Declarative Systems for Large-Scale Machine Learning

Vinayak Borkar[1], Yingyi Bu[1], Michael J. Carey[1], Joshua Rosen[2],
Neoklis Polyzotis[2], Tyson Condie[3], Markus Weimer[3] and Raghu Ramakrishnan[3]
[1]University of California, Irvine, [2]University of California, Santa Cruz,
[3]Yahoo! Research

Abstract

In this article, we make the case for a declarative foundation for data-intensive machine learning systems. Instead of creating a new system for each specific flavor of machine learning task, or hard-coding new optimizations, we argue for the use of recursive queries to program a variety of machine learning algorithms. By taking this approach, database query optimization techniques can be utilized to identify effective execution plans, and the resulting runtime plans can be executed on a single unified data-parallel query processing engine.

1 Introduction

Supported by the proliferation of "Big Data" platforms such as Apache Hadoop, organizations are collecting and analyzing ever larger datasets. Increasingly, machine learning (ML) is at the core of data analysis for actionable business insights and optimizations. Today, machine learning is deployed widely: recommender systems drive the sales of most online shops; classifiers help keep spam out of our email accounts; computational advertising systems drive revenues; content recommenders provide targeted user experiences; machine-learned models suggest new friends, new jobs, and a variety of activities relevant to our profiles in social networks. Machine learning is also enabling scientists to interpret and draw new insights from massive datasets in many domains, including such fields as astronomy, high-energy physics, and computational biology.

The availability of powerful distributed data platforms and the widespread success of machine learning has led to a virtuous cycle wherein organizations are now investing in gathering a wider range of (even bigger!) datasets and addressing an even broader range of tasks. Unfortunately, the basic MapReduce framework commonly provided by first-generation "Big Data analytics" platforms like Hadoop lacks an essential feature for machine learning: MapReduce does not support iteration (or equivalently, recursion) or certain key features required to efficiently iterate "around" a MapReduce program. Programmers building ML models on such systems are forced to implement looping in ad-hoc ways outside the core MapReduce framework; this makes their programming task much harder, and it often also yields inefficient programs in the end. This lack of support has motivated the recent development of various specialized approaches or libraries to support iterative programming on large clusters. Examples include Pregel, Spark, and Mahout, each of which aims to support a particular family of tasks, e.g., graph analysis or certain types of ML models, efficiently. Meanwhile, recent MapReduce

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

extensions such as HaLoop, Twister, and PrItr aim at directly addressing the iteration outage in MapReduce; they do so at the physical level, however.

In our proposed approach to scalable machine learning, programmers need not learn to operate and use a plethora of distinct platforms. In the database world, relational database systems separate the conceptual, logical and physical schemas in order to achieve *logical* and *physical data independence*. Similarly, we aim to open the door in the machine learning world to principled optimizations by achieving a separation between:

- The user's program (in a domain-specific language that provides a library of available templates and user-definable functions) and the underlying *logical query*, expressed in Datalog. This shields the user from any changes in the logical framework, e.g., how the Datalog program is optimized.
- The logical Datalog query and an optimized *physical runtime plan*, reflecting details related to caching, storage, indexing, the logic for incremental evaluation and re-execution in the face of failures, etc. This ensures that any enhancements to the plan execution engine will automatically translate to more efficient execution, without requiring users to re-program their tasks to take advantage of the enhancements.

In essence, the separation identifies "modules" (such as the plan execution engine or the optimization of the logical Datalog program) where localized enhancements lead to higher overall efficiency. To illustrate our approach, we will describe a new high-level programming language called ScalOps, in which the data scientist encodes their machine learning task. We will then describe the ScalOps translation into Datalog for the example of a specific class of supervised machine learning algorithms. Lastly, we will demonstrate that an appropriately chosen data-intensive computing substrate, namely Hyracks [3], is able to handle the computational requirements of such programs through the application of dataflow processing techniques like those used in parallel databases [9].

