible to read from different sources. In our case we are reading the data from a source whose data follow a schema made of three columns, namely room_id, timestamp, and measure; and it is represented by a CSV file. The resulting data defines a Spark Dataset named sensorsData.

Then, sensorsData is used to print all the available measurements for room_1. To this end, first the filter transformation is applied for filtering all the rows having room_1 as value of the column "room_id", then then the action show prints the result of our query.

The execution of a Spark application is typically done in parallel. More precisely, there is one driver process and many worker processes. The driver process is responsible for managing the execution of the Spark application and distributing the work across the workers (or executors). The executors carry out the work that is assigned to them by the driver program. When Spark is executed on a cluster, executors are scattered across the network and so the executors may be spawned on different physical machines. However, Spark provides also a local mode, where executor processes are spawned on the same machine but they use different CPU cores.

2.2. Datalog

Terms can be either *variables* or *constants*. Constants are strings starting with lowercase letter or integers, while variables are strings stating with uppercase letter. An *atom* a is an expression of the form $p(t_1, \ldots, t_n)$ where p is a predicate of arity n and $t_1, \ldots, t_n$ are terms. An atom is *ground* if all its terms are constants. A *literal* is either an atom a or its negation $not\ a$, where not denote negation as failure. A literal of the form a is said to be *positive*, while a literal of the form $not\ a$ is said to be *negative*. Given a literal $l = a$ (resp $not\ a$), $\bar{l}$ denotes the *complement* of l, that is $not\ a$ (resp a). A *rule* is an expression of the form $h \leftarrow l_1, \ldots, l_n$ where h is an atom referred to as *head*, and $l_1, \ldots, l_n$ referred to as *body*, is a conjunction of literals. Given a rule r, B_r denotes the set of literals in the body of rule r, and H_r the atom appearing in the head of r. Given a set of literals S, we denote S^+(resp S^-) the set of positive (resp. negative) literals of S. A *program* Π is a set of rules. The dependency graph of a program Π, G_Π is a directed labeled graph where the nodes are predicates appearing in Π and the set of the edges contains a positive (resp. negative) edge (u, v) if there exists a rule $r \in \Pi$ such that $v \in H_r$ and u appears in B_r^+ (resp. in B_r^-). A program Π is said to be *stratified* if G_Π does not contain loops involving negative edges. A program is said to be *recursive* if G_Π contains some loops.

In this paper we consider stratified programs that contain no loops in G_Π.

Example 2. *Let us consider sensors data from Example 1. In order to compute all the available measurements we can write the following Datalog program:*

$$\begin{aligned} sensor(room\_1, 1, 10) &\leftarrow \\ sensor(room\_2, 1, 12) &\leftarrow \\ sensor(room\_1, 2, 14) &\leftarrow \\ sensor(room\_1, 3, 18) &\leftarrow \\ room1Measures(T, M) &\leftarrow sensor(room\_1, T, M) \end{aligned}$$

Here the facts over the $sensor$ predicate encode the ground truth about sensors measure. Then the rule $room1Measures(T, M) \leftarrow sensor(room\_1, T, M)$ is used to derive all those measures relative to "room_1" which are encoded by atoms over predicate $room1Measures$, that are $room1Measures(1, 10)$, $room1Measures(2, 14)$, and $room1Measures(3, 18)$.

For further details we refer the reader to dedicate literature [5].

3. Compiling Datalog into Spark

To bridge Datalog and Spark worlds we provide a compilation-based approach aimed at compiling Datalog programs into ad-hoc Spark applications which serve as parallel and large-scale evaluator for the compiled programs. To this end, we first propose an extension of the base Datalog language in order to support typed predicates and aggregation function; and then we present our compilation techniques.

3.1. Extended Datalog

Relational databases typically have a well defined schema in which columns of a given relation are associated to a specific data type. However, predicates in Datalog programs do not enforce a fixed schema. The lack of such restriction may introduce unexpected behavior when Datalog predicates are mapped over Spark Datasets which impose a fixed schema. A natural way of filling this gap is to extend the Datalog base language with ad-hoc *type directives*. A type directive is an expression of the form $\#type\ pred\_name(type_1, ..., type_n)$, where $pred\_name$ is a predicate name and $type_1, ..., type_n$ represent terms' data types. More precisely, supported data types are: `numeric`, for representing integers and floats, and `string` for representing strings.

Example 3. *Let us consider the Datalog program from Example 2, and the following type directives:*

$$\begin{aligned} &\#type\ sensor(string, numeric, numeric) \\ &\#type\ room1Measures(numeric, numeric) \end{aligned}$$

By means of these two type directives we are imposing that, for the ground atoms over $sensor$ predicate, terms must be respectively of type string, numeric, and numeric. Similarly for ground atoms over predicate $room1Measures$, both terms must be of numeric type.

Type directives are not mandatory, and so for all those predicates for which a type directive does not exist, terms are either considered as string (for input predicates) or their type is inferred during compilation (for non-input predicates). By exploiting such typing mechanism, users can specify predicates' schemas which must be satisfied during query evaluation.

