The VLDB Journal (2021) 30:859–881
https://doi.org/10.1007/s00778-021-00665-6

REGULAR PAPER

Formal semantics and high performance in declarative machine learning using Datalog

Jin Wang[1] · Jiacheng Wu[2] · Mingda Li[1] · Jiaqi Gu[1] · Ariyam Das[1] · Carlo Zaniolo[1]

Received: 26 May 2020 / Revised: 26 February 2021 / Accepted: 23 March 2021 / Published online: 31 May 2021

Abstract
With an escalating arms race to adopt machine learning (ML) in diverse application domains, there is an urgent need to support declarative machine learning over distributed data platforms. Toward this goal, a new framework is needed where users can specify ML tasks in a manner where programming is decoupled from the underlying algorithmic and system concerns. In this paper, we argue that declarative abstractions based on Datalog are natural fits for machine learning and propose a purely declarative ML framework with a Datalog query interface. We show that using aggregates in recursive Datalog programs entails a concise expression of ML applications, while providing a strictly declarative formal semantics. This is achieved by introducing simple conditions under which the semantics of recursive programs is guaranteed to be equivalent to that of aggregate-stratified ones. We further provide specialized compilation and planning techniques for semi-naive fixpoint computation in the presence of aggregates and optimization strategies that are effective on diverse recursive programs and distributed data platforms. To test and demonstrate these research advances, we have developed a powerful and user-friendly system on top of Apache Spark. Extensive evaluations on large-scale datasets illustrate that this approach will achieve promising performance gains while improving both programming flexibility and ease of development and deployment for ML applications.

Keywords Datalog · Declarative machine learning · Apache spark · Scalability

1 Introduction

The past decades have witnessed a booming demand for large scale data analysis in diverse application domains, such as online advertisement, news recommendation, driverless cars and voice-controlled devices. As machine learning (ML) has achieved widespread success for many data-driven analytical tasks, demand for scaling ML algorithms to ever larger datasets became inevitable. Recently, researchers from both academia and industry have devoted great efforts to building powerful distributed data processing platforms, such as Hadoop and Apache Spark, which utilize and extend the Map-Reduce computation model. The availability of such platforms provides a great opportunity for scaling up ML applications due to their natural in-memory support of advanced big-data applications. Many scalable ML libraries based on different high-level programming languages have been provided by these platforms. A number of remarkable projects underscore the significant progress of systems and applications in this area, including MLlib [1], Mahout [2] and MADlib [3]. Although these systems and libraries ease the burden of implementing ML applications, they still impose strict requirements on developers. Specifically, it often takes considerable efforts to develop new or customize existing ML algorithms, since developers must manage details of the distributed implementations of ML algorithms over the

✉ Carlo Zaniolo
zaniolo@cs.ucla.edu

Jin Wang
jinwang@cs.ucla.edu

Jiacheng Wu
wu-jc18@mails.tsinghua.edu.cn

Mingda Li
limingda@cs.ucla.edu

Jiaqi Gu
gujiaqi@cs.ucla.edu

Ariyam Das
ariyam@cs.ucla.edu

[1] University of California, Los Angeles, USA

[2] Tsinghua University, Beijing, China

underlying platforms, without having full control on how and when the data is accessed.

In order to make better use of the computing resources and simplify the development and deployment, a declarative ML framework is needed where programming can be decoupled from the underlying algorithmic and system concerns. In other words, a framework is needed that allows users to focus on the data flow instead of low-level interfaces. We believe that *Datalog* is a particularly attractive choice for expressing ML algorithms because its natural support for reasoning and recursion simplifies ML applications. Recently, a renaissance of interest has focused on Datalog because of its succinct and declarative expression of a wide spectrum of data-intensive applications, including data mining [4], knowledge reasoning [5], data center management [6] and social networks [7]. A common trend in the new generation of Datalog applications is the usage of aggregates in recursion, since they enable the concise expression and efficient support of much more powerful programs than those expressible by ones that are stratified w.r.t. negation and aggregates. Recent theoretical advances [8–10] allow us to provide formal declarative semantics to the powerful recursive queries that use aggregates in recursion. These findings outline the promising blueprints of a declarative ML framework using Datalog.

In this paper, we propose a declarative framework for efficiently expressing a broad range of ML applications. Unlike the previous studies that rely on user-defined functions (UDF) [11] and those that employ a hybrid imperative and declarative framework [12–14], our framework uses purely declarative programs which only uses the basic logic-based constructs of Datalog. The success of a framework critically depends on the ability of the underlying engine to turn declarative queries and programs into efficient and scalable executions. To this end, we implement our ML framework as an extension of BigDatalog [15], which is a shared-nothing Datalog engine built on top of Apache Spark, to take advantage of its power in dealing with iterative computation on massive datasets. Compared with simpler recursive applications, ML applications require recursions involving more complex structures, e.g., mutual and nonlinear recursion, and multiple aggregates. This calls for optimized semi-naive fixpoint computation techniques not tackled in previous studies. To address these issues, we propose a series of compilation and planning techniques to support these powerful Datalog programs. Moreover, we further provide a number of novel optimizations to improve the overall performance for ML workloads. Note that our proposed techniques are platform-independent and cannot limited to the BigDatalog system.

The effectiveness of Datalog in expressing ML applications is due to the great expressive power achieved by allowing the use of aggregates satisfying particular conditions in recursions. This basic idea was first proposed in [8,9], and proved quite effective at expressing a rich set of graph and data mining algorithms [10,16]. The formal semantics of such queries lies in the fact that programs satisfying the Pre-Mappability (PreM) property [9] can be transformed into equivalent aggregate-stratified programs. Unfortunately, while the notions in [9] work well for the `min` and `max` constraints used in simple recursive queries, they proved insufficient to deal with the classical ML applications which, along with extrema, also make extensive usage of other aggregates, such as `sum`, `count` and `average`. In this paper, we find that ML applications tend to apply aggregates over sets of relations whose cardinality could be pre-computed ahead of time, whereby the computation of all kinds of aggregates becomes monotonic. Following this route, we provide formal semantics for ML applications expressed in Datalog from fixpoint computation.

As a result of these advances, this paper makes the following contributions:

- We devise a declarative ML framework with Datalog query interface. We implement our system on top of Apache Spark and, to enhance its usability, we provide DataFrame APIs that are similar to, and actually more general than, those of Apache MLlib.
- We propose a series of compilation and planning techniques to enable the efficient expression and execution of ML applications (Sect. 5). We further develop several optimizations for the recursive plans of ML workloads (Sect. 6).
- We provide the formal semantics of Datalog programs for ML applications (Sect. 4).
- We evaluate our framework on several popular benchmarks. Experimental results show that our framework outperforms, by an obvious margin, existing ML libraries on Spark and other special-purpose ML systems as well.

The rest of the paper is organized as follows: Sect. 2 reviews the basics about Datalog language and machine learning. Section 3 discusses how ML applications can be expressed in Datalog and the advantages of this approach. Section 4 provides the formal semantics of the above ML queries. Section 5 presents the system implementation and proposes necessary techniques to support complicated Datalog programs for ML applications. Section 6 further presents several optimizations ranging from planning to execution. Section 7 discusses important issues such as usability and generality. Section 8 reports and discusses the experimental results. Section 9 surveys the related work. Finally, Sect. 10 concludes the whole paper.

2 Preliminary

2.1 Datalog and its evaluation

A Datalog program P consists of a finite set of rules operating on sets of facts described by database-like schemas. A rule r has the form $h \leftarrow r_1, r_2, ..., r_n$, where h is the head of rule, $r_1, r_2, ..., r_n$ is the body and the comma separating atoms in the body is logical conjunction (AND). The rule head h and each r_i are atoms having form $p(t_1, t_2, ..., t_k)$, where p is a predicate and $t_1, t_2, ..., t_k$ are terms which can be variables or constants. On occasions, we use the terms predicate, table and relation interchangeably. A rule defines a logical implication: if all predicates in the body are true, then so is the head h. There are two kinds of relations: (i) the base relations are defined by tables in the *EDB* (extensional database) and (ii) the derived relations are defined by the heads of rules and form the *IDB* (intentional database).

Example 1 *Query 1* - Transitive Closure (TC)

$\mathsf{r}_{1,1} : \mathtt{tc(X, Y)} \leftarrow \mathtt{arc(X, Y)}$
$\mathsf{r}_{1,2} : \mathtt{tc(X, Y)} \leftarrow \mathtt{tc(X, Z)}, \mathtt{arc(Z, Y)}$

An example of recursive Datalog program is shown above in the Transitive Closure program in Query 1. Next, we will illustrate some Datalog concepts and terminology with the help of it.

Query 1 derives the IDB relation *tc* from the EDB table arc representing the edges of a graph. Since the predicate *tc* is contained in both the head and the body of rule $\mathsf{r}_{1,2}$, *tc* is a recursive predicate and $\mathsf{r}_{1,2}$ is a *recursive rule*. The recursive predicate *tc* is also the head predicate for $\mathsf{r}_{1,1}$ which is non-recursive and therefore provides the *base rule* in the fixpoint definition and computation of the recursive predicate. In fact the process of query evaluation first initializes *tc* using $\mathsf{r}_{1,1}$ and then uses $\mathsf{r}_{1,2}$ to recursively produce new *tc* facts from the conjunction of *tc* facts generated in previous iterations and the arc relation. Since at most one recursive relation is included in the body of any rule, Query 1 represents a case of *linear recursion*; the term *non-linear recursion* denotes instead the case where some rules contain multiple recursive relations.

The state-of-the-art method for evaluating a Datalog program is the *semi-naive* (SN) evaluation [17]. SN performs the differential fixpoint computation of Datalog programs in a bottom-up manner. It starts with the application of the base rule and then iteratively applies recursive delta rules until a *fixpoint* is reached. The core idea of the SN optimization is that, instead of using the original rules, the evaluation can use delta rules that are based on the facts which were generated in the previous iteration step.

Algorithm 1: Semi-naive Evaluation of Query 1

```
1  begin
2      δtc = arc(X, Y);
3      tc = δtc;
4      do
5          δtc' = Π_{X,Y}(δ tc(X, Z) ⋈ arc(Z, Y)) − tc;
6          tc = tc ∪ δtc';
7          δtc = δtc'
8      while δtc ≠ ∅ ;
9      return tc;
10 end
```

For example, consider how the Transitive Closure program of Query 1 is evaluated by Algorithm 1. The semi-naive evaluation starts by applying the base rule $\mathsf{r}_{1,1}$ (line: 2) and then iterates over the recursive rule $\mathsf{r}_{1,2}$ (line: 4-8) until fixpoint is reached. We use tc and tc' to denote the set of facts in the recursive relation *tc* at the beginning and end of the current iteration, respectively. Then, the set of facts generated in the current iteration could be calculated as $\delta\mathsf{tc} = \mathsf{tc}' - \mathsf{tc}$ (line: 5). And the contents of tc and tc' are updated for the next iteration of evaluation (line: 6–7). During the evaluation of $\mathsf{r}_{1,2}$ in the next iteration, instead of using the whole relation $\mathsf{tc}(X, Z)$, SN just joins $\delta\mathsf{tc}(X, Z)$ with $\mathsf{arc}(Z, Y)$. The termination condition of Datalog evaluation is defined by its *fixpoint semantics*. In this example, the fixpoint is reached when $\delta\mathsf{tc} = \emptyset$ (line: 8). SN has been widely applied in evaluating recursive Datalog programs and simple SN extensions for recursive queries with aggregates have been proposed for the single-node [18], multi-core [19] and distributed [15] environments.

2.2 Basics of machine learning

Generally speaking, the ML problem can be formalized as follows: Given a training set $\mathcal{D}$ with n instances, each instance consists of a d-dimensional feature vector X_i ($i \in [1, n]$) with the jth dimension as x_{ij} and a numeric target y_i. For the regression problems, we have $y_i \in R$, while for classification problems, we have $y_i \in \{-1, 1\}$. The process of discovering the model can be formalized as an optimization problem using the given $\mathcal{D}$. We are given a function $f(\theta; X)$ that makes prediction with a given model θ on the unseen data. The objective is to find a set of parameters θ^* that minimizes the loss function L on f, i.e., $\theta^* = argmin_\theta L(f(\theta; X), Y)$. This can be achieved with the family of first-order gradient optimization methods, namely gradient descent (GD).

There are different ways to compute the gradient depending on the portion of training instances that is used to update the model at each iteration, namely batch gradient descent

(BGD), stochastic gradient descent (SGD) and mini-batch gradient descent (MGD). As is shown in the practice of Google's SQML project [20], BGD is widely adopted in modern ML on relational engines. In this paper, we start our discussion from BGD, which computes the gradients by performing a complete pass on the training data at each iteration. BGD starts from an initial model θ^0 and iterates with Eq. (1) by the increasing number of iterations k until convergence is reached.

$$\theta^{k+1} = \theta^k - \sum_{(X,y)\in\mathcal{D}} \nabla L(f(\theta^k; X), y) + \Omega(\theta^k) \quad (1)$$

where L is the loss function, ∇ is the gradient function based on L and Ω is the regularization.

2.3 Terminology for recursive queries

To describe the recursive queries expressed in Datalog, we introduce some necessary terminologies from [17] and [21].

The *monotonicity* property for the rules defining a recursive predicate ensures that the fixpoint procedure previously described produces a unique result that is the least fixpoint of the mapping defined by the rules.

Rules that do not use negation or aggregates are monotonic: these rules can be implemented using union, select, projection, Cartesian product, natural join, i.e., the monotonic constructs of relational algebra. However, rules using negation are non-monotonic and cannot be used in recursive queries. Rules using aggregates are only monotonic is some special cases, such as those discussed in Sect. 4 where the aggregates are applied to relations that are completely known or can be computed prior to the processing of the recursive rules.

Given a Datalog program P, its *dependency graph* G_P can be constructed as following: Every rule is a vertex, and an edge $\langle r_i, r_j \rangle$ appears in the graph whenever the head of r_i appears in the body of r_j. If non-monotonic constructs are applied before r_i, the node corresponding to it in G_p is a *negated node*. With the help of its dependency graph, the stratification of a Datalog program can be formally stated by Definition 1.