2 High-level Abstractions for Machine Learning

A large class of machine learning algorithms are expressible in the statistical query model [11]. Statistical queries (e.g. max, min, sum, ...) themselves decompose into a data-local function and subsequent aggregation [6]. The MapReduce [8] programming model has been successful at performing this decomposition over a cluster of machines while abstracting away runtime concerns—like parallelism, fault-tolerance, and scheduling—from the programmer. However, for tasks in the machine learning domain, MapReduce is missing a key ingredient: iteration. Furthermore, the MapReduce programming model contains a group-by key component that is not needed for many machine learning tasks. Therefore, we have defined an alternative programming model called Iterative Map-Reduce-Update, which consists of the following three UDFs:

`map` receives read-only global state value (i.e., the model) as side information and is applied to all training data points in parallel.

`reduce` aggregates the `map`-output into a single aggregate value. This function is commutative and associative.

`update` receives the combined aggregate value and produces a new global state value for the next iteration or indicates that no additional iteration is necessary.

An Iterative Map-Reduce-Update runtime executes a series of iterations, each of which first calls `map` with the required arguments, then performs a global `reduce` aggregation, and lastly makes a call to `update`.

Example: Convex Optimization A large class of machine learning—including Support Vector Machines, Linear and Logistic Regression and structured prediction tasks such as machine translation—can be cast as convex optimization problems, which in turn can be solved efficiently using an Iterative Map-Reduce-Update approach [14]. The objective is to minimize the sum over all data points of the divergences (the loss) between the model's prediction and the known data. Usually, the loss function is convex and differentiable in the model,

Listing 1: Batch Gradient Descent in ScalOps

```
1 def bgd(xy: Table[Example], g:(Example, Vector)=>Vector, e:Double, l:Double) =
2   loop(zeros(1000), 0 until 100) {
3     w => (w - (xy.map(x => g(x, w)).reduce(_+_) * e) * (1.0 - e * l)
4   }
```

and therefore the *gradient* of the loss function can be used in iterative optimization algorithms such as Batch Gradient Descent. [1] Each model update step is a single Map-Reduce-Update iteration. The `map` UDF computes (loss, gradient) tuples for all data points, using the current model as side input. The `reduce` UDF sums those up and `update` computes a new model, which becomes the input to the next iteration of the optimization algorithm.

2.1 ScalOps: A Domain-Specific Language for Machine Learning

The Iterative Map-Reduce-Update programming model can be naturally represented in a high-level language. ScalOps is an internal domain-specific language (DSL) that we are developing for "Big Data" analytics [13]. ScalOps moves beyond single pass data analytics (i.e., MapReduce) to include multi-pass workloads, supporting iteration over algorithms expressed as relational queries on the training and model data. As a concrete example, consider the update rule for learning an L2-regularized linear regression model through batch gradient descent (BGD).[2] In each iteration of BGD, the gradient of the loss function with respect to the model w is computed for each data point and summed up:

$$w_{t+1} = \left(w_t - \eta \sum_{x,y} \delta_{w_t}(l(y, \langle w_t, x \rangle)) \right) (1 - \eta\lambda) \tag{1}$$

Here, λ and η are the regularization constant and learning rate, and δ_{w_t} denotes the derivative with respect to w_t. The sum in (1) decomposes per data point (x, y) and can therefore be trivially parallelized over the data set. Furthermore, the application of the gradient computation can be phrased as `map` in a MapReduce framework, while the sum itself can be phrased as the `reduce` step, possibly implemented via an aggregation tree for additional performance.

Listing 1 shows an implementation of this parallelized batch gradient descent in ScalOps. Line 1 defines a function `bgd` with the following parameters: the input data `xy`, the gradient function `g` (δ_{w_t} in (1)), the learning rate `e` (η) and the regularization constant `l` (λ). Line 2 defines the body of the function to be a `loop`, which is ScalOps' looping construct. It accepts three parameters: (a) Input state—here, we assume 1000 dimensions in the weight vector—(b) a "while" condition that (in this case [3]) specifies a range (`0 until 100`) and (c) a loop body. The loop body is a function from input state to output state, which in this case is from a Vector `w` to its new value after the iteration. In the loop body, the gradient is computed for each data point via a `map` call and then summed up in a `reduce` call. Note that `_+_` is a Scala shorthand for the function literal `(x,y) => x+y`.