Another natural language extension is the usage of aggregation functions. Aggregates are widely adopted in declarative formalisms and allow to specify complex query in a very compact and natural way [13]. To this end, we consider standard aggregates defined in the ASPCore-2 as well as novel aggregate functions which are also supported by the Spark programming API. More precisely, a *symbolic set* is a pair of the form $V : Conj$, where V is a list of terms and $Conj$ is a conjunction of literals. A *ground set* is a set of pair $v : conj$ the v is a list of constants and $conj$ is conjunction of ground literals. An *aggregate function* is an expression of the form $f(S)$ where S is either a symbolic or ground set and $f \in$

$\{\#count, \#sum, \#min, \#max, \#avg, \#median, \#stddev, \#var\}$ is an *aggregate function symbol*. Note that, $\#avg$, $\#median$, $\#stddev$, and $\#var$ aggregate function symbols stand for descriptive statistics in their mathematical terms (i.e., average, median, standard deviation and variance respectively). An *aggregate atom* is an expression of the form $f(S) \prec T$, where $f(S)$ is an aggregation function, $\prec \in \{<, \leq, >, \geq, =\}$ is a comparison operator and T is a term that is called guard. An aggregate atom $f(S) \prec T$ is ground if S is a ground set and T is a constant.

In the extended Datalog language we allow the usage of aggregate atoms in rule bodies.

Example 4. *Let us consider the Datalog program from Example 2. The following rule can used to derive the atom q (i.e., our query) if there are at least two measures available for "room_1":*

$$q \leftarrow \#count\{T, M : room1Measures(T, M)\} \geq 2$$

By means of such aggregate atoms it is possible to model more involved query requiring aggregations which are very frequent in Big Data analytics context.

3.2. Compilation

We now describe how extended Datalog programs (i.e. Datalog programs with type directives and aggregates) can be compiled into ad-hoc Spark applications. The idea behind our approach consists of interpreting predicates of a Datalog program Π as Spark datasets and rules of Π as transformations over such datasets. In particular, each predicate in Π is associated with a Spark dataset which has as many columns as the arity of the predicate; whereas each rule is evaluated by means of multiple join transformations between predicates appearing in the body and ad-hoc transformations for aggregate atoms.

To compile rules containing aggregates we first apply a program rewriting technique which normalizes rules containing aggregates into rules where each aggregate atom has exactly one literal in its body.

W.l.o.g., we assume each rule of an extended Datalog program contains at most one aggregate atom. Let r be a rule of the form $h \leftarrow l_1, \ldots, l_n, \#f\{V : Conj\} \succ T$ then it is rewritten as follows:

$$\begin{aligned} body_r(\tau) &\leftarrow l_1, \ldots, l_n \\ agg\_set_r(V, \rho) &\leftarrow body_r(\tau), Conj \\ H &\leftarrow body_r(\tau), \#f\{V : agg\_set(V, \rho)\} \succ T \end{aligned}$$

where τ is the union of all the terms appearing in literals $L_1, \ldots, L_n$, and ρ are those terms appearing both in $Conj$ and τ. The following example should better clarify our rewriting.

Example 5. *Let r be the following rule:*

$$h(X, C) \leftarrow a(X, Y), b(Y, Z), \#count\{W : c(Y, W), d(W, Z)\} = C$$

Then our rewriting techniques transform r in the following rules:

$$\begin{aligned} body_r(X, Y, Z) &\leftarrow a(X, Y), b(Y, Z) \\ agg\_set_r(W, Y, Z) &\leftarrow body_r(X, Y, Z), c(Y, W), d(W, Z) \\ h(X, C) &\leftarrow body_r(X, Y, Z), \#count\{W : agg\_set_r(W, Y, Z)\} = C \end{aligned}$$

Intuitively, the first rule encodes the truth of the literals in the body r without considering the aggregate atoms by means of the predicate $body_r$. Then, the second rule defines the join between literals in the body of r and the conjunction of literals in the aggregate atom by means of the agg_set_r predicate. Finally, the $body_r$ and agg_set_r predicates are used to reconstruct the original rule.

After rewriting all rules containing aggregates, the rewritten program is translated into a Spark application. The resulting application serves as a solver for evaluating arbitrary instances of the compiled program.

In order to describe the proposed compilation techniques, we proceed with the following example of extended Datalog program that we refer to as Sensors:

Example 6.

$$
\begin{aligned}
meanR(R,T,Mean) &\leftarrow t(T), r(R), \#avg\{M : measure(R,T,M)\} = Mean.\\
int(T1,T2,B) &\leftarrow t(T2), t(T1), intSize(D), D = T2 - T1, atLeastCrit(B).\\
alertH(T1,T2,R) &\leftarrow int(T1,T2,B), r(R), comfort(R,Min,Max),\\
&\quad \#count\{T : T > T1, T <= T2, meanR(R,T,M), M > Max\} >= B.\\
trash(B) &\leftarrow alertH(T1,T2,R), stored(B,R,S,E), T1 >= S, T2 <= E.
\end{aligned}
$$

In this example, program instances are indeed sequences of measures made by sensors that are placed in controlled environments called rooms, encoded by predicates $measure/3$ and $room/1$, and batches of products stored into rooms for a given amount of time, encoded by predicate $stored/4$. Then, we aim at identifying abnormal room conditions according to aggregated sensors' measures. More precisely, a room r experiences an abnormal condition over an interval I if the average measure of the sensors in the room falls outside the comfort range (specified for r) for at least b time points of the interval I. Finally, we determine which batches of products were stocked in a room under abnormal conditions. Such products should be trashed. A batch B of products stored in a room R from time $T1$ to time $T2$ should be trashed if the room R experienced some abnormal condition within time range $T1 - T2$.