Definition 1 By applying topological sorting over G_p, its node can be partitioned into n strata $S_1, ..., S_n$ with larger i in a lower stratum. The program P is stratified when $\forall$ edges $\langle r_i, r_j \rangle \in G_p$ where $r_i \in S_y$ and $r_j \in S_x$ we have that: (i) $y \geq x$ if r_i corresponds to a non-negated node and, (ii) $y < x$ if r_i corresponds to a negated one.

3 Datalog for machine learning

In this section, we express ML applications with Datalog and provide the formal semantics of such programs. We first describe how to write Datalog queries for ML applications in Sect. 3.1. Then, we further cover the issues of supporting generalized gradient descent and identifying the stop condition in Sects. 3.2 and 3.3, respectively.

3.1 Expressing ML applications

We will next discuss how to express ML applications with Datalog. As data sparsity is ubiquitous in ML applications, many training sets are represented in the verticalized format to save space, such as those in the famous LIBSVM benchmark [22]. For each training instance $X = \langle Id, Y, x_1, \cdots, x_d \rangle$, the verticalization process would produce at most d instances $\langle Id, Y, k, x_i \rangle$ ($k \in [1, d]$) as dimensions with value 0 will be omitted. When writing the Datalog programs, we use a verticalized view *vtrain*(`Id`, `C`, `V`, `Y`) to denote the training set, where `Id` denotes the id of a training instance; `Y` denotes the label; `C` and `V` denote the dimension and the value along that dimension, respectively.

With such a verticalized relation, we can now write the Datalog query to describe the training process with BGD using three recursive relations:

- *model* represents the trained model in verticalized form, where each tuple contains the following three attributes: `J` is the iteration counter; `C` is a dimension in the model; and `P` is the value of parameter in that dimension.
- *gradient* represents the results of gradient computed at each iteration by the three attributes `G`, `C` and `J`: `G` is the gradient value of the `C`th dimension in the `J`th iteration.
- *predict* represents the intermediate prediction results with model in the current iteration for each training instance. Its schema has three attributes: `J` is the iteration counter; `Id` is the id of the training instance; `YP` is the predicted y value for the training instance.

Among these steps, the gradient computation and prediction with the current model can be easily represented with aggregates in recursion. Therefore, the iterative training process can be expressed with a recursive Datalog program Query 2. Firstly, the model is initialized according to some predefined mechanisms in $r_{2,1}$ (Here we use all 0.01 as example). Then, the function f is used to make prediction on all training instances according to the model obtained in the previous iteration in $r_{2,4}$. Next the gradient is computed by the function g (derived according to the loss function L) using the predicted results in $r_{2,3}$. Finally, in $r_{2,2}$ the model is updated w.r.t the gradients (and optional regularization Ω). Here lr denotes the learning rate and n is the number of train-

Table 1 Settings for ML algorithms

Algorithm	Predict function f	Loss function L	Gradient $g = \bigtriangledown_P L$	∂Regularizer Ω
Linear regression	$YP = V * P$	$(YP - Y)^2$	$2 * (YP - Y) * V$	N/A
Logistic regression	$YP = \frac{1}{1+e^{-V*P}}$	$\begin{cases} -\log(YP), & Y = 1 \\ -\log(1 - YP), & Y = 0 \end{cases}$	$(YP - Y) * V$	N/A
SVM	$YP = V * P$	$\max(0, 1 - Y * YP)$	$\begin{cases} -Y * V, & \text{if } Y * YP < 1 \\ 0, & \text{otherwise} \end{cases}$	N/A
L2 regularized SVM	$YP = V * P$	$\max(0, 1 - Y * YP)$	$\begin{cases} -Y * V, & \text{if } Y * YP < 1 \\ 0, & \text{otherwise} \end{cases}$	$\mu * P$
Lasso regression	$YP = V * P$	$(YP - Y)^2$	$2 * (YP - Y) * V$	$\mu * \text{sgn}(P)$
Ridge regression	$YP = V * P$	$(YP - Y)^2$	$2 * (YP - Y) * V$	$\mu * P$

For SVM, we append an extra 1/-1 for each instance to save the bias parameter; μ is a hyper-parameter which controls the weight of regularization term. Meanwhile, we use a *sign function* to deal with the derivative near 0 of L1 regularization in Lasso regression

ing instances. And the training process moves on to the next iteration (Increase J by 1).

Query 2 - Batch Gradient Descent (BGD)

```
r2,1 :          model(0, C, 0.01) ← vtrain(_, C, _, _).
r2,2 :           model(J1, C, NP) ← model(J, C, P),
                                    gradient(J, C, G),
                                    NP = P − lr ∗ (G/n + (P)),
                                    J1 = J + 1.
r2,3 : gradient(J, C, sum⟨G0⟩) ← vtrain(Id, C, V, Y),
                                    predict(J, Id, YP),
                                    G0 = g(YP, Y, V).
r2,4 : predict(J, Id, sum⟨Y0⟩) ← vtrain(Id, C, V, _),
                                    model(J, C, P),
                                    Y0 = f(V, P).
```

The advantage of Query 2 lies in its generality: by varying the set of functions (f, g, Ω), it can support a wide spectrum of ML algorithms [1], whereby an incomplete list of ML applications that can be expressed by Query 2 is shown in Table 1. Besides, the Mini-batch Gradient Descent (MGD) can also be expressed with Datalog queries with minor changes on Query 2 (details in Sect. 3.2).

The output of Query 2 is the trained model. Other necessary steps in machine learning, i.e., validation and test, can be easily implemented in a similar way. Take the evaluation on a test set as example: this can be accomplished by joining a verticalized test set *vtest* with the table *model* using a process that is similar to Query 2. Furthermore, Query 2 can be easily extended to memorize the evaluation result of each training instance in a table, which can be used to calculate other metrics such as AUC, precision, recall and accuracy. To support validation sets, a verticalized *vvalidate* table can be created to compute the loss after updating the model with $r_{2,2}$ in each iteration.

[1] In this paper, we limit our discussion to the linear models and leave the issue of deep learning models as future work.

We further show a concrete example of training the Linear Regression model with Batch Gradient Descent as Query 3. We will use this as the running example to demonstrate our proposed techniques in the following sections.

Query 3 - BGD for Linear Regression

```
r3,1 :          model(0, C, 0.01) ← vtrain(_, C, _, _).
r3,2 :           model(J1, C, NP) ← model(J, C, P),
                                    gradient(J, C, G),
                                    NP = P − lr ∗ G/n,
                                    J1 = J + 1.
r3,3 : gradient(J, C, sum⟨Id, G0⟩) ← vtrain(Id, C, V, Y),
                                    predict(J, Id, YP),
                                    G0 = 2 ∗ (YP − Y) ∗ V.
r3,4 :  predict(J, Id, sum⟨C, Y0⟩) ← vtrain(Id, C, V, _),
                                    model(J, C, P),
                                    Y0 = V ∗ P.
```

To demonstrate the benefits of ML applications written in *Datalog*, we will compare them with Scala programs that perform direct manipulations on RDDs. Figure 1 shows a fragment of a Scala program that expresses the very process of Query 3 by manipulating and directly transforming the RDDs. We can observe from this process that compared with such a Scala program, the *Datalog* program shown in Query 3 is more succinct and simpler to define since it does not require the programmer to: (i) know the details of query evaluation; (ii) specify the physical plan of dataflow and make lower-level optimizations.

3.2 Supporting mini-batch gradient descent

Previously we discussed how BGD can be expressed with Datalog. Here, we further show how to support Mini-batch Gradient Descent (MGD). A major challenge is due to the fact that MGD requires the training data to be randomly shuffled before every iteration, and this can be expensive in a distributed environment. To tackle this issue, we adopt the

```
var data = sc.parallelize(input, numParts)
             .map(d => (d.label, d.feature))
var weights = Vectors.dense(initW.toArray)
var n = weights.size
var converged = false
var i = 1
while (!converged && i <= numIterations) {
  val bcWeights = data.context.broadcast(weights)
  val seqOp = (grad, (label, feature)) => {
    var diff = dot(feature, bcWeights.value) -
        label
    grad += dot(diff, feature)
    grad
  }
  val combOp = (c1, c2) => {c1 += c2}
  val gradientSum = data.treeAggregate(
      DenseVector.zeros(n))(seqOp, combOp)
  weights += dot(stepSize, gradientSum / data.size)
  prevWeights = currWeights
  currWeights = Some(weights)
  converged = isConverged(prevWeights.get,
      currWeights.get, tol=1e-6)
  i += 1
}
weights
```

Fig. 1 Snippet Scala code: BGD for linear regression

trade-off proposed in [11]: instead of making random shuffles before each iteration step, the dataset is optimally shuffled once at the beginning. Then, the training data is split into batches and MGD can be expressed in a way that is similar to BGD.

As described above, we need to randomly shuffle the training data before the query begins. Actually, most parts of MGD are the same as in Query 2; the only difference comes from the way in which the *predict* relation is computed and used to calculate the gradient in the current iteration. To optimize decisions, here we need the hyper-parameters of (i) batch size bs and (ii) cardinality of training set n. The total number of batches in the training set can be calculated as n/bs. We can recognize the batch of training instances that will be involved in each iteration in the following way: Suppose at iteration J, the Bth batch instead of the whole dataset is used for training. Then, given the Id of a training instance, we can identify the batch it belongs to as $Id\ \%\ (n/bs)$. For the Jth iteration, only training instances belonging to the Bth batch, where $B = J\ \%\ (n\ /\ bs)$, should be involved when calculating the table *predict*. Therefore, the computation of Mini-batch Gradient Descent can be realized by replacing $\mathsf{r}_{2,4}$ with the following rule:

$$\begin{aligned}\mathsf{r}_{2,4'}: \mathtt{pred(J, Id, sum\langle Y0\rangle)} \leftarrow\ & \mathtt{vtrain(ID, C, V, \_)},\\ & \mathtt{model(J, C, P)},\\ & \mathtt{Y0 = f(V, P)},\\ & \mathtt{Id\%}(n/bs) == \mathtt{J\%}(n/bs).\end{aligned}$$

3.3 Termination condition

Finally, we discuss the termination condition of Query 2. In recursive Datalog programs, evaluation terminates when the Datalog program reaches a *fixpoint*, producing a unique minimal model. However, this model could be infinite, in which case the fixpoint computation would never terminate. For example, in Query 2 the temporal argument J ranges over an infinite time domain. As J denotes the number of iterations, increasing J by 1 means training for a new iteration. In this case, the delta relation of *model* relation will always be non-empty.

To address this issue, we add conditions that terminate the iterative computation when at least one of the following conditions is satisfied:

- The number of iteration reaches a predefined maximum number $max\,J$.
- The difference between training losses of two adjacent iterations is smaller than a predefined value ϵ.

Popular ML libraries, such as MLlib, enable users to specify hyper-parameters to control termination and limit the number of iteration in a similar manner. In our programs, we can limit the number of iterations by specifying $max\,J$ and adding the condition $J \geq max\,J$ to $\mathsf{r}_{3,2}$ in Query 3, which now becomes:

$$\begin{aligned}\mathsf{r}_{3,2'}: \mathtt{model(J1, C, NP)} \leftarrow\ & \mathtt{model(J, C, P)}, \mathtt{grad(J, C, G)},\\ & \mathtt{NP} = \mathtt{P} - lr * \mathtt{G}/n,\\ & \mathtt{lesser(MaxJ, J + 1, J1)}.\end{aligned}$$

Although IF-THEN-ELSE is a built-in construct in many Datalog systems that could be used to express `lesser`, it cannot be applied here to replace `lesser`. The reason is that the semantics of IF-THEN-ELSE is defined using negation, which would take us back to the depths of the non-monotonic conundrum. As a result, the formal semantics of the program will no longer hold. Therefore, we use the `lesser` predicate defined as follows in these rules:

$$\begin{aligned}\mathtt{lesser(MJ, I, I)} &\leftarrow \mathtt{I} < \mathtt{MJ}.\\ \mathtt{lesser(MJ, I, MJ)} &\leftarrow \mathtt{I} \geq \mathtt{MJ}.\end{aligned}$$

Similar revisions of our rules will also allow us to terminate the SN computation when the difference between training losses in two successive iterations becomes smaller than a predefined value ϵ.

4 Formal semantics

In this section, we define the formal semantics for our recursive queries. We introduce the requirement of formal semantics in Sect. 4.1. Next, we describe the PREM property as a partial solution to this problem in Sect. 4.2. Finally, we extend this solution by introducing the Pre-Computable Cardinality (PCC) property in Sect. 4.3.

4.1 Requirement

We have proposed the use of aggregates in recursion to express important procedures in the training process, such as making prediction, computing gradient and updating model. To guarantee the correctness of these procedures on different systems and execution platforms, we need to provide a rigorous *formal semantics* for such queries. For basic Datalog programs consisting of Horn clauses the *least fixpoint* [21] provides an ideal formal semantics because of its equivalence with the proof-theoretic and model-theoretic semantics of logic, and its amenability to efficient implementation via the semi-naive fixpoint procedure [21]. However, the semantics Datalog programs that uses negation or aggregates in recursion are faced with difficult on-monotonic semantics issues that have been the topic of much previous research [23,24].

Currently, many Datalog systems and the SQL3 standards only allow the use of negation and aggregates in stratified programs (see Definition 1). Stratified programs are easily identify from their PCG, and implemented by a standard procedure called iterated fixpoint which produces the canonical minimal model for the program [21]. However, to express complex algorithms such as those discussed in this paper, we need programs that are not stratified since they use aggregates in recursion. Now, although these programs can be characterized under powerful formal semantics [25], such as stable model semantics, we are still lacking efficient algorithms to compute their canonical minimal model(s) (more than one can exist for each program) and deciding whether stable models exist for a given program is also difficult. Fortunately, recent research has identified two classes of programs which combine formal semantics with efficient computation of their canonical minimal models and apply to our algorithms. These are discussed next.

4.2 The PREM property

The Pre-Mappability(PREM) property [9] provides formal conditions for pushing extrema aggregates, i.e., `max` and `min`, into recursion while preserving the semantics of the original stratified program. As shown in Definition 2, its definition is based on viewing a Datalog program as a mapping $T(R)$ where T is a relational algebra expression, and R is the vector of relations used in the expression.