ScalOps bridges the gap between imperative and declarative code by using data structures and language constructs familiar to the data scientist. This is evident in Listing 1, which is nearly a 1:1 translation of Equation 1. ScalOps query plans capture the semantics of the user's code in a form that can be reasoned over and optimized. For instance, since the query plan contains the loop information we can recognize the need to cache as much of `xy` in memory as possible, without explicit user hints. Furthermore, the expression inside of the loop body itself

[1] A more detailed discussion can be found in the Appendix of our technical report [5].

[2] Batch Gradient Descent, while not a state-of-the-art optimization algorithm, exemplifies the general structure of a larger class of convex optimization algorithms.

[3] We also support boolean expressions on the state object i.e., the weight vector.

is also captured. [4] This could offer further optimization opportunities reminiscent to traditional compilers i.e., dead-code elimination and constant-propagation.

3 A Declarative Approach to Machine Learning

The goal of machine learning (ML) is to turn observational data into a *model* that can be used to predict for or explain yet unseen data. While the range of machine learning techniques is broad, most can be understood in terms of three complementary perspectives:

- **ML as Search:** The process of training a model can be viewed as a *search problem*. A domain expert writes a program with an objective function that contains, possibly millions of, unknown parameters, which together form the *model*. A runtime program *searches* for a good set of parameters based on the objective function. A *search strategy* enumerates the parameter space to find a model that can correctly capture the known data and accurately predict unknown instances.
- **ML as Iterative Refinement:** Training a model can be viewed as iteratively closing the gap between the model and underlying reality being modeled, necessitating iterative/recursive programming.
- **ML as Graph Computation:** Often, the interdependence between the model parameters is expressible as a graph, where nodes are the parameters (e.g., statistical/random variables) and edges encode interdependence. This view gives rise to the notion of *graphical models*, and algorithms often reflect the graph structure closely (e.g., propagating refinements of parameter estimates iteratively along the edges, aggregating the inputs at each vertex).

Interestingly, each of these perspectives lends itself readily to expression as a Datalog program, and thereby to efficient execution by applying a rich array of optimization techniques from the database literature. The fact that Datalog is well-suited for iterative computations and for graph-centric programming is well-known [12], and it has also been demonstrated that Datalog is well-suited to search problems [7]. The natural fit between Datalog and ML programming has also been recognized by others [2, 10], but not at "Big Data" scale. It is our goal to make the optimizations and theory behind Datalog available to large-scale machine learning while facilitating the use of established programming models. To that end, we advocate *compilation* from higher order programming models to Datalog and subsequently physical execution plans.

3.1 Iterative Map-Reduce-Update in Datalog

We begin with the predicates and functions that form the building blocks of the Datalog program.

training_data(Id,Datum) is an extensional predicate corresponding to the training dataset.

model(J,M) is an intensional predicate recording the global model M at iteration J. Initialization is performed through the function predicate *init_model(M)* that returns an initial global model.

***map*(M, R,** S**)** is a UDF predicate that receives as input the current model M and a data element R, and generates a statistic S as output.

reduce is an aggregation UDF function that aggregates several per-record statistics into one statistic.

***update*(J, M, AggrS,** *NewM)* is a UDF predicate that receives as input the current iteration J, the current model M and the aggregated statistic $AggrS$, and generates a new model $NewM$.

[4] We currently capture basic math operators (+, -, *, /) over primitive types (int, float, double, vector, etc.).

Listing 2: Datalog runtime for the Iterative Map-Reduce-Update programming model. The temporal argument is defined by the *J* variable.

```
% Initialize the global model
G1: model(0, M) :- init_model(M).

% Compute and aggregate all outbound messages
G2: collect(J, reduce<S>) :- model(J, M),
  training_data(Id, R), map(R, M, S).

% Compute the new model
G3: model(J+1, NewM) :-
   collect(J, AggrS), model(J, M),
   update(J, M, AggrS, NewM), M != NewM.
```

The Datalog program in Listing 2 is a specification for the Iterative Map-Reduce-Update programming model. A special temporal variable (`J`) is used to track the iteration number. Rule *G1* performs initialization of the global model at iteration 0 using the function predicate `init_model`, which takes no arguments and returns the initial model in the (`M`) variable. Rules *G2* and *G3* implement the logic of a single iteration. Let us consider first rule *G2*. The evaluation of `model(J, M)` and `training_data(Id, R)` binds (`M`) and (`R`) to the current global model and a data record respectively. Subsequently, the evaluation of `map(M, R, S)` invokes the UDF that generates a data statistic (`S`) based on the input bindings. Finally, the statistics from all records are aggregated in the head predicate using the `reduce` UDF.