Let us consider the rule

$$meanR(R,T,Mean) \leftarrow t(T), r(R), \#avg\{M : measure(R,T,M)\} = Mean.$$

This rule is rewritten as follows:

$$
\begin{aligned}
tr(T,R) &\leftarrow t(T), r(R).\\
meanR(R,T,Mean) &\leftarrow tr(T,R), \#avg\{M : measure(R,T,M)\} = Mean.
\end{aligned}
$$

Note that, in this case, the rewriting does not generate the rule defining agg_set_r since the conjunction inside the aggregate atom already contains exactly one literal.

These two rules are then compiled into Spark transformations. The rule $tr(T,R) \leftarrow t(T), r(R)$ basically generates the cartesian product between time points and rooms. Thus, it can be compiled as follows: first of all, the compiler prints the instructions for loading predicates t and r from CSV files into two Spark Datasets; then the compiler prints the instruction that computes the cartesian product between the two datasets and saves the result of this transformation in a new Dataset tr. The following code snippet reports the code generated by the *Compiler* for the rule $tr(T,R) \leftarrow t(T), r(R)$.

```
Dataset<Row> t = session.read()
    .format("csv")
    .schema("t0 float")
    .load(instance_path + "t.csv")
    .dropDuplicates();
Dataset<Row> r = session.read()
    .format("csv")
    .schema("t0 float")
    .load(instance_path + "r.csv")
    .dropDuplicates();
Dataset<Row> tr = t.crossJoin(r);
```

At this point the Dataset tr can be used in the evaluation of the next rule. The rule $meanR(R,T,Mean) \leftarrow tr(T,R), \#avg\{M : measure(R,T,M)\} = Mean$ computes the average of measures for each time point T and each room R. For this rule, the compiler first prints the instruction that loads the predicate $measure$ into a new Dataset and then prints the instruction for computing the join between the Datasets tr and $measure$. More precisely, such join is defined on the equalities between (i) the first term of tr and the second term of $measure$ and (ii) the second term of tr and the first term of $measure$. The result of the join defines a new Dataset, namely *trmeasure*. Then, the compiler prints the transformations for aggregating measures relative to each time point and

each room. More precisely, the *trmeasure* Dataset is first grouped by columns representing time and room and then the measures of each group are aggregated using the average (avg) function. Finally, the resulting Dataset is used to model the $meanR$ predicate which appear in the head of the rule. The following snippet reports the compiled code describe above.

```
Dataset<Row> measure = session.read()
    .format("csv")
    .schema("t0 float, t1 float, t2 float")
    .load(instance_path + "measure.csv")
    .dropDuplicates();
Dataset<Row> trmeasure = tr.alias("tr")
    .join(
        measure.alias("m"),
        col("tr.t0").equalTo("m.t1")
        .and(col("tr.t1").equalTo("m.t0"))
    )
    .select(
        col("tr.t1").as("t0"),
        col("tr.t0").as("t1"),
        col("m.t2")
    );
Dataset<Row> meanR = trmeasure.groupBy(col("t0"),col("t1"))
    .agg(avg("t2").as("t2"));
```

All the remaining rules are first rewritten and then compiled by following the same idea described so far. Thus, we avoid reporting the lengthy code would be generated for all the remaining rules.

3.3. Implementation

We implemented the Extended Datalog language and the compilation-based approach described above in a novel system named *Datalog2Spark*, which is able to compile Extended Datalog programs into Spark applications. The architecture of *Datalog2Spark* is illustrated in Figure 1.

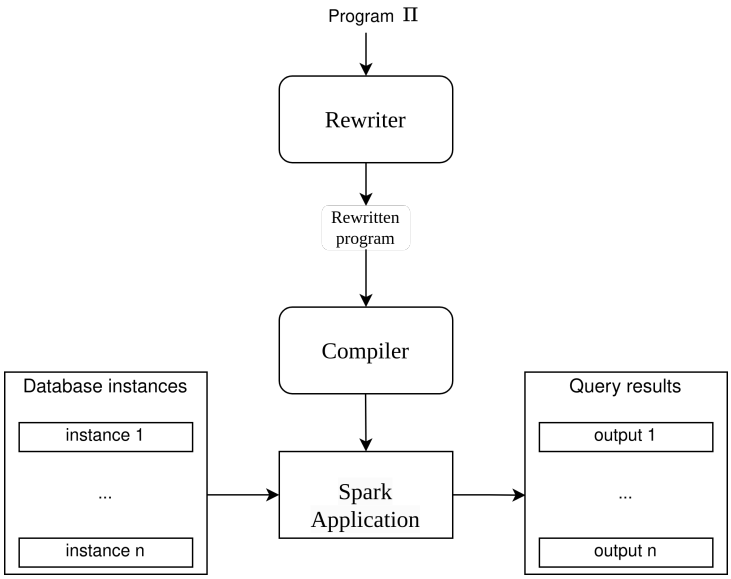

Figure 1: System architecture

Datalog2Spark takes as input an Extended Datalog program, say Π, which is then analyzed and

rewritten by the *Rewriter* module. More precisely, the module reads all type directives which will be instrumental in defining the schema of the resulting Spark Datasets. For all those input predicates for which the user did not specify a type directive, the *Rewriter* module defines a type directive in which all terms are assigned to the type string. Once input types have been defined, the *Rewriter* module proceeds by inferring the types of remaining predicates and by checking that there are no type mismatches. Whenever a mismatch occurs, the compilation aborts. The following example should clarify how both type inference works and how mismatches are detected.