Definition 2 (PREM) Given a function $T(R_1, \ldots R_k)$ defined by relational algebra and a constraint γ, γ is said to be Pre-Mappable to T if the following property holds:

$$\gamma(T(R_1, \ldots, R_k)) = \gamma(T(\gamma(R_1), \ldots, \gamma(R_k))).$$

For instance, if T denotes the union operator, and γ denotes the `min` or `max` constraint, we can pre-map (i.e., push) γ to the relations taking part in the union.

In fact, if extrema in recursive programs satisfy the PREM property, those programs produce the same results as their equivalent aggregate-stratified version, for which they just provide an optimized implementation obtained by "pushing" the `min` and `max` aggregates into recursion. Thus, the SN fixpoint of the program simply provides a more efficient realization of the aggregate-stratified semantics already adopted by Datalog systems and SQL3 standards.

Query 5 - All Pair Shortest Path

```
r5,1 :        spath(X, Y, D) ← arc(X, Y, D).
r5,2 : spath(X, Y, min⟨D⟩) ← spath(X, Z, D1), arc(Z, Y, D2),
                              D = D1 + D2.
```

For example, Query 5 expresses the 'All Pairs Shortest Path' computation which identifies the shortest paths between all pairs of nodes in the graph. In rule $r_{5,1}$, arc denotes the edges in the graph, while D is the distance from node X to node Y. The rule $r_{5,2}$ takes arcs originating in Z and appends them to the previously produced paths terminating at Z, whereby the length of the new arc is $D = D1 + D2$. In this process, it is safe to pre-map the `min` aggregate to D as it only filters out tuples in spath that will produce non-minimal values for D. Consequently, the performance of the query is much more efficient than in the stratified version that only applies the `min` filter at the end of the recursive iterations. More details regarding the ability of PREM to optimize graph queries have been demonstrated in [26,27], where efficient techniques for testing the validity of PREM for the applications at hand were also discussed. Regarding techniques for proving PREM, the interested readers can find more details in [9,28]. However, the PREM property only applies to constraints with `min` and `max` aggregates. This is not the case for `sum`, `count` (when represented in unary as a collection of facts), `average` and other aggregates. To resolve such issues, we need to propose new approaches to deal with them in unstratified programs containing such aggregates.

4.3 Extension to completed aggregates

4.3.1 Motivation and definition

While extrema could be viewed as constraints premappable into recursive queries, allowing `count`, `sum` and `average`

in recursive computations requires a different approach. Thus, we propose an approach that exploits the incremental computation by which these aggregates can be defined in Datalog. For instance, the computation of `average` consists of two phases: in the first phase, monotonic rules are used to compute a pair $\langle num, total \rangle$ by increasing the *num* by 1 (as in continuous count) and adding to the current sum the new value (as in continuous sum). This monotonic phase completes when all elements in the set have been processed, In the second phase, the maximum value of *num* and the value of *total* associated with it are extracted and the ratio of the latter over the former is returned as the answer. Thus, the decision that the first phase is completed enables us conclude that the current count is the max value of *num*, and this represents the quintessential non-monotonic decision taken in the implementation of such aggregates. But when the cardinality of the set involved is known or can be pre-computed before we enter into the recursive computation, this information could simply be passed to the fixpoint computation that follows and used to set the value of *num* whereby no non-monotonic decision will be taken. Moreover, whether we actually pre-compute *num* or let it be derived as the final step in the recursive computation the results are the same and they can be computed efficiently by a semi-naive fixpoint. Thus, the `average` aggregate expressed using monotonic constructs can be used freely in recursion. Moreover, to compute the `sum` we can still compute the pair $\langle num, total \rangle$ in order to achieve monotonicity, but then only return the value of *total* as the result. Remarkably, this *Pre-Countable Cardinality* (PCC) condition occurs in many programs of great practical significance of Datalog [29]. We will now formally provide the PCC idea in Definition 3.

Definition 3 (PCC) Let R be a recursive relation in Datalog, and let δR_i denote the delta values of R obtained at each iteration i during the SN fixpoint computation. Then, R satisfies the PCC condition when:

(i) The cardinality of δR_i is nonzero and is the same for each i;
(ii) The cardinality of δR_i can be known ahead of time before the SN fixpoint computation begins and stays unchanged.

4.3.2 Semantics provided by PCC

As previously described, the non-monotonic aggregate `sum` can be computed by incrementally computing the pair $\langle num, total \rangle$ and returning the *total* value associated with the *num* value *that is equal to the cardinality pre-computable before the recursive computation*. In this way, the computation process will involve only monotonic constructs, since the incremental computation of continuous count and sum is monotonic. In other words, the program with `sum` aggregates in recursion is equivalent to stratified programs where the cardinality is pre-computed at a lower stratum, which precedes the SN computation of the equivalent program that only use monotonic constructs[2]. Observing that similar properties also hold for other aggregates, we can summarize our finding in Theorem 1, which is a summary of the high level idea of [29].

Theorem 1 *If the* PCC *property is satisfied by a recursive Datalog program P that uses* `sum`*,* `avg` *and* `count` *in recursion, then there exists an equivalent aggregate-stratified program which defines its formal semantics.*

4.3.3 Semantics of programs for ML

We can thus show that the semi-naive fixpoint computation of Query 2 indeed realizes the formal semantics defined above. In fact, the first `J` in Query 2 coincides with the successive steps of the semi-naive fixpoint, and the cardinality of arguments of the `sum` aggregate remains the same at each step, and can in fact be pre-computed before the recursive computation starts. Here the value *n* is the cardinality of training set, i.e., *vtrain*. In the process of recursive computation, a step of the semi-naive computation terminates after processing exactly the same number *n* of input values for each value of `J`. Thus, the SN computation for Query 2 realizes the formal fixpoint semantics of the equivalent stratified where the cardinality is pre-computed before the semi-naive fixpoint computation begins.

Then, we formally conclude these findings with the following Theorem 2. We can use the similar techniques proposed in [26] for testing PREM to enable automatically testing of the PCC property.

Theorem 2 *The results produced by Query 2 are equivalent to the same results produced with a query that is stratified with respect to the* `sum` *aggregate.*

5 Query evaluation

In this section, we introduce the query evaluation and optimization techniques that enabled the superior performance of our framework. In this paper, we focus on providing a detailed description of their implementation on BigDatalog along with the extensive experiments that prove their effectiveness. However, it is clear the techniques and their promising performance can be generalized to different shared-nothing Datalog systems. We first briefly introduce the background knowledge of BigDatalog system which our framework is built on (Sect. 5.1). Then, we introduce the new techniques

[2] If this program contains min and max, a third stratum is needed on top of these two to defined its formal stratified semantics.

to deal with complex recursions (Sect. 5.2) and query execution (Sect. 5.3).

5.1 The BigDatalog system

BigDatalog [15] is a Datalog language implementation on Apache Spark. It supports relational algebra, aggregation and recursion, as well as a host of declarative optimizations. BigDatalog uses and extends Spark SQL operators, and also introduces several operators implemented in the Catalyst framework so that its planning features can be used on the recursive plans of Datalog programs.

The input processed by the BigDatalog compiler includes the Data Definition Language (DDL) to specify the database schema and the Datalog program for expressing particular applications. The compiler analyzes the input query and creates a logical plan from it. To resolve recursion, the compiler recognizes recursive tables and switches from the task of building the operator tree for non-recursive queries to the specialized task required by recursive queries. Thus, after recognizing the recursive references, the compiler produces the Predicate Connection Graph (PCG) [30] to identify the dependency of relations within the program.

The logical plan maps the PCG to a tree containing standard relational operators and recursion operators. Such *recursion operators* are used in the logical and physical plan to process the recursive query. The plan actually consists of the following two parts: (i) The *base plan* specifies the base case of the recursion which starts the iterations; and (ii) the *recursive plan* defines behaviors within each iteration. In this process, the aggregates and group-by columns are automatically identified for each sub-query.

The physical plan is generated by analyzing the logical plan with the Spark SQL analyzer and applying rules defined in the optimizer. The BigDatalog operators use Spark SQL row type much in the same way in which Spark SQL uses the standard relational operators [15]. In order to support recursion, our system introduces specialized recursion and shuffle operators into the physical plan. The proper settings for shuffle operators are identified by calling on Catalyst optimizer of Spark SQL. Finally, the query plan is executed by the Spark engine using the RDDs and transformation operators such as distinct, union and subtract.

5.2 Supporting complex recursions

5.2.1 Challenges

Compared with the simpler applications now supported by BigDatalog, such as those discussed in [10,15], ML applications require much more complex recursive queries than those discussed in [10,15]. This is illustrated by the dependency graph between the four relations of Query 3 shown in Fig. 2. We can see that the plan involves two kinds of complex recursions:

- *Mutual recursion* occurs when multiple recursive relations rely on each other to compute the result. For example, in rules $r_{3,2}$ through $r_{3,4}$, the recursive relations *model*, *gradient* and *predict* rely on each other and thus create a cycle which denotes a mutual recursion in Fig. 2.
- *Nonlinear recursion* means that there are more than one recursive relation in the body of a rule. For example, rule $r_{3,2}$ involves two recursive relations *model* and *gradient*.

By analyzing the PCG, the compiler recognizes these two kinds of recursion and marks the rules with special tags. These tags identify the particular recursion types and the different techniques used to process them, which are described next.

5.2.2 New recursion operator

To support mutual recursion, we define a special recursion operator named *Mutual Recursion Operator* (MRO). It provides a major extension to the basic Recursion Operator (RO) of BigDatalog that cannot be used for mutual recursion since it only allows one recursive relation in the recursive plan. MRO instead allows mutual references among multiple recursive relations by including them in the recursive plan in a cascading manner. For each set of mutually recursive relations, only one MRO has the base plan, since the base case for other MROs is provided by the operator that precedes them in the plan.

Example 2 The logical plan for Query 3 is shown in Fig. 3. The root of the plan is an MRO with both base and recursive plan. The left child is the base plan with only the vtrain relation representing rule $r_{3,1}$, which provides the base case

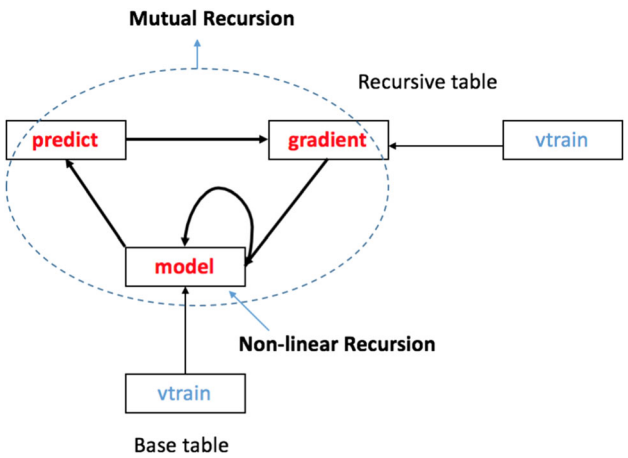

Fig. 2 Dependency between Tables in Query 3

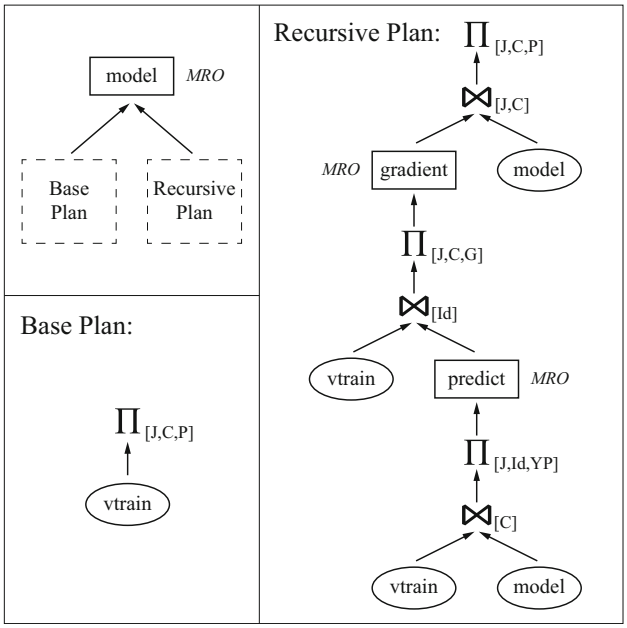

Fig. 3 Logical plan of Query 3

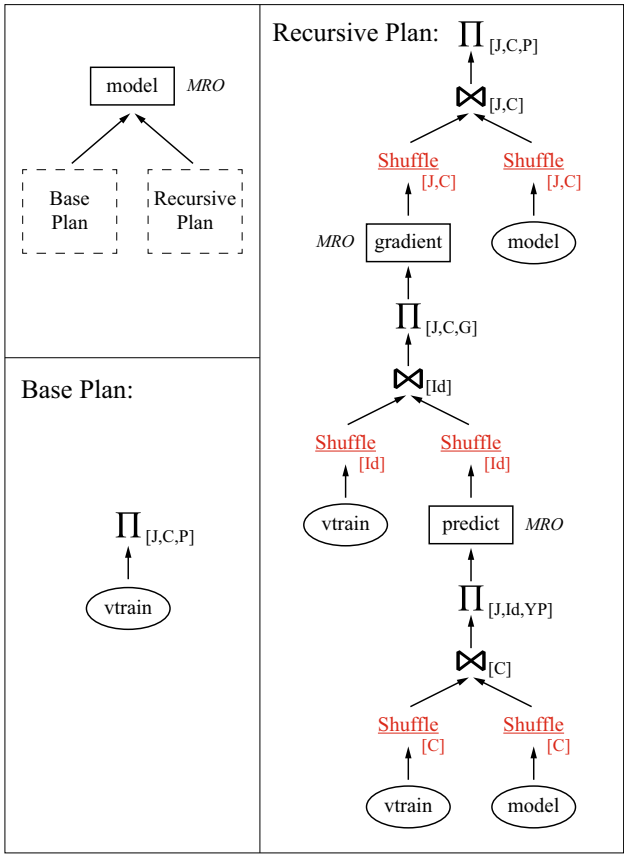

Fig. 4 Physical Plan of Query 3

of the mutual recursion. The right child is the recursive plan representing rules $r_{3,2}$ through $r_{3,4}$. Each MRO represents a rule within the mutual recursion. We can see that all MROs belonging to the recursive plan have a NULL base plan (omitted in Fig. 3).