Rule *G3* updates the global data model using the aggregated statistics. The first two body predicates simply bind (`M`) to the current global model and (`AggrS`) to the aggregated statistics respectively. The subsequent function predicate `update(J, M, AggrS, NewM)` calls the `update` UDF; accepting (`J, M, AggrS`) as input and producing an updated global model in the (`NewM`) variable. The head predicate records the updated global model at time-step `J+1`.

Program termination is handled in rule *G3*. Specifically, `update` is assumed to return the same model when convergence is achieved. In that case, the predicate `M != NewM` in the body of *G3* becomes false and we can prove that the program terminates. Typically, `update` achieves this convergence property by placing a bound on the number of iterations and/or a threshold on the difference between the current and the new model.

3.2 Semantics and Correctness

The Datalog program for Iterative Map-Reduce-Update contains recursion that involves an aggregation, and hence we need to show that it has a well-defined output. Subsequently, we have to prove that this output corresponds to the output of the target programming model. In this section, we summarize a more formal analysis of these two properties from our technical report [5], which is based on the following theorem.

Theorem 1: The Datalog program in Listing 2 is XY-stratified [16].

The proof can be found in the Appendix of our technical report [5] and is based on the machinery developed in [16]. XY-stratified Datalog is a more general class than stratified Datalog. In a nutshell, it includes programs whose evaluation can be stratified based on data dependencies even though the rules are not stratified.

Following XY-stratification, we can prove that the output of the program in Listing 2 is computed from an initialization step that fires *G1*, followed by several iterations where each iteration fires *G2* and then *G3*. By translating the body of each rule to the corresponding extended relational algebra expression, and taking into account the data dependencies between rules, it is straightforward to arrive at the logical plan shown in Figure 1.

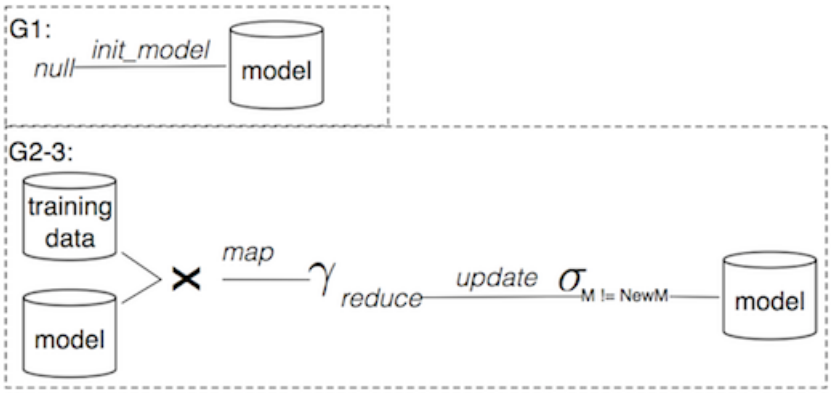

Figure 1: Logical query plan for Iterative Map-Reduce-Update.

The plan is divided into two separate dataflows, each labeled by the rules they implement in Listing 2. The dataflow labeled *G*1 initializes the global model using the `init_model` UDF, which takes no input, and produces the initial model. The *G*2–3 dataflow executes one iteration. The `cross-product` operator combines the model with each tuple in the training dataset, and then calls the `map` UDF on each result. (This part corresponds to the body of rule *G2*.) The mapped output is passed to a `group-all` operator, which uses the `reduce` aggregate (e.g., sum) to produce a scalar value. (This part corresponds to the head of rule *G2*.) The aggregate value, together with the model, is then passed to the `update` UDF, the result of which is checked for a new model. (This part corresponds to the body of *G3*.) A new model triggers a subsequent iteration, otherwise the `update` output is dropped and the computation terminates. (This part corresponds to the head of *G3*.)

4 A Runtime for Machine Learning

This section presents a high-level description of the physical plan that executes our Iterative Map-Reduce-Update programming model on a cluster of machines. A more detailed description can be found in our technical report [5]. Our physical plan consists of dataflow processing elements (operators) that execute in the Hyracks