Example 7. *Consider the following program* Π*:*

$$q(D, Tot) \quad \leftarrow \quad m(D), \#sum\{Q, M : p(M, D, Q)\} = Tot.$$

And the following type directive:

$$\#type\ p(string, string, numeric)$$

Input predicates of Π *are* m *and* p*. However, there is one type directive for the predicate* p *and so, the Rewriter module introduces a default directive for* m *of the form* $\#type\ m(string)$*.*

At this point, terms of all input predicates have a precise type and so, the Rewriter proceeds by checking that such types do not lead to a mismatch.

More precisely, for the predicate q*, the Rewriter infers the type directive* $\#type\ q(string, numeric)$*. Intuitively, the first term of* q *inherits its type from the first term of* m *that is* $string$*, while the second term of* q *inherits its type from the result of the sum aggregation function, which is* $numeric$*.*

Since no type directives have been inferred/specified for q *then no conflicts arise.*

Conversely, a type directive of the form $\#type\ q(string, string)$ *would have led to a mismatch on the type of the second term of* q*.*

Then, the *Rewriter* module applies the rewriting technique described above by replacing each rule containing aggregates with the ones produced by the rewriting. As a result, the *Rewriter* generates a program, namely *Rewritten Program* denoted by Π', which is then given to the *Compiler* module for compiling each rule into ad-hoc Spark transformations. More precisely, the compiler first computes the dependency graph of Π'. Then, it computes strongly connected components $C_1, ..., C_n$ that give us a topological order of $G_{\Pi'}$ such that no paths exist from C_j to C_i if $i < j$. By following that order, for each component C (which is indeed a set of predicates appearing in Π'), the compiler compiles all the rules having in the head predicates appearing in C.

Such an order guarantees that each rule $r \in \Pi'$ is evaluated only when all the predicates appearing in the body of r have been previously evaluated.

As a result, we obtain an ad-hoc Spark application which can be used to evaluate arbitrary instances of the program Π. More precisely, an instance of Π must be a folder made of different predicate-specific CSV files, which represent input facts.

4. Experiments

To assess performances of the proposed approach we conducted an empirical evaluation over computation-intensive benchmarks. In our comparison we considered the CLINGO and DLV systems and four versions of *Datalog2Spark* with the following research goals: (**G1**) assessing the impact of *Datalog2Spark* w.r.t. CLINGO and DLV; (**G2**) studying the scalability of *Datalog2Spark* with an increasing number of Spark workers. The four different versions of *Datalog2Spark* are reported as SPARK-N where N represents the number of workers used by Spark.

Benchmarks In our evaluation we consider both benchmarks taken from the literature [14] and a benchmark built on top of the Sensors program reported in the previous section. More precisely, the

first three benchmarks are formulations of data-intensive join operations, while the last benchmark is useful to assess how *Datalog2Spark* performs over queries involving aggregates.

Data intensive joins fall in the large join category considered by [14]. Each of the joins presents different features. DBLP is a well-known Computer Science bibliography database. The DBLP problem extracts articles data by computing a 4-way join between the single input relation att, and selects almost all the columns of the join.

$$q(Id, T, A, Y, M) \leftarrow att(Id, title, T), att(Id, year, Y), att(Id, author, A), att(Id, month, M).$$

Instances generated for DBLP include a number of articles between 200000 to 6000000. Each article could include up to 6 or up to 9 authors, while both month and year of publication are taken at random. The Join1 problem defines four binary joins between predicates of arity two.

$$\begin{aligned} a(X,Y) &\leftarrow b1(X,Z), b2(Z,Y). \\ b1(X,Y) &\leftarrow c1(X,Z), c2(Z,Y). \\ b2(X,Y) &\leftarrow c3(X,Z), c4(Z,Y). \\ c1(X,Y) &\leftarrow d1(X,Z), d2(Z,Y). \end{aligned}$$

Instances for Join1 were generated by varying the domain size of the input predicates (namely d1, d2, c3, c4, c2) and the number of facts for each predicate. We varied the domain of each term of the input predicates between 300 and 1000, and we considered either all the tuples in the cartesian product of the domains or a subset of it (ranging from 40% to 100% of the tuples).