The corresponding physical plan is shown in Fig. 4. It consists of operators translated from the logical plan along with the shuffle operators and their partitioning information. For example, in the recursive plan, when the join between recursive relations *model* and *gradient* is computed, both operands must be shuffled according to their join keys J and C. The recursive plan in Fig. 4 also shows that this join operation is followed by two more joins, each of which requires two shuffle operations. Therefore, a total of six shuffle operations are performed at each iteration.

5.2.3 Distributed semi-naive evaluation

To evaluate the program in a distributed environment, the physical plan assigns each MRO to the master node where it executes and becomes responsible for driving the distributed query evaluation. The most important step is the scheduling of shuffle operators that are injected between successive steps of the physical plan presiding to the distributed evaluation. The shuffle operators are used to re-partition the dataset in all cases where the output produced by an operator is different from that of the operator using it as input according to the execution plan. Then, the BigDatalog engine utilizes fixpoint computation to drive the iterative evaluation process using the distributed version of semi-naive (DSN) evaluation.

The execution of DSN in the MapReduce framework requires the recursive relations and base relations within one stage to be co-partitioned on a given key K. After that, the execution goes through Map and Reduce stages. Results of the current iteration are generated in the Map stage, while the new delta and the relations needed in the next iteration are generated in the Reduce stage. Algorithm 2 describes the process in more details. The algorithm first defines two auxiliary functions to specify the Map (line 3–5) and Reduce (line 6–10) stages, respectively. In the map stage, the join operation between base and delta relations on the specified join key K is performed on each mapper generating the intermediate results that are allocated to reducers (line 4–5). Here we can also perform selection or projection operations based on the requirement of the Datalog program, which is denoted as F. In the reduce stage, the distributed semi-naive evaluation is then performed. Specifically, each reducer first generates the result D of the current iteration (line 8), which will be emitted later (line 10). Then, the recursive relation R is updated for the next iteration (line 9).

The main process of distributed semi-naive starts at line 11, where the recursive relation is initialized. For each iteration, the algorithm first generates the intermediate results of the map stage (line 14) and then performs shuffle operations

to allocate the results to reducers (line 15). Then, the results of each iteration are computed as the union of results produced by all reducers (line 16). The distributed semi-naive will terminate when the fixpoint is reached (line 17), and the results of R are returned at this point (line 18).

Algorithm 2: DSN Evaluation (B, K)

```
   Input: B: The Base Relation, K: The partition key
   Output: R: All results in the recursive table
1  begin
2      // δR, δR′: Recursive relation (Delta)
3      Map Stage(δR, B)
4      foreach partition pair of (δR, B) do
5          emit F(δR ⋈_{δR.K=B.K} B)
6      Reduce Stage(δR′, R)
7      foreach partition pair of (δR′, R) do
8          D ← δR′ − R
9          R ← δR′ ∪ R
10         emit D
11     δR ← Results of Base Case, R ← ∅
12     repeat
13         i ← i + 1
14         MapOutput ← MapStage(δR, B)
15         δR′ ← ShuffleExchange(MapOutput, key = K)
16         δR ← ReduceStage(δR′, R)
17     until δR == ∅ ;
18     return R;
19 end
```

However, since programs for ML applications include non-linear and mutual recursion, we must revise the evaluation approach described above. For mutual recursion, the solution is relatively easy: One recursive relation is regarded as the driver for DSN, e.g., the *model* relation in Fig. 4, while the others are evaluated by the MROs in the recursive plan. These extensions do not impact the techniques currently used for linear recursion.

A more complex solution is required for non-linear recursion. In fact, let X and Y denote two recursive relations that are involved in a non-linear recursion since they appear as goals in the body of the same rule. Then, the SN evaluation should be performed by enumerating the combination of delta relations as shown in Eq. (2):

$$\begin{aligned}\delta(X \bowtie Y \bowtie B) = &(\delta X \bowtie Y \bowtie B) \cup \\ &(X \bowtie \delta Y \bowtie B) \cup \\ &(\delta X \bowtie \delta Y \bowtie B)\end{aligned} \tag{2}$$

where B is a base relation that is optional in this process. Therefore, unlike the case of linear recursion, we need to keep the whole recursive relations rather than just deltas in order to support non-linear recursion in DSN.

To integrate this optimization into Algorithm 2, the steps described in line 7 –11 of it should be replaced with the operations defined Eq. (2) in order to support non-linear recursion. Similar observations also apply when computing aggregates in recursion.

Example 3 For the example at hand, we can see that non-linear recursion appears in rule $r_{3,2}$ of Query 3 where the *model* relation in the head is obtained by joining *model* and *gradient* on the keys J and C. Then, the delta relation of $r_{3,2}$ should be computed as the union of $model \bowtie \delta gradient$, $\delta model \bowtie gradient$ and $\delta model \bowtie \delta gradient$. Therefore, as shown in Fig. 4, it keeps the whole relation instead of only the delta in our physical plans.

5.3 Execution

To avoid data redundancy in the process of SN evaluation, BigDatalog [15] extended the Resilient Distributed Datasets (RDDs) [31] in Spark and adopted the SetRDD mechanism for executing recursive queries in Spark. SetRDD stores distinct rows of data into a HashSet data structure to optimize the execution of set operators in the DSN. Thus, SetRDD is made mutable under the union operation, which saves system memory by not copying redundant data from up-stream RDDs. However, this optimization may not work when dealing with nonlinear recursion: According to the mechanism of SetRDD, when a recursive relation is referenced in one rule, its corresponding RDD would be modified by the *set union* and *set difference* operations. However, in the case of non-linear recursion, a recursive relation can be referenced more than once within each iteration. Thus, if the recursive relation has been modified by one rule and it is also evaluated by another rule in the same iteration, then its RDD is no longer the same as it was before the first evaluation, whereby the execution results would be incorrect.

To address this issue, we propose a smart strategy to divide the RDDs into *Intra-Iteration* and *Inter-Iteration* ones. Thus, for non-linear recursion, we are able to identify when the RDDs will be re-used in the same iteration. If so, we classify it as *Intra-Iteration* RDD and treat it as immutable, i.e., we generate a new RDD by copying data from the up-stream one. But when an RDD will only be used in the next iteration, we classify it as an *Inter-Iteration* RDD and process it as SetRDD to save memory.

Example 4 Figure 5 shows the series of RDDs generated in the execution step of Query 3. In Query 3 the recursive relation *model* is involved in the nonlinear query and we require to create both Intra-Iteration and Inter-Iteration RDDs for it. Here the green rectangles denote Intra-Iteration RDDs, while the blue dashed ones denote Inter-Iteration ones. We are aware that in the ith iteration, *model* is updated by rule $r_{3,2}$, which would be used in the $i+1th$ iteration. Meanwhile, this table is also used in rule $r_{3,4}$ that updates *predict*. Therefore,

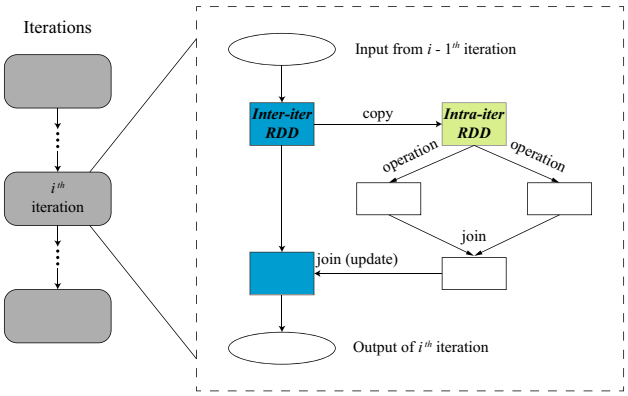

Fig. 5 Intra- versus Inter-Iteration RDDs: To guarantee the correctness of non-linear recursion, we need to create two RDDs for relation *model*: The green one is intra-iteration which is mutable and used in another rule within the same iteration, while the blue one is inter-iteration which will be immutable and used in the next iteration

the RDD of *model* generated by $r_{3,2}$ should be *Inter-Iteration*, while that used in $r_{3,4}$ should be *Intra-Iteration*.

6 Performance optimization

In this section, we present several techniques that have proven to be quite effective in optimizing the performance of our framework.

6.1 Eliminating unnecessary evaluation

For programs with nonlinear recursions, we need to enumerate the combinations of delta relations as shown in Eq. (2) when performing semi-naive evaluations. As a result, the DSN could be significantly more expensive than that with only linear recursions. An example can be observed in Query 3 where the non-linear recursion is used in $r_{3,2}$ when updating the model with the gradient computed in current iteration. The evaluation would require using the whole recursive relations *model* and *gradient* in the physical plan as shown in Fig. 4.

As our investigation progressed from formal semantics to operational semantics, we find that while the textbook techniques for SN optimization of nonlinear queries remain valid, they can be further optimized for specific ML queries. Take again Query 3 as our example: When adopting Eq. (2) to evaluate the query, we need to consider the items in $model \bowtie \delta gradient$, $\delta model \bowtie gradient$ and $\delta model \bowtie \delta gradient$ and thus need to include the full relations *model* and $gradient$. However, note that the join key between *model* and *gradient* is $\langle J, Col \rangle$. In the Jth iteration, since tuples in *model* are from the $J-1th$ iteration, while those in $\delta gradient$ are from the Jth iteration, we have that $model \bowtie \delta gradient = \emptyset$ due to mismatched values of J. Similarly, $\delta model \bowtie gradient = \emptyset$ also holds. Therefore, we only need to evaluate the item $\delta model \bowtie \delta gradient$. As a result, the items *model* and *gradient* can be replaced with $\delta model$ and $\delta gradient$ in the physical plan, which significantly reduces the computational overhead and the network transmission caused by shuffle operations. Since this optimization is based on the execution process of gradient descent, it can be applied for training all linear models with BGD and MGD. Figure 6 shows the physical plan after applying optimizations: the full relations *model* and *gradient* are replaced with delta ones.

6.2 Join optimization with replica

For programs with linear recursion, it is often better to use broadcast join between the delta recursive relation and the base relation in the physical plan by loading the base relation into a lookup table and shared by all workers via broadcasting. Since the overhead of broadcasting can be amortized over the recursion, this approach is rather effective for graph queries where the base table is usually much smaller than the intermediate results [15]. However, the characteristics of ML workloads are totally different from those of graph queries: the size of intermediate results that participates in the computation and must be kept in memory is independent from

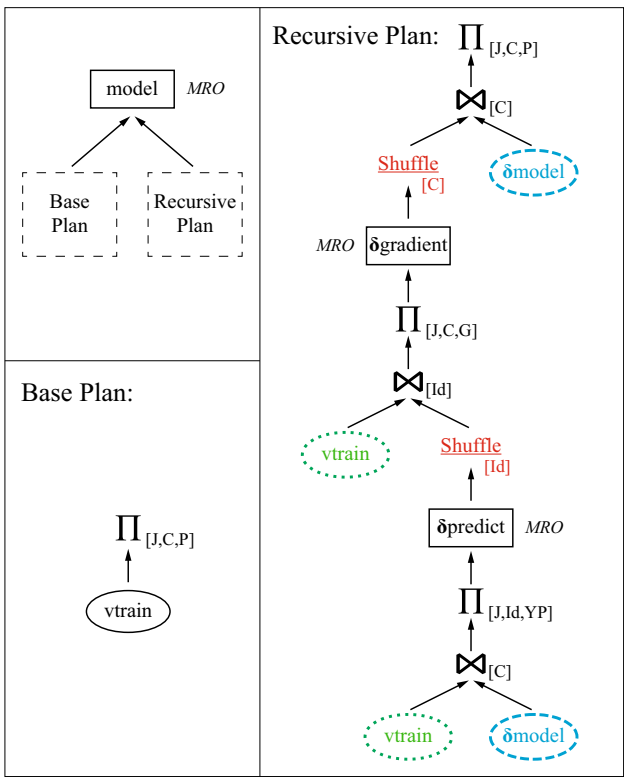

Fig. 6 Optimized physical plan

the number of iterations and is relatively small: the size of *predict* is $2n$ where n is the size of training data; the size of *gradient* and *model* is $2d$ in both cases, where d is the dimension of a training instance [3]. By contrast, the base relation, i.e., the training set, tends to be very large. Moreover, the size of base relation always exceeds the maximum memory of a single worker, making the broadcast join not applicable. As a result, the broadcast joins that proved so effective on graph queries will encounter serious problem on ML workloads. Consequently, there would be multiple shuffle operations per iteration on the base relation, causing significant overhead for the overall performance. As previously observed, shuffle operations on the base relation happen when the base relation is joined with recursive ones on different keys. For example, in $r_{3,3}$ *vtrain* must be joined with *predict* on the key `Id`; and in $r_{3,4}$ the join key between *vtrain* and *model* becomes `C`. Thus, the *vtrain* relation will be shuffled twice.

To address this issue, our framework instead adopts a smart-replica optimization approach that relies on careful trade-offs between memory usage and join performance. We find that the shuffle operations can be avoided by *making replicas of the base relation partitioned by different keys on the same worker*. Specifically, in above example we just make two different replicas of the *vtrain* relation on all workers: one is partitioned by the key *Id* and the other is partitioned by *C* [4]. Then, the former will be used in $r_{3,3}$, while the latter will be used in $r_{3,4}$. The green dotted items in Fig. 6 are relations where the shuffle operations can be avoided by making replicas of *vtrain*. *Here the number of replica, as well the number of shuffle operations it could save, is equivalent to that of the different join keys the base relation gets involved.* As we can see, two shuffle operations could be saved compared with the original physical plan in Fig. 4.

We also want to point out that the space overhead brought by replicas is tolerable. The essence of broadcast join is to trade the memory for join performance. Since the whole base relation is transmitted, the memory overhead on each worker would be the size of the base relation. Meanwhile, the memory overhead of our replica mechanism is the size of base table divided by the number of workers on average. This offers similar benefit as broadcast join does and it avoids its shortcoming of memory consumption. Furthermore, the decision of making replicas can be made automatically: The fact that the base relation need to participate in join operations on different keys can be recognized in the process of formalizing the logical plan. Thus, the usage of replicas will be decided before the actual physical plan is generated. Note that the Spark APIs cannot make such optimizations since the program is directly expressed in terms of physical operations.

6.3 Scheduling optimization

As illustrated in [32], recursive queries that can be compiled into decomposable plans will potentially benefit from a well-chosen partition strategy. In such cases, the produced RDDs preserve the original partition of input recursive table. Then, the executor on the same partition can continue to work without global synchronization. Consequently, the shuffle operations could be saved. The correctness of this property can be guaranteed by the replica mechanism on base relations even if the join key will change for the next operator. The blue dashed items in Fig. 6 are the shuffle operations that can be saved by the scheduling optimizations. For rule $r_{3,2}$, the shuffle operation can be removed since delta of the recursive relation *model* can be acquired locally for each worker. Similarly, in rule $r_{3,4}$, the recursive relation *model* comes from $r_{3,2}$, which has already been partitioned by the same key `C`. Therefore, the shuffle operation on *model* can also be removed.

7 Discussion

In this section, we discuss the usability issues of our proposed framework. Therefore, we will first raise our vantage point by discussing in Sect. 7.1 how to express ML applications with SQL queries that are equivalent to the *Datalog* ones. Then, in Sect. 7.2 we briefly describe how our library of *Datalog* queries for ML has been fully integrated with DataFrame APIs to achieve usability and interoperability with other Apache Spark application libraries. Finally, in Sect. 7.3, we discuss deep neural networks and the many opportunities and challenges that our framework will encounter in such applications.

7.1 Equivalent SQL queries

SQL has delivered great benefits in relational DBMS and big data platforms due to its declarative nature and portability. We show here that SQL can support many ML applications by providing SQL queries that have equivalent semantics to the Datalog ones introduced above. This represents, an important extension to the RaSQL language and its system [26] which supported aggregates in recursion by introducing a simple extension in the syntax of the SQL:2003 SQL standards. Specifically, RaSQL supports basic aggregates, i.e., min, max, sum, count, in recursion by minimal extensions of the Common Table Expressions (CTE) used by current SQL standard with the basic syntax shown below.

[3] The total size of intermediate results would be nJ for *predict* and dJ for *gradient* and *model*. Results from older iterations would be dumped into disk for the sake of crash recovery.

[4] The distribution of replicas partitioned by different keys might be different on the same worker

```
WITH [recursive] VIEW1 (v1_column1,
    v1_column2, ...)
AS (SQL-expression11) UNION
    (SQL-expression12) ...,
[recursive] VIEW2 (v2_column1, v2_column2,
    ...)
AS (SQL-expression21) UNION
    (SQL-expression22) ...
SELECT ... FROM VIEW1 | VIEW2 | ...
```

The WITH RECURSIVE construct of RaSQL

The CTE starts with the keyword "WITH RECURSIVE", which is followed by definitions of the recursive view. The view content is defined by a union of sub-queries, which define the *base table* and *recursive table*. This is similar to the base and recursive relations of Datalog. Here a table is the base table if its FROM clause definition does not refer to any recursive CTE; otherwise it is a recursive table. The RaSQL query that is equivalent with Query 3 is shown in Query 4.

Query 4 - RaSQL: BGD for Linear Regression

```
Base tables: vtrain(Id: int, C: int, V:
    double, Y: double)
WITH recursive model (J, C, P) AS
(SELECT 0, vtrain.C, 0.01 FROM vtrain)
UNION
((SELECT 1+m.J, m.C, m.P+2.0/n*LR*g.G
FROM model AS m, gradient AS g
WHERE m.C = g.C and m.J = g.J),
recursive gradient(J, C, sum() AS G) AS
(SELECT p.J, t.C, (t.Y - p.YP)*t.V
FROM vtrain AS t, predict AS p
WHERE p.Id = t.Id),
recursive predict(J, Id, sum() AS YP) AS
(SELECT m.J, t.Id, m.P*t.V
FROM vtrain AS t, model AS m
WHERE t.C = m.C)),
SELECT * FROM model
```

Such RaSQL queries for ML applications can be compiled into Spark SQL operators and recursive operators in a similar way to that discussed in Sect. 5. Moreover, such RaSQL queries can be encapsulated into a library called by DataFrame operations as MLlib did.

7.2 Usability: supporting DataFrame APIs

To improve usability and attract a wide participation by data scientists, we further encapsulate the *Datalog* queries for ML algorithms in a more elegant and succinct library using DataFrame APIs. Currently, such a library can support all queries introduced in Sect. 3.1. With the help of such a library, users can express the whole process of machine learning using the *Datalog* queries introduced above where the hyper-parameters and data source can be specified in a similar way as MLlib does. Next, we illustrate the basic usage of our API with a running example in Fig. 7.

```
1  val session = DatalogMLlibSession.builder()
2  .appName("LR") .master("local[*]")
3  .getOrCreate()
4  // Import data.
5  var Vschema =
6  StructType(List(StructField("Id", IntegerType, true),
7  StructField("C", IntegerType, true),
8  StructField("V", DoubleType, true),
9  StructField("Y", IntegerType, true)))
10 var df = spark.read.format("csv")
11 .option("header", "false").schema(Vschema)
12 .load("dataDTrain")
13 // Training on the input relation df.
14 import edu.ucla.cs.wis.bigdatalog.spark.DatalogMLlib.
15 {DL_LogisticRegression,
       DL_LogisticRregressionTransformer}
16 val lr = new DL_LogisticRegression().setMaxIter(10)
17 val lrModel = lr.fit(df, session)
18 // Testing with pre-trained model.
19 var test = spark.read.format("csv")
20 .option("header", "false").schema(Vschema)
21 .load("dataDTest")
22 val lrPredict = new
       DL_LogisticRregressionTransformer()
23 val prediction = lrPredict.transform(lrModel, test,
       session)
```

Fig. 7 Snippet code for DataFrame API: logistic regression

The example in Fig. 7 expresses the process of training a Logistic Regression classifier on the training data dataDTrain, and making prediction on the test data, dataDTest. The two datasets are stored in a verticalized view with Vschema (Id, C, V, Y) as introduced in Sect. 3.1. To make use of the *Datalog* programs for machine learning, we first construct a working environment, i.e., DatalogMLlibSession for our library of machine learning algorithm (line: 1–3). Then, we load the training data to a Dataframe df. After importing the required training and predicting functions for Logistic Regression (line: 14–15), we can build executable objects for training lr (line 16) and predicting lrPredict (line: 22). The lr object wraps all the logical rules and required relations (e.g., parameters with default value 0) of the *Datalog* implementation for Logistic Regression. When initializing lr, users can exploit the built-in functions to set the hyper-parameters that control the maximum number of iterations, the method used for parameter initialization, and many others. After fitting the model to df, the lrPredict object could make predictions on the testing instances with the pre-trained model, lrModel. In both the fitting and predicting processes, the information of *Datalog* execution runtime can be obtained by using session as an input argument, which is same as the practice of MLlib.

For the sake of comparison, we also show how Apache Spark MLlib will be used to implement the above example. The snippet code is shown in Fig. 8. The pipeline of functionalities is very similar to that of our APIs; this will make it much easier using the DataFrame APIs in our library for those who are already familiar with MLlib. Although there

are minor differences in the aspects of data formatting and usage of some public functions, e.g., transform and assembler, the expression of MLlib and our library are very similar and both user-friendly.

7.3 Expressing deep learning applications

In this section, we show that it is also possible to express deep neural network models with Datalog and briefly discuss the opportunity to support them with our framework. First of all, unlike linear models which are vectors, the parameters to be learned in deep neural networks are usually matrices, which can be expressed as that in Sect. 3.1. Given a matrix M with m rows and n columns, each element can be represented by the row and column it belongs to and its value. Then, the matrix can be expressed with a set of quadruples $\langle Id, M, N, V, Y\rangle$, where M and N are the row and column number, respectively. There will be no more than $m * n$ such quadruples as we just store the nonzero elements.

Query 5 - Feed Forward Neural Network (BGD)

$$
\begin{aligned}
\mathsf{r}_{5,1}: &\quad \mathtt{model(0,0,C,HC,0.01)} \leftarrow \mathtt{vtrain(\_,C,\_)},\\
&\qquad\qquad\qquad\qquad\quad \mathtt{hidden(HC)}.\\
\mathsf{r}_{5,2}: &\quad \mathtt{model(0,1,HC,1,0.01)} \leftarrow \mathtt{hidden(HC)}.\\
\mathsf{r}_{5,3}: &\ \mathtt{pred(J,0,Id,HC,sum\langle HV\rangle)} \leftarrow \mathtt{vtrain(Id,C,V)},\\
&\qquad\qquad\qquad\qquad\quad \mathtt{model(J,0,C,HC,W)},\\
&\qquad\qquad\qquad\qquad\quad \mathtt{HV = V * W}.\\
\mathsf{r}_{5,4}: &\quad \mathtt{pred(J,1,Id,1,sum\langle Y'\rangle)} \leftarrow \mathtt{pred(J,0,Id,HC,HV)},\\
&\qquad\qquad\qquad\qquad\quad \mathtt{model(J,1,HC,1,W)},\\
&\qquad\qquad\qquad\qquad\quad \mathtt{Y' = \mathbb{E}(HV) * W}.\\
\mathsf{r}_{5,5}: &\quad \mathtt{error(J,1,Id,HC,1,}_1\mathtt{)} \leftarrow \mathtt{ylabel(Id,Y)},\\
&\qquad\qquad\qquad\qquad\quad \mathtt{pred(J,1,Id,1,Y')},\\
&\qquad\qquad\qquad\qquad\quad {}_1 = 2 * (\mathtt{Y'} - \mathtt{Y}).\\
\mathsf{r}_{5,6}: &\quad \mathtt{error(J,0,Id,C,HC,}_0\mathtt{)} \leftarrow \mathtt{error(J,1,Id,HC,1,}_1\mathtt{)}\\
&\qquad\qquad\qquad\qquad\quad \mathtt{model(J,1,HC,1,W)},\\
&\qquad\qquad\qquad\qquad\quad \mathtt{pred(J,0,Id,HC,HV)},\\
&\qquad\qquad\qquad\qquad\quad {}_0 = \mathtt{W} * {}_1 * \mathbb{E}'(\mathtt{HV}).\\
\mathsf{r}_{5,7}: &\quad \mathtt{grad(J,1,HC,1,sum\langle G_1\rangle)} \leftarrow \mathtt{error(J,1,Id,HC,1,}_1\mathtt{)}\\
&\qquad\qquad\qquad\qquad\quad \mathtt{pred(J,0,Id,HC,HV)},\\
&\qquad\qquad\qquad\qquad\quad \mathtt{G_1} = {}_1 * \mathbb{E}(\mathtt{NV}).\\
\mathsf{r}_{5,8}: &\quad \mathtt{grad(J,0,C,HC,sum\langle G_0\rangle)} \leftarrow \mathtt{error(J,0,Id,C,HC,}_0\mathtt{)}\\
&\qquad\qquad\qquad\qquad\quad \mathtt{vtrain(Id,C,V)},\\
&\qquad\qquad\qquad\qquad\quad \mathtt{G_0} = {}_0 * \mathtt{V}.\\
\mathsf{r}_{5,9}: &\quad \mathtt{model(J1,L,C_i,C_o,W)} \leftarrow \mathtt{model(J,L,C_i,C_o,W')},\\
&\qquad\qquad\qquad\qquad\quad \mathtt{grad(J,L,C_i,C_o,G)},\\
&\qquad\qquad\qquad\qquad\quad \mathtt{W} = \mathtt{W'} - lr * (\mathtt{G}/n),\\
&\qquad\qquad\qquad\qquad\quad \mathtt{J1} = \mathtt{J} + 1.
\end{aligned}
$$

Then, Query 5 shows how to express the training process of a feed-forward neural network with BGD. For simplicity of presentation, we just display the training process for a two-layer neural network, with Mean Squared Deviation as the loss function and ignore the regularization items. We use ϕ to denote the activation function and ϕ' to denote its derivative. For instance, if $\phi(z) = \tanh(z)$, then $\phi'(z) = 1 - \phi^2(z)$. Furthermore, to simplify the query, we separate the datasets

```
1  val session = SparkSession.builder().appName("LR")
2  .master("local[*]").getOrCreate()
3  // Import data.
4  var schema = StructType(List(StructField("X1",
       IntegerType, true), StructField("X2", IntegerType
       , true),
5  StructField("X3", DoubleType, true),
6  StructField("label", IntegerType, true)))
7  var df = spark.read.format("csv").option("header", "
       false").schema(schema).load("dataSTrain")
8  // Training on the input relation df.
9  import org.apache.spark.ml.Pipeline
10 import org.apache.spark.ml. classification .
       LogisticRegression
11 import org.apache.spark.ml.feature.VectorAssembler
12 val assembler = new VectorAssembler()
13 .setInputCols(Array("X1", "X2", "X3"))
14 .setOutputCol("features")
15 val lr = new LogisticRegression().setMaxIter(10)
16 val pipeline = new Pipeline().setStages(Array(
       assembler, lr))
17 val lrModel = pipeline. fit (df)
18 // Testing with pre-trained model.
19 var test = spark.read.format("csv").option("header",
       "false").schema(schema).load("dataSTest")
20 val prediction = lrModel.transform(lrModel, test)
```

Fig. 8 Snippet code: implementation with MLlib

into two relations, i.e., vtrain(Id, C, V) (the input data) and ylable(Id, Y) (the corresponding labels). The relation hidden(HC) just contains numbers from 1 to the number of features in the hidden layer, used to initialize the relation model.