The LUBM problem defines three independent queries over a more complex database. In this case, multiple joins between distinct relations are computed and then, except for query1, all columns of the resulting join are selected.

$$\begin{aligned} query1(X) &\leftarrow takesCourse(X, graduateCourse0), graduateStudent(X). \\ query2(X,Y,Z) &\leftarrow graduateStudent(X), memberOf(X,Z), undergraduateDegreeFrom(X,Y), \\ &\quad university(Y), department(Z), subOrganizationOf\_0(Z,Y). \\ query9(X,Y,Z) &\leftarrow advisor(X,Y), teacherOf(Y,Z), takesCourse(X,Z), \\ &\quad student(X), faculty(Y), course(Z). \end{aligned}$$

Instances for LUBM are generated by varying many parameters like the number of students, the number of universities, the number of courses followed by each student, and a few others. For generating instances that are a plausible representation of real-world universities, we decided to fix a parameter configuration and then to multiply by a factor all the parameters, except the number of courses attended by student. In this way we generated instances ranging from 50000 students split into 12 universities, up to 800000 students split in 225 universities. For each fixed parameters configuration we made the number of courses followed by student vary from 5 to 20.

The last benchmark, instead, is an extension of the sensors program from Example 6. In particular, we added rules encoding further abnormal room conditions detected by measures that are below the minimum comfort range value. More precisely, the following rules are added:

$$\begin{aligned} alertL(T1,T2,R) &\leftarrow int(T1,T2,B), r(R), comfort(R,Min,Max), \\ &\quad \#count\{T : T > T1, T <= T2, meanR(R,T,M), M < Min\} >= B. \\ trash(B) &\leftarrow alertL(T1,T2,R), stored(B,R,S,E), T1 >= S, T2 <= E. \end{aligned}$$

Instances for sensors are generated by varying the total number of rooms, the number of sensors per room, and the number of time points for which a measure exists. We varied the number of rooms from 100 to 600, the number of time points from 2000 to 4000 and the number of sensors from 4 to 6.

Hardware setup All experiments were executed on Intel(R) Xeon(R) CPU E7-8880 v4 @ 2.20GHz, running Debian GNU/Linux 12; with memory and CPU (i.e. user+system) limited to 64GB and 1800s. Executables and benchmarks are available at https://osf.io/jc4wn/?view_only=88e6cf62edec4678b3417217cb9b1252

Results. Obtained results are reported in the plots in Figures 2-5. We recall that in the cactus plots instances are sorted by time, and a point (i, j) of the plot indicates that a system solved the i-th instance with time limit of j seconds. Each line in a cactus plot reports execution times for a given system.

For assessing the impact of the proposed approach w.r.t. the CLINGO and DLV systems (**G1**) we compare them with SPARK-1 which uses only one worker for a fair comparison.

Overall, it is possible to observe that considered benchmarks highlight both strength and limitation of the proposed approach. For DBLP, SPARK-1 achieves better results w.r.t. CLINGO and DLV since it pre-filters the predicate *att* over the specific columns and then computes the 4-way join between the