The relation model(J, L, C_i, C_o, W) stores all parameters for neural networks, where each tuple denotes the value of weight W associated with the connection between unit C_i in layer L, and unit C_o in layer L+1. I n Query 5, we first initialize the model in $\mathsf{r}_{5,1}$ and $\mathsf{r}_{5,2}$. To train our network, we need to do forward propagation in $\mathsf{r}_{5,3}$ and $\mathsf{r}_{5,4}$. The activation on HV of hidden layer is moved to $\mathsf{r}_{5,4}$ considering the sum semantics. Next, for each record and each feature in layer 0 and 1, we would like to compute an error term in $\mathsf{r}_{5,5}$ and $\mathsf{r}_{5,6}$ that measures the errors each feature was responsible for in the output generated by back propagation. Then, we compute the desired gradients which are just the partial derivatives of the model parameter and summarize the gradients contributed by different records Id in $\mathsf{r}_{5,7}$ and $\mathsf{r}_{5,8}$. Finally, we average the summed gradient, update it on the model and move to the next iteration in $\mathsf{r}_{5,9}$. To support neural networks with more layers, we can simply extend the query by incorporating more rules to calculate different layers of pred, error and grad.

Actually as discussed above in Sect. 7.2, our framework is developed in contrast with Apache Spark's inherited machine learning library MLlib, which also does not aim at supporting deep learning applications. From the above example, we conclude that it is possible to express deep learning applications with Datalog. Therefore, exploring how to efficiently

support deep learning applications expressed by Datalog programs represents an interesting direction for future research.

8 Experiments

8.1 Experimental setup

8.1.1 Workloads and datasets

We evaluate the performance of our framework on the task of training linear models via gradient descent optimizers. As is stated before, we mainly focus on BGD. But we also report the results of MGD using the method described in Sect. 3.2. Specifically, we use Linear Regression, Logistic Regression and SVM as benchmark models in this paper.

The datasets used in the experiments are summarized in Table 2, where cardinality means the number of training instances, while "# Features" means the number of dimensions in each training instance. We conduct experiments on 4 public datasets provided by LIBSVM [22], a popular benchmark for evaluating linear models: URL [33] is a dataset for identifying malicious URLs. KDD10 comes from Carnegie Learning and DataShop that was used in KDD Cup 2010. KDD12 [34] is a CTR prediction task from KDD Cup 2012. WEBSPAM [35] is a dataset of email spams. Currently, we are focusing on training linear models to learn from sparse datasets, which occur frequently in real-life applications, and indeed all the above-selected datasets are from real world scenarios. Results on dense datasets are presented later in Sect. 8.6. Considering the memory available at each node and in the overall system, the cardinality of these datasets provide a good basis for evaluation. Besides the dataset, the memory must hold the intermediate results and system runtime, and the same is true for baseline systems used in our comparisons.

8.1.2 Baselines and metrics

As BigDatalog is implemented on top of Apache Spark, we mainly compare it against two Spark-based competitors: MLlib 2.3.0 and SystemML 1.2.0, where MLlib [1] is the official Spark package for machine learning [5]. As MLlib comes with an implementation with MGD, we implement BGD by setting the batch size as the cardinality of the training set. SystemML [36] is a state-of-the-art ML system on top of Spark using a declarative R-like language [6]. We implement the training process with BGD and MGD using its script language following the official documentation. We are also aware that there are several special-purposed machine learning systems, including TensorFlow, PyTorch, MXNet and Petuum. Due to the space limitation, we just select PyTorch [7] as the representative for comparison. Other studies published on Datalog for machine learning [37] and [14] do not provide a good basis for comparison. This is because simple query interfaces rather than end-to-end systems are provided in [37] and [14], and no publicly available implementation is available for [38].

Note that the main purpose of this work is not to claim that the implementation of our proposed framework is fundamentally more efficient than other special purposed ML systems, or to argue that Datalog is more suitable than the math-like syntax interfaces have provided in other ML platforms. Instead, we aim at demonstrating that it is possible to optimize a general recursive query engine to achieve the competitive or even better performance than special-purpose ML systems in a family of ML applications.

We use execution time as the evaluation metric in the experiments. Since BGD uses all training instances in one iteration, the results regarding accuracy/loss are the same for all systems. Therefore, we only report the end-to-end query execution time for models trained with BGD. For MGD we report the results of training loss vs. training time as it was done in many previous studies of ML systems. To ensure fairness, we allocate the same number of workers/servers and sufficient memory to guarantee the performance for different platforms. We ensure that algorithms on different platforms are equivalent in terms of workload and convergence by configuring the implementation on all systems with exact the same parameters.

In the experiments, the original LIBSVM data format can be supported by our approach and also by MLlib and PyTorch. For SystemML, we converted our data format into their supported binary format following the instructions in SystemML's official documentation, and we did not include this preprocessing time into the total query time.

Table 2 Statistics of datasets

Name	Cardinality	# Features	Size (GB)
URL	2,396,130	3,231,961	2.1
KDD10	19,264,097	29,890,095	4.8
KDD12	149,639,105	54,686,452	21.1
WEBSPAM	350,000	16,609,143	23.3

8.1.3 Environment

The experiments of all the four systems are conducted on a cluster with 16 node: one node acts as the master and other

[5] https://spark.apache.org/mllib/

[6] https://systemml.apache.org/

[7] https://pytorch.org/

15 nodes as workers. For the distributed computing, since our *Datalog* framework, SystemML and MLlib are all based on Apache Spark, they use the bulk synchronous parallel architecture. Meanwhile, PyTorch runs under the parameter server architecture. All nodes are connected with 1Gbit network. Each node runs Ubuntu 14.04 LTS and has an Intel i7-4770 CPU (3.40GHz, 4 core/8 thread), 32GB memory and a 1 TB 7200 RPM hard drive. Each worker node is allocated 30 GB RAM and 8 CPU cores (120 total cores) for execution. BigDatalog is built on top of Spark 2.0 and Hadoop 2.2. All systems are activated with in-memory computation by default. Since hype-parameter tuning is outside the scope of this paper, the hyper-parameter settings are the same for all systems: the learning rate is 10^{-2} and the number of iterations for BGD is 100.

8.2 End-to-end performance

To begin with, we report the end-to-end execution time of the three models trained with BGD. The results are shown in Fig. 9, where our approach is denoted as *Datalog*. Note that some results of SystemML and PyTorch are denoted by the word "OOM" in red, since they run out of memory under those settings. One thing we would like to clarify is that for PyTorch we directly use the GD implementation provided by the lib itself. Nevertheless, there might be some better ways to optimize the implementation and avoid the OOM issue, such as by additive gradient updates on mini-batches. Since such optimizations on PyTorch are out of scope of this paper, we just report results with its default implementation.

From the results, we can make the following observations:

Firstly, *Datalog* consistently outperforms the other two Spark-based systems MLlib and SystemML for all three models. SystemML has the worst performance as its optimizations focus on physical-level computation within one iteration rather than the whole iterative training process. Such results make sense since the strong point of SystemML lies in directly computing the ML models by matrix operations. As the bottleneck of the training process with BGD is not computation over large matrices but recursive gradient computation, SystemML cannot benefit from above optimizations. MLlib outperforms SystemML because it adopts a tree aggregate mechanism to accelerate the gradient computation in distributed environment; however, *Datalog* is approximately 2X to 4X faster than MLlib. Our preliminary investigations suggest that performance gains of our approach over MLlib come from higher-level logical optimizations, which were particularly successful in reducing shuffle operations.

Secondly, the performance of *Datalog* is comparable with that of PyTorch, one of the most popular special-purposed ML systems. On some datasets, such as the KDD10 dataset, *Datalog* even outperforms PyTorch by up to 2 times. This must be credited to our system's success in optimizing each computation step from planning to execution to fully harness the potential of the Spark engine. We also see that PyTorch requires much more memory: it runs out of memory on the large datasets KDD12 and WEBSPAM. A possible reason for that is that PyTorch needs additional memory to make a replica of gradients and parameters for each thread rather than each node. For large sparse dataset, PyTorch will run out of memory when broadcasting after an iteration.

Lastly, the advantage of *Datalog* over other competitors is more obvious on larger datasets. On the smallest dataset URL, the performance is comparable for all four systems. When it scales up to KDD10, MLlib and SystemML are approximately 2X and 5X slower than *Datalog*, respectively. For example, on the KDD10 dataset, the total execution time for Linear Regression on PyTorch, MLlib, and SystemML is 3889, 4689, 11351 seconds, respectively, while *Datalog* only takes 2338 seconds. We believe that is because, for small datasets the computation time of each iteration is relatively short. As a result, the communication time between workers will dominant the end-to-end execution time and the difference between different systems is not obvious. Meanwhile, for larger dataset the computation time becomes the bottleneck and thus the effect of our optimizations is more obvious. Finally for KDD12, SystemML runs out of memory and *Datalog* outperforms MLlib by 5X. A possible reason for which

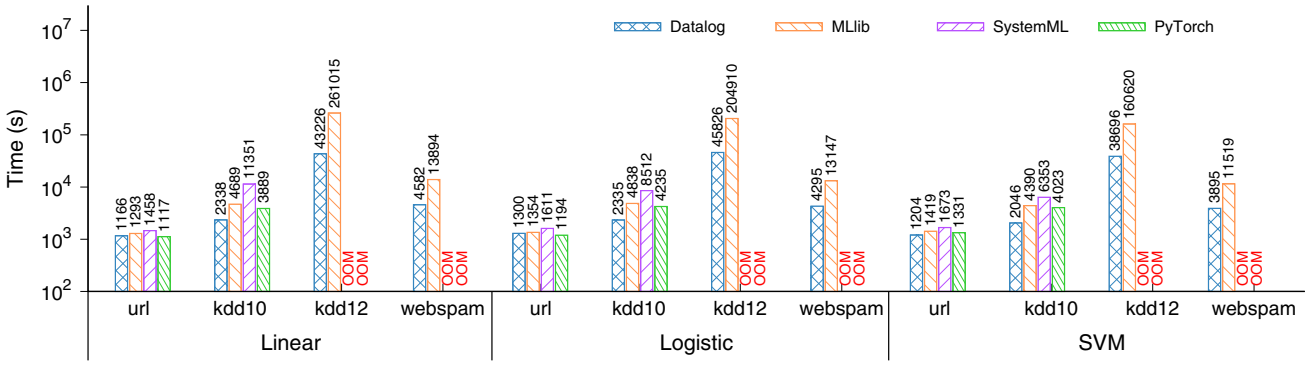

Fig. 9 Performance comparison: training with batch gradient descent

SystemML runs out of memory could be that it conducts the ML application in the way in which matrix operations are optimized. Thus, even for sparse datasets, SystemML requires large volumes of memory to keep the intermediate results.

Figure 11 shows the results of adding L_2 regularization on the three ML applications for KDD10, respectively. We can see that the trend of results with regularization is similar to that in Fig. 9.

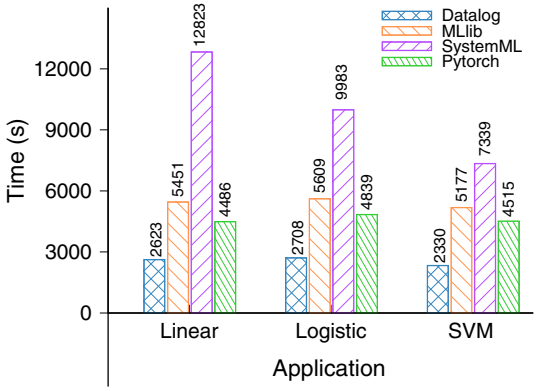

Fig. 11 Performance comparison with L_2 regularization

8.3 Results for mini-batch GD

Next, we report the experimental results on training the three ML models with the MGD optimizer. We set the batch size as 8,192 empirically. Due to space limitations, we only report the results on KDD10 dataset. On the other datasets without memory issues, the results have similar trends. For experiments with MGD, we do not fix the number of iterations. Instead, the training process will terminate when convergence is reached (when the difference of training losses between two adjacent iterations is smaller than 10^{-3} or the maximum 25,000 iterations is reached).

As we can see from Fig. 10, PyTorch has the best performance under most settings. This is not surprising since specialized ML systems have implemented several optimizations and improvements designed specifically for training with MGD. As it has been widely shown in previous studies, BGD is more suitable for ML systems based on relational engines, e.g., Spark and relational DBMS. Note that the main contribution claimed in this paper is to propose a purely declarative ML framework by taking advantage of the characteristics of Datalog, rather than implementing an ML system that provides richer and more efficient ML functions than other systems. Consequently, the main purpose of evaluation is to show that with the aggregates-in-recursion mechanism supported by sound optimization techniques, the ML workloads can be expressed by Datalog and its implementation can outperform other Spark-based systems. Remarkably, our implementation of MGD with trade-off did show very promising results in the quality of training. The training loss that *Datalog* achieves at convergence for Linear Regression, Logistic Regression and SVM is 0.418, 0.372 and 0.376, respectively, while that of PyTorch is 0.407, 0.363 and 0.365, respectively.

Moreover, we can see that *Datalog* converges faster than the other two Spark-based competitors while achieving similar training loss as PyTorch. For example, for the SVM model, *Datalog* requires only about 5,000 iterations to converge with 530 ms per iteration. Meanwhile, the results for SystemML are about 6,000 iterations with 1,048 ms per each iteration. Finally, MLlib had not reached converge after 20,000 seconds, which is beyond the x-axis of Fig. 10. A reason MLlib performs worst here might be that it does not exploit all the MGD optimization steps used in SystemML.

8.4 Scalability

In a final set of experiments, we test the performance of BGD on different systems when scaling up the size of the training data. For that we used the synthetic datasets proposed in the previous study [39]. We vary the size of the dataset from 10GB to 40GB. Other detailed settings of the synthetic data are the same as that discussed in Sect. 6. Using the

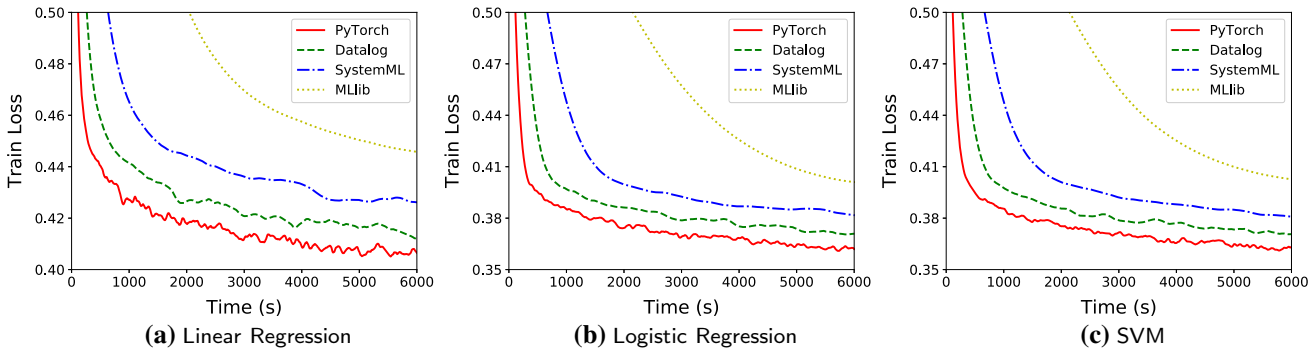

Fig. 10 Performance comparison: training with mini-batch gradient descent

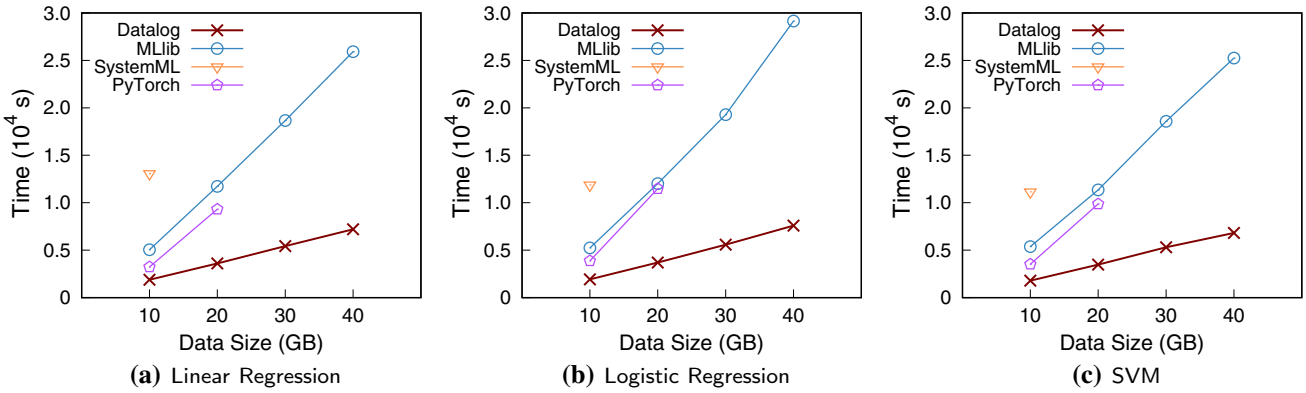

Fig. 12 Scalability

charts shown in Fig. 12, we discover that *Datalog* achieves nearly linear scalability for all three ML algorithms trained with BGD. This demonstrates the great potential of applying our approach to the workloads generated by larger training datasets.

Furthermore, we can also observe that *Datalog* consistently outperforms MLlib and SystemML for increasing cardinalities of the training sets. For example, for the Linear Regression model, *Datalog* outperforms MLlib by 2X to 6X and outperforms SystemML by up to one order of magnitude. Note that when the size of the dataset exceeds 20GB, PyTorch and SystemML run out of memory. Thus, many data points are missing for these systems in the figures. This further demonstrates the advantage of our framework over other Spark-based ML systems. Moreover, our *Datalog* also achieves comparable performance with the special-purposed ML system PyTorch in scalability.

8.5 Evaluate optimization techniques

To measure the effectiveness of each optimization proposed in Sect. 6, we use the Datalog programs to train Linear Regression (Linear), Logistic Regression (Logistic) and SVM with BGD on a synthetic dataset. The data generator used here is the one proposed in a previous experimental study for ML applications [39]. We use the option of sparse data with density 1.67×10^{-6}. The total size of training set is 40 GB. The training process of BGD is conducted over 100 iterations.

The effect of eliminating unnecessary evaluations (Sect. 6.1) is shown in Table 3. The results show that this optimization for the SN evaluation of nonlinear recursive programs for ML is quite substantial, which achieves up to about $1.5\times$ performance gain. This is hardly a surprise given that the full relations are replaced by the delta ones at every iteration of the SN computation.

The effects of applying the replica mechanism (Sect. 6.2) are shown in Table 4. We can see that with the help of replica mechanism, it achieves a performance gain of 3X to 3.4X. This underscores the considerable amount of shuffle operations that are removed from all iterations because of our carefully designed replica mechanism.

Table 5 shows the effect of scheduling optimizations (Sect. 6.3). The overall performance is improved over the un-optimized approach by approximately 1.2X. Actually the elimination of shuffle operations in $r_{3,4}$ can be done automatically once the replica mechanism is applied. Therefore, the performance gain brought by scheduling optimization is not so obvious compared with the other two optimizations described above.

8.6 Results on dense datasets

To include the whole spectrum of datasets and make a comprehensive evaluation, we also conduct experiments on a

Table 3 Nonlinear recursion optimization

Time (s)	Linear	Logistic	SVM
w/elimination	7196.4	7582.9	6814.6
w/o elimination	10358.1	11319.5	10166.7

Table 4 Effect of replica

Time (s)	Linear	Logistic	SVM
w/replica	7196.4	7582.9	6814.6
w/o replica	22664.9	26312.3	20660.0

Table 5 Effect of scheduling optimization

Time (s)	Linear	Logistic	SVM
w/optimization	7196.4	7582.9	6814.6
w/o optimization	7961.0	8339.2	7719.7

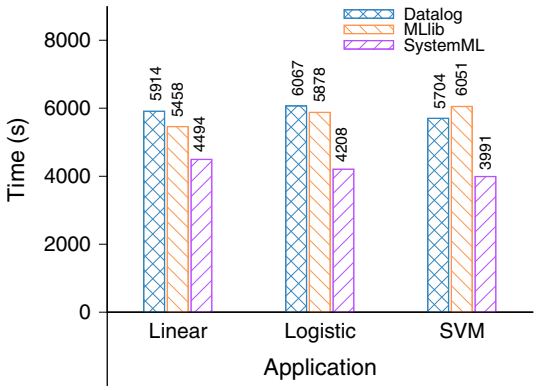

Fig. 13 Performance comparison on dense dataset

dense synthetic dataset. We continue using the synthetic datasets proposed in [39] but set the density as 0.5.

We set the cardinality of dataset as 30 GB to make sure that all systems will not run out of memory [8].

There is no doubt that PyTorch has much better performance than *Datalog*, MLlib and SystemML on dense data since it is optimized for supporting deep learning models, which involve many computations between dense matrices. Therefore, here we only show the results of comparing with the other two Spark-based systems SystemML and MLlib.

The results are shown in Fig. 13. We can see that *Datalog* still achieves comparable performance with SystemML and MLlib. Although our proposed framework is designed for applications with sparse vectors, it still has reasonable performance on dense ones. Indeed, Datalog-ML is optimized for applications on sparse training data, where the majority of dimensions are zeroes in one training instance. For dense datasets, the benefits of proposed optimizations are far from obvious and thus the resulting performance is not as good as that obtained in previous experiments. Therefore, we have included this last experiment to provide a more comprehensive and balanced view of characteristics of our proposed framework.

9 Related work

9.1 Datalog for machine learning

Previous efforts in expressing ML applications with Datalog include the following ones. Borkar et al. [40] proposed a declarative workflow system, which also supports ML functionalities. Bu et al. [38] developed a Datalog query interface for it. MLog [14] provided a set of imperative Datalog-style ML libraries over the TensorFlow system. LogiQL [37] proposed to express ML applications with Datalog and script-like constructs. These studies focus on using Datalog as part of the query interface. The work describe in this paper addresses the whole spectrum of advances needed to support effectively ML applications in Datalog and other declarative query languages such as SQL. These include (i) formal declarative semantics for the query language, (ii) efficient system implementations with very effective optimization on parallel platforms and (iii) enhancements providing usability and interoperability in a data frame environment.

9.2 Recursive query processing

A long stream of database research work on recursive query processing has sought to provide formal declarative semantics for the usage of aggregates in recursion [23,41,42]. In particular, Ross et al. [43,44] used semantics based on specialized lattices to express the use of min, max, count and sum, while Ganguly et al. [45] sought to optimize programs with extrema. More recently, Mazuran et al. [8] showed that continuous count and sum, are monotonic and thus can be used freely in recursion. Monotonic aggregates have been implemented in the Datalog system named DeALS [18] and scaled up to distributed systems [15] and multi-core machines [19]. Recently, [9] introduced the Pre-mappability (PreM) property under which programs using min and max in recursion are equivalent to aggregate-stratified programs. The extension of SQL with extrema in recursion based on PreM [26,46] based on PreM proved quite effective on graph applications. New opportunities for reducing staleness and communication costs in distributed data processing were studied in [28]. Past work has also recognized that Datalog is well-suited for large-scale analytical queries due to its amenability to data parallelism and the great expressive power of its recursive queries. In fact, Generalized Pivoting [47] and Parallel Semi-naive [48] techniques enable parallel evaluation of *Datalog* programs. OverLog [49] and NDlog [50] proved effective at providing declarative networking. Systems that use Datalog to support data analytics in distributed environments include SociaLite [51], LogicBlox [52], Myria [53] and GraphRex [6]. However, the challenges of ML applications were not tackled by these systems. Therefore, they cannot support the queries expressed in this paper.

9.3 Large-scale machine learning

Supporting large-scale machine learning applications has become a hot topic in the database community. Several research works aim at optimizing the performance of linear algebra, which provides a common formal representation language for machine learning algorithms [39,54–57]. Many

[8] Note that in previous experiments with sparse dataset, SystemML will run out of memory as it needs to convert the dataset into its own data format, which would be much larger than the original sparse dataset as it might add some information to complement the omitted zero-dimensions

previous studies focus on in-database machine learning. The basic idea is to formalize ML operators as optimization primitives and devise an engine on top of relational DBMS to solve the ML problem using such primitives [11,58]. SimSQL [59] employs a hybrid imperative and declarative framework to express linear models [12,60], Bayesian learning [61] as well as deep neural networks [13]. While most previous solutions require many additional primitives, our framework is a purely declarative one that can be realized using basic constructs of Datalog, or a simple relaxation of current SQL standards.

To take advantage of distributed data platforms, many ML frameworks were developed over Apache Spark as extensions. MLBase [62] proposes a declarative ML framework by providing APIs of high level programming languages. Anderson et al. [63] integrates Spark with MPI to improve the performance of graph and ML applications. KeystoneML [64] and Helix [65] provide more effective pipelines for ML workload. ML4all [66] optimizes computation of gradient descent algorithms. PS2 [6] integrates the parameter server with Apache Spark. Our work shows that the ML applications supported by such works can be expressed efficiently via Datalog by generalizing the existing query optimization and data parallelism techniques.

9.4 Machine learning and big data systems

Apache Spark [31] has been one of the most popular distributed data processing platforms which provides APIs for relational queries, graph analytics, data streaming and machine learning. DryadLINQ [67], REX [68] and Naiad [69] provide effective interfaces to support large-scale workloads with iterations. Distributed graph systems provide vertex-centric APIs for graph analytics workloads. Typical examples include Graphlab [70], Pregel [71] and GraphX [72].

Recently, many ML systems have emerged to efficiently support different kinds of ML algorithms in distributed environments. The parameter server architecture [73] opens up a new pathway to distributed model training. Examples adopting parameter servers include PyTorch [74], TensorFlow [75], Petuum [76] and MXNet [77]. SystemML [36] is a declarative ML framework with plan optimizations on top of Apache Spark. LMFAO [78] aims at optimizing the analytic workloads with batched aggregation, including the Linear Regression queries. Ray [79] provides a unified interface that supports multiple tasks and settings.

10 Conclusion

This paper has presented a powerful, declarative ML framework on top of Apache Spark with Datalog query interfaces. Thanks to the great expressive power of Datalog, users can write queries to express a series of ML algorithms trained by gradient descent optimizers without involving new constructs. The power of allowing aggregates in recursive Datalog programs is illustrated by the fact that it can be used for both expressing ad hoc queries and for producing a library of ML functions, i.e., a task for which procedural languages are normally required. We formally demonstrated that the training process expressed with Datalog programs has formal semantics by showing the Pre-Countable Cardinality property. Then, we proposed several planning and optimization techniques to efficiently support the evaluation of Datalog programs with complex recursions, which are essential to support ML applications. We also provided an equivalent SQL-based implementation with a very succinct syntax based on current SQL standard. Experiments on large-scale real-world benchmarks demonstrated the superiority of our proposed framework over existing ML systems.

As future work, we plan to extend our framework to cover more machine learning algorithms, such as deep neural networks. Besides, we also plan to extend our system to GPU settings and further optimize the performance.

Acknowledgements The authors would like to thank the reviewers and the editor for many suggested improvements.