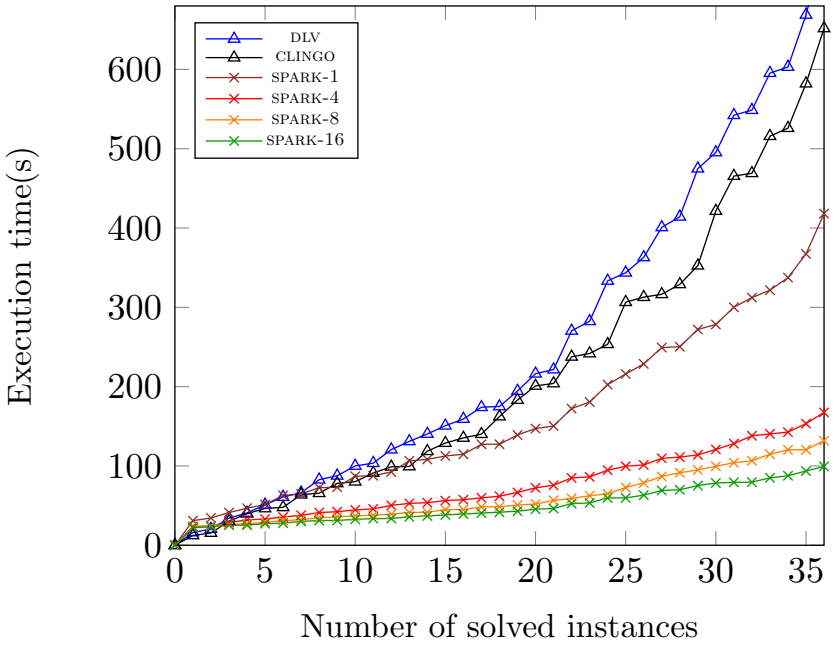

Figure 2: DBLP

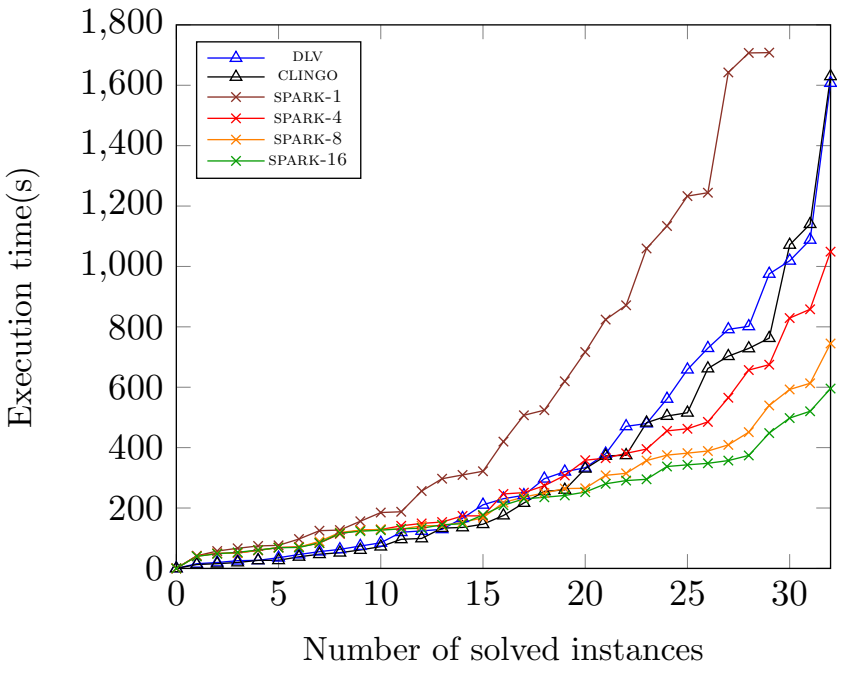

Figure 3: Join1

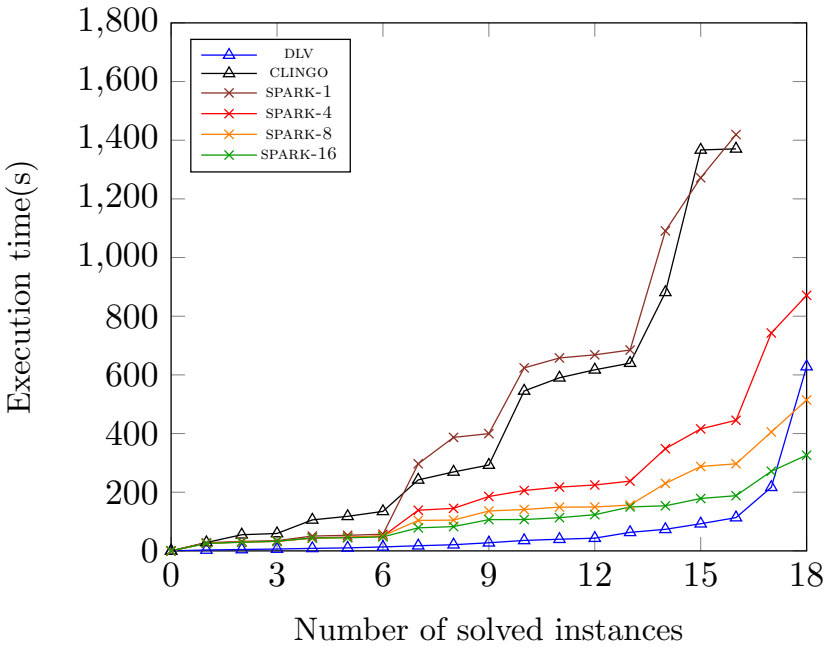

Figure 4: LUBM

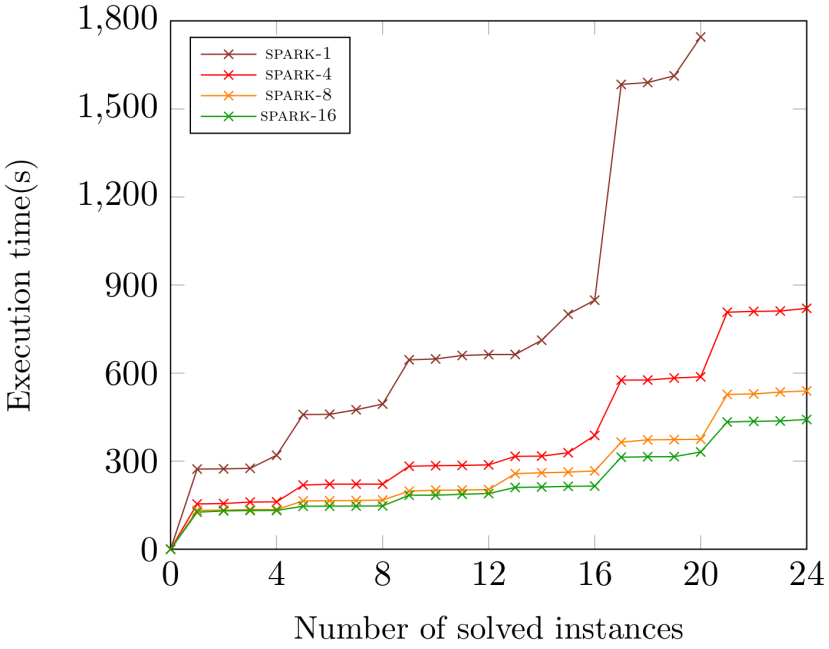

Figure 5: Sensors

filtered Datasets, which is way more efficient than iterating the entire predicate set. On the contrary, for JOIN1 benchmark, SPARK-1 exhibited some overhead w.r.t. CLINGO and DLV. Such an overhead is mainly due to the large number of duplicates produced by the join operation. In this case SPARK-1 is not able to identify such duplicates and so it first generates duplicated tuples and then drops them, which is detrimental as soon as the number of duplicates start increasing.