References

1. Meng, X., Bradley, J.K., Yavuz, B., et al.: Mllib: machine learning in apache spark. J. Mach. Learn. Res. **17**, 34:1–34:7 (2016)
2. Apache Mahout . https://mahout.apache.org/
3. Hellerstein, J.M., Ré, C., Schoppmann, F., Wang, D.Z., Fratkin, E., Gorajek, A., Ng, K.S., Welton, C., Feng, X., Li, K., Kumar, A.: The madlib analytics library or MAD skills, the SQL. Proc. VLDB Endow. **5**(12), 1700–1711 (2012)
4. Li, Y., Wang, J., Li, M., Das, A., Gu, J., Zaniolo, C.: Kddlog: Performance and scalability in knowledge discovery by declarative queries with aggregates. In: *IEEE International Conference on Data Engineering (ICDE)*, (2021)
5. Bellomarini, L., Sallinger, E., Gottlob, G.: The vadalog system: datalog-based reasoning for knowledge graphs. Proc. VLDB Endow. **11**(9), 975–987 (2018)
6. Zhang, Q., Acharya, A., Chen, H., Arora, S., Chen, A., Liu, V., Loo, B.T.: Optimizing declarative graph queries at large scale. In:

ACM International Conference on Management of Data, SIGMOD Conference, 1411–1428, (2019)
7. Seo, J., Park, J., Shin, J., Lam, M.S.: Distributed socialite: a datalog-based language for large-scale graph analysis. Proc. VLDB Endow. **6**(14), 1906–1917 (2013)
8. Mazuran, M., Serra, E., Zaniolo, C.: Extending the power of datalog recursion. VLDB J. **22**(4), 471–493 (2013)
9. Zaniolo, C., Yang, M., Das, A., Shkapsky, A., Condie, T., Interlandi, M.: Fixpoint semantics and optimization of recursive datalog programs with aggregates. Theory Pract. Log. Program. **17**(5–6), 1048–1065 (2017)
10. Zaniolo, C., Yang, M., Interlandi, M., Das, A., Shkapsky, A., Condie, T.: Declarative bigdata algorithms via aggregates and relational database dependencies. In: *Alberto Mendelzon International Workshop on Foundations of Data Management*, (2018)
11. Feng, X., Kumar, A., Recht, B., Ré, C.: Towards a unified architecture for in-rdbms analytics. In: *ACM International Conference on Management of Data, SIGMOD Conference*, 325–336, (2012)
12. Luo, S., Gao, Z.J., Gubanov, M.N., Perez, L.L., Jermaine, C.M.: Scalable linear algebra on a relational database system. In: *IEEE International Conference on Data Engineering (ICDE)*, 523–534, (2017)
13. Jankov, D., Luo, S., Yuan, B., Cai, Z., Zou, J., Jermaine, C., Gao, Z.J.: Declarative recursive computation on an RDBMS. Proc. VLDB Endow. **12**(7), 822–835 (2019)
14. Li, X., Cui, B., Chen, Y., Wu, W., Zhang, C.: Mlog: towards declarative in-database machine learning. Proc. VLDB Endow. **10**(12), 1933–1936 (2017)
15. Shkapsky, A., Yang, M., Interlandi, M., Chiu, H., Condie, T., Zaniolo, C.: Big data analytics with datalog queries on spark. In: *ACM International Conference on Management of Data, SIGMOD Conference*, 1135–1149, (2016)
16. Condie, T., Das, A., Interlandi, M., Shkapsky, A., Yang, M., Zaniolo, C.: Scaling-up reasoning and advanced analytics on bigdata. Theory Pract. Log. Program. **18**(5–6), 806–845 (2018)
17. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Boston (1995)
18. Shkapsky, A., Yang, M., Zaniolo, C.: Optimizing recursive queries with monotonic aggregates in deals. In: *IEEE International Conference on Data Engineering (ICDE)*, 867–878, (2015)
19. Yang, M., Shkapsky, A., Zaniolo, C.: Scaling up the performance of more powerful datalog systems on multicore machines. VLDB J. **26**(2), 229–248 (2017)
20. Syed, U., Vassilvitskii, S.: SQML: large-scale in-database machine learning with pure SQL. In: *ACM Symposium on Cloud Computing, (SoCC)*, 659, (2017)
21. Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R.T., Subrahmanian, V.S., Zicari, R.: Advanced Database Systems. Morgan Kaufmann, Massachusetts (1997)
22. LIBSVM Data . https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/
23. Ganguly, S., Greco, S., Zaniolo, C.: Minimum and maximum predicates in logic programming. In: *ACM Symposium on Principles of Database Systems (PODS)*, 154–163, (1991)
24. Sudarshan, S., Ramakrishnan, R.: Aggregation and relevance in deductive databases. In: *International Conference on Very Large Data Bases (VLDB)*, 501–511, (1991)
25. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: *International Conference on Logic Programming (ICLP)*, 1070–1080, (1988)
26. Gu, J., Watanabe, Y., Mazza, W., Shkapsky, A., Yang, M., Ding, L., Zaniolo, C.: Rasql: Greater power and performance for big data analytics with recursive-aggregate-sql on spark. In: *ACM International Conference on Management of Data, SIGMOD Conference*, 467–484, (2019)
27. Das, A., Li, Y., Wang, J., Li, M., Zaniolo, C.: Bigdata applications from graph analytics to machine learning by aggregates in recursion. In: *International Conference on Logic Programming (ICLP)*, pages 273–279, (2019)
28. Das, A., Zaniolo, C.: A case for stale synchronous distributed model for declarative recursive computation. Theory Pract. Log. Program. **19**(5–6), 1056–1072 (2019)
29. Zaniolo, C., Das, A., Gu, J., Li, Y., Li, M., Wang, J.: Monotonic properties of completed aggregates in recursive queries. *CoRR*, arXiv:1910.08888, (2019)
30. Arni, F., Ong, K., Tsur, S., Wang, H., Zaniolo, C.: The deductive database system LDL++. Theory Pract. Log. Program. **3**(1), 61–94 (2003)
31. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 15–28, (2012)
32. Wolfson, O., Silberschatz, A.: Distributed processing of logic programs. In: *ACM International Conference on Management of Data, SIGMOD Conference*, 329–336, (1988)
33. Ma, J., Saul, L.K., Savage, S., Voelker, G.M.: Identifying suspicious urls: an application of large-scale online learning. In: *International Conference on Machine Learning (ICML)*, 681–688, (2009)
34. Juan, Y., Zhuang, Y., Chin, W., Lin, C.: Field-aware factorization machines for CTR prediction. In: *ACM Conference on Recommender Systems (RecSys)*, 43–50, (2016)
35. Webb, S., Caverlee, J., Pu, C.: Introducing the webb spam corpus: using email spam to identify web spam automatically. In: *The Third Conference on Email and Anti-Spam (CEAS)*, (2006)
36. Boehm, M., Dusenberry, M., Eriksson, D., Evfimievski, A.V., Manshadi, F.M., Pansare, N., Reinwald, B., Reiss, F., Sen, P., Surve, A., Tatikonda, S.: SystemML: declarative machine learning on spark. Proc. VLDB Endow. **9**(13), 1425–1436 (2016)
37. Makrynioti, N., Vasiloglou, N., Pasalic, E., Vassalos, V.: Modelling machine learning algorithms on relational data with datalog. In: *DEEM@ACM International Conference on Management of Data, SIGMOD Conference*, 5:1–5:4, (2018)
38. Bu, Y., Borkar, V.R., Carey, M.J., Rosen, J., Polyzotis, N., Condie, T., Weimer, M., Ramakrishnan, R.: Scaling datalog for machine learning on big data. *CoRR*, arXiv:1203.0160, (2012)
39. Thomas, A., Kumar, A.: A comparative evaluation of systems for scalable linear algebra-based analytics. Proc. VLDB Endow. **11**(13), 2168–2182 (2018)
40. Borkar, V.R., Bu, Y., Carey, M.J., Rosen, J., Polyzotis, N., Condie, T., Weimer, M., Ramakrishnan, R.: Declarative systems for large-scale machine learning. IEEE Data Eng. Bull. **35**(2), 24–32 (2012)
41. Mumick, I.S, Pirahesh, H., Ramakrishnan, R.: The magic of duplicates and aggregates. In: *International Conference on Very Large Data Bases (VLDB)*, 264–277, (1990)
42. Furfaro, F., Greco, S., Ganguly, S., Zaniolo, C.: Pushing extrema aggregates to optimize logic queries. Inf. Syst. **27**(5), 321–343 (2002)
43. Ross, K.A., Sagiv, Y.: Monotonic aggregation in deductive databases. In: *ACM Symposium on Principles of Database Systems (PODS)*, 114–126, (1992)
44. Ross, K.A., Sagiv, Y.: Monotonic aggregation in deductive database. J. Comput. Syst. Sci. **54**(1), 79–97 (1997)
45. Ganguly, S., Greco, S., Zaniolo, C.: Extrema predicates in deductive databases. J. Comput. Syst. Sci. **51**(2), 244–259 (1995)
46. Wang, J., Xiao, G., Gu, J., Wu, J., Zaniolo, C.: RASQL: A powerful language and its system for big data applications. In: *ACM International Conference on Management of Data, SIGMOD Conference*, 2673–2676, (2020)

47. Seib, J., Lausen, G.: Parallelizing datalog programs by generalized pivoting. In: *ACM Symposium on Principles of Database Systems (PODS)*, 241–251, (1991)
48. Shaw, M., Koutris, P., Howe, B., Suciu, D.: Optimizing large-scale semi-naïve datalog evaluation in hadoop. Datalog Acad. Ind. 165–176, (2012)
49. Loo, B.T., Condie, T., Hellerstein, J.M., Maniatis, P., Roscoe, T., Stoica, I.: Implementing declarative overlays. In: *ACM Symposium on Operating Systems Principles (SOSP)*, 75–90, (2005)
50. Loo, B.T., Condie, T., Garofalakis, M.N., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking: language, execution and optimization. In: *ACM International Conference on Management of Data, SIGMOD Conference*, 97–108, (2006)
51. Seo, J., Guo, S., Lam, M. S.: Socialite: Datalog extensions for efficient social network analysis. In: *IEEE International Conference on Data Engineering (ICDE)*, 278–289, (2013)
52. Aref, M., Cate, B. ten, Green, T. J., Kimelfeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T. L., Washburn, G.: Design and implementation of the logicblox system. In: *ACM International Conference on Management of Data, SIGMOD Conference*, 1371–1382, (2015)
53. Wang, J., Balazinska, M., Halperin, D.: Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. Proc. VLDB Endow. **8**(12), 1542–1553 (2015)
54. Elgohary, A., Boehm, M., Haas, P.J., Reiss, F.R., Reinwald, B.: Compressed linear algebra for large-scale machine learning. Proc. VLDB Endow. **9**(12), 960–971 (2016)
55. Elgohary, A., Boehm, M., Haas, P.J., Reiss, F.R., Reinwald, B.: Compressed linear algebra for large-scale machine learning. VLDB J. **27**(5), 719–744 (2018)
56. Chen, L., Kumar, A., Naughton, J.F., Patel, J.M.: Towards linear algebra over normalized data. Proc. VLDB Endow. **10**(11), 1214–1225 (2017)
57. Elgamal, T., Luo, S., Boehm, M., Evfimievski, A.V., Tatikonda, S., Reinwald, B., Sen, P.: SPOOF: sum-product optimization and operator fusion for large-scale machine learning. In *Innov. Data Syst. Res. (CIDR)*, (2017)
58. Schleich, M., Olteanu, D., Ciucanu, R.: Learning linear regression models over factorized joins. In: *ACM International Conference on Management of Data, SIGMOD Conference*, 3–18, (2016)
59. Cai, Z., Vagena, Z., Perez, L.L., Arumugam, S., Haas, P.J., Jermaine, C.M.: Simulation of database-valued markov chains using simsql. In: *ACM International Conference on Management of Data, SIGMOD Conference*, 637–648, (2013)
60. Luo, S., Gao, Z.J., Gubanov, M.N., Perez, L.L., Jermaine, C.M.: Scalable linear algebra on a relational database system. IEEE Trans. Knowl. Data Eng. **31**(7), 1224–1238 (2019)
61. Gao, Z.J., Luo, S., Perez, L.L., Jermaine, C.: The BUDS language for distributed bayesian machine learning. In: *ACM International Conference on Management of Data, SIGMOD Conference*, 961–976, (2017)
62. Kraska, T., Talwalkar, A., Duchi, J.C., Griffith, R., Franklin, M.J., Jordan, M.I.: Mlbase: A distributed machine-learning system. Innov. Data Syst. Res. (CIDR)
63. Anderson, M.J., Smith, S., Sundaram, N., Capota, M., Zhao, Z., Dulloor, S., Satish, N., Willke, T.L.: Bridging the gap between HPC and big data frameworks. Proc. VLDB Endow. **10**(8), 901–912 (2017)
64. Sparks, E.R., Venkataraman, S., Kaftan, T., Franklin, M.J., Recht, B.: Keystoneml: Optimizing pipelines for large-scale advanced analytics. In: *IEEE International Conference on Data Engineering (ICDE)*, 535–546, (2017)
65. Xin, D., Macke, S., Ma, L., Liu, J., Song, S., Parameswaran, A.G.: Helix: holistic optimization for accelerating iterative machine learning. Proc. VLDB Endow. **12**(4), 446–460 (2018)
66. Kaoudi, Z., Quiané-Ruiz, J., Thirumuruganathan, S., Chawla, S., Agrawal, D.: A cost-based optimizer for gradient descent optimization. In: *ACM International Conference on Management of Data, SIGMOD Conference*, 977–992, (2017)
67. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P.K., Currey, J.: Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1–14, (2008)
68. Mihaylov, S.R., Ives, Z.G., Guha, S.: REX: recursive, delta-based data-centric computation. Proc. VLDB Endow. **5**(11), 1280–1291 (2012)
69. McSherry, F., Murray, D.G, Isaacs, R., Isard, M.: Differential dataflow. Innov. Data Syst. Res. (CIDR)
70. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Distributed graphlab: a framework for machine learning in the cloud. Proc. VLDB Endow. **5**(8), 716–727 (2012)
71. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: *ACM International Conference on Management of Data, SIGMOD Conference*, 135–146, (2010)
72. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: Graphx: Graph processing in a distributed dataflow framework. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 599–613, (2014)
73. Li, M., Andersen, D.G., Park, J.W., Smola, A.J., Ahmed, A., Josifovski, V., Long, J., Shekita, E.J., Su, B.: Scaling distributed machine learning with the parameter server. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 583–598, (2014)
74. Steiner, B., DeVito, Z., Chintala, S., et al.: Pytorch: An imperative style, high-performance deep learning library. In: *Annual Conference on Neural Information Processing Systems (NeurIPS)*, (2019)
75. Abadi, M., Barham, P., Chen, J., et al.: Tensorflow: a system for large-scale machine learning. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 265–283, (2016)
76. Xing, E.P., Ho, Q., Dai, W., Kim, J.K., Wei, J., Lee, S., Zheng, X., Xie, P., Kumar, A., Yu, Y.: Petuum: A new platform for distributed machine learning on big data. In: *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 1335–1344, (2015)
77. Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., Zhang,Z.: Mxnet: a flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, arXiv:1512.01274, (2015)
78. Schleich, M., Olteanu, D., Khamis, M.A., Ngo, H.Q., Nguyen, X.: A layered aggregate engine for analytics workloads. In: *ACM International Conference on Management of Data, SIGMOD Conference*, 1642–1659, (2019)
79. Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M.I., Stoica, I.: Ray: a distributed framework for emerging AI applications. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 561–577, (2018)