For the LUBM benchmark, SPARK-1 exhibited performance comparable to that of CLINGO, while DLV emerged as the best-performing system. In this case, SPARK-1 applies limited pre-filtering, restricted to the *takesCourse* predicate. Unlike the *dblp* benchmark, this filtering has little effect on overall performance; however, SPARK-1 introduces no significant overhead compared to CLINGO. By contrast,

Table 1
Spark versions' speedup

Benchmarks	SPARK-1		SPARK-4		SPARK-8		SPARK-16	
	Total	speedup	Total	speedup	Total	speedup	Total	Speedup
DBLP	5985	-	2789	2.15	2214	2.70	1817	3.29
JOIN1	27390	-	10202	2.68	8249	3.32	7383	3.47
LUBM	14956	-	4408	3.39	2905	5.15	2103	7.11
SENSORS	29602	-	9578	3.09	6665	4.44	5727	5.17

DLV benefits from advanced rule ordering strategies, allowing it to scale efficiently on smaller instances. Nonetheless, as the instance size grows, DLV does not scale as well as SPARK. However, it is important to point out that, even though the use of only one worker is not the typical Spark deployment mode, SPARK-1 revealed to be competitive w.r.t. CLINGO and DLV. Finally, for the sensors encoding there is no line for CLINGO and DLV since they do not support the aggregate $\#avg$ used in this benchmark.

We now study the scalability of *Datalog2Spark* by considering an increasing number of Spark workers (**G2**). To this end, we compared performance of SPARK-1, SPARK-4, SPARK-8, and SPARK-16.

Table 1, reports the cumulative PAR-2 score for all the spark versions, and the speedup of each version with respect to SPARK-1 for each benchmark. We recall that the PAR-2 score is obtained by considering the total runtime for completed instance and 2 times the timeout for timed out instances. Whereas, speedups are then computed by dividing the total execution time of SPARK-1 by the total execution time of the each SPARK-N.

In almost all the considered benchmarks, it is possible to observe a significant improvement introduced by SPARK-4 w.r.t. SPARK-1. This suggests that partitioning datasets over different Spark workers is very effective even if the obtained speedup is not linear. Probably this is due to the size of the considered instances, and so by considering larger instances it is expected that the speedup observed for SPARK-8 and SPARK-16 increases. Among considered benchmarks, LUBM and SENSORS are the one on which Sparks scales the most. More precisely, for SENSORS benchmark SPARK-1 experiences four timed out instances while SPARK-16 allows to solve all the instances roughly within 600s (i.e. a third of the considered timeout). This is very positive result which highlights also the effectiveness of the proposed approach in modeling typical Big Data analysis requiring aggregation.

5. Related Work

Datalog has long served as a core declarative framework for querying and reasoning over data [4, 5]. Its evaluation is typically handled by main-memory systems [14, 15, 10, 9], which struggle to scale in the realm of Big Data applications. This limitation has led to a growing body of work focused on scalable engines which aim at parallelizing the evaluation of logic programs both on single machine and in distributed environments [16, 17, 6, 7]. Tachmazidis et al. introduced a MapReduce-based approach for evaluating logic programs under well-founded semantics, achieving scalability over large datasets. However, their approach is tailored for specific example of programs, which limits its applicability, and more importantly, it may not benefit from the advancements introduced by modern Big Data technologies such as in-memory computation supported by Spark [3]. In this direction, Shkapsky et al. and Rogala et al. proposed approaches for integrating Datalog queries within Spark applications. Both systems allow the user to integrate Datalog queries, expressed a new query language inspired to Datalog, into complex Spark pipelines. However, both approaches force the user to explicitly embed the query within a Spark application. In contrast, our compilation-based approach can automatically compose Spark applications which serve as ad-hoc distributed solvers for Datalog programs.

Related techniques are also those realized for the parallel instantiation of ASP program of which Datalog evaluation is a subcase [19]. Compilation-based techniques have recently gained popularity and proved to be highly effective for evaluation of both ASP and Datalog programs [20, 21, 22, 23]. Notably, Cuteri and Ricca introduced a technique to compile Datalog into ad-hoc C++ engine. However,

resulting engines still remain limited to sequential main-memory execution which represents a strong limitation for large-scale programs.

6. Conclusions

In this paper, we presented *Datalog2Spark*, a preliminary system that explores the compilation of Datalog queries into Spark programs for scalable evaluation over modern distributed data management platforms. Our prototype focuses on non-recursive Datalog with stratified negation, extended with type definitions and powerful aggregation functions, addressing key requirements for effective Big Data analytics. A preliminary experimental evaluation against main-memory engines, DLV and CLINGO, demonstrated that *Datalog2Spark* delivers acceptable performance on single-server environments while offering seamless scalability across distributed platforms. Additionally, our system supports expressive aggregation features that are not supported by available engines.

These encouraging results suggest that *Datalog2Spark* can be a promising foundation for a scalable Datalog implementation, and future work will focus on extending the system to support recursive queries, optimizing compilation strategies, and further assessments of performance on large-scale distributed environments.

Acknowledgments

This work was supported by the Italian Ministry of Industrial Development (MISE) under project EI-TWIN n. F/310168/05/X56 CUP B29J24000680005; and by the Italian Ministry of Research (MUR) under projects: PNRR FAIR - Spoke 9 - WP 9.1 CUP H23C22000860006, Tech4You CUP H23C22000370006, and PRIN PINPOINT CUP H23C22000280006.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] G. Harrison, S. Feuerstein, MySQL stored procedure programming - building high-performance web applications with PHP, Perl, Python, Java and .NET: covers MySQL 5, O'Reilly, 2006.

[2] K. Li, H. Jiang, L. T. Yang, A. Cuzzocrea (Eds.), Big Data - Algorithms, Analytics, and Applications, Chapman and Hall/CRC, 2015. URL: https://doi.org/10.1201/b18050. doi:10.1201/B18050.

[3] Apache Software Foundation, Apache spark, 2025. URL: https://spark.apache.org/.

[4] H. Garcia-Molina, J. D. Ullman, J. Widom, Database systems - the complete book (2. ed.), Pearson Education, 2009.

[5] S. Ceri, G. Gottlob, L. Tanca, Logic Programming and Databases, Surveys in computer science, Springer, 1990. URL: https://www.worldcat.org/oclc/20595273.

[6] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, C. Zaniolo, Big data analytics with datalog queries on spark, in: F. Özcan, G. Koutrika, S. Madden (Eds.), Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016, ACM, 2016, pp. 1135–1149. URL: https://doi.org/10.1145/2882903.2915229. doi:10.1145/2882903.2915229.

[7] A. Shkapsky, M. Yang, C. Zaniolo, Optimizing recursive queries with monotonic aggregates in deals, in: J. Gehrke, W. Lehner, K. Shim, S. K. Cha, G. M. Lohman (Eds.), 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015, IEEE Computer Society, 2015, pp. 867–878.

[8] F. Ricca, N. Leone, Disjunctive logic programming with types and objects: The dlv$^+$ system, J. Appl. Log. 5 (2007) 545–573.
[9] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV system for knowledge representation and reasoning, ACM Trans. Comput. Log. 7 (2006) 499–562.
[10] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, P. Wanko, Theory solving made easy with clingo 5, volume 52 of *OASICS*, Schloss Dagstuhl, 2016, pp. 2:1–2:15.
[11] B. Chambers, M. Zaharia, Spark: The definitive guide: Big data processing made simple, " O'Reilly Media, Inc.", 2018.
[12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, 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, USENIX Association, USA, 2012, p. 2.
[13] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, Asp-core-2 input language format, Theory Pract. Log. Program. 20 (2020) 294–309.
[14] S. Liang, P. Fodor, H. Wan, M. Kifer, Openrulebench: an analysis of the performance of rule engines, in: J. Quemada, G. León, Y. S. Maarek, W. Nejdl (Eds.), Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009, ACM, 2009, pp. 601–610. URL: https://doi.org/10.1145/1526709.1526790. doi:10.1145/1526709.1526790.
[15] H. Jordan, B. Scholz, P. Subotic, Soufflé: On synthesis of program analyzers, in: CAV (2), volume 9780 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 422–430.
[16] A. Dovier, A. Formisano, G. Gupta, M. V. Hermenegildo, E. Pontelli, R. Rocha, Parallel logic programming: A sequel, Theory Pract. Log. Program. 22 (2022) 905–973.
[17] I. Tachmazidis, G. Antoniou, W. Faber, Efficient computation of the well-founded semantics over big data, Theory Pract. Log. Program. 14 (2014) 445–459.
[18] M. Rogala, J. Hidders, J. Sroka, Datalogra: datalog with recursive aggregation in the spark RDD model, in: P. A. Boncz, J. L. Larriba-Pey (Eds.), Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016, ACM, 2016, p. 3. URL: https://doi.org/10.1145/2960414.2960417. doi:10.1145/2960414.2960417.
[19] S. Perri, F. Ricca, M. Sirianni, Parallel instantiation of ASP programs: techniques and experiments, Theory Pract. Log. Program. 13 (2013) 253–278.
[20] B. Cuteri, F. Ricca, A compiler for stratified datalog programs: preliminary results, in: S. Flesca, S. Greco, E. Masciari, D. Saccà (Eds.), Proceedings of the 25th Italian Symposium on Advanced Database Systems, Squillace Lido (Catanzaro), Italy, June 25-29, 2017, volume 2037 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2017, p. 158. URL: https://ceur-ws.org/Vol-2037/paper_23.pdf.
[21] G. Mazzotta, F. Ricca, C. Dodaro, Compilation of aggregates in ASP systems, in: AAAI, AAAI Press, 2022, pp. 5834–5841.
[22] C. Dodaro, G. Mazzotta, F. Ricca, Compilation of tight ASP programs, in: ECAI, volume 372 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2023, pp. 557–564.
[23] C. Dodaro, G. Mazzotta, F. Ricca, Blending grounding and compilation for efficient ASP solving, in: KR, 2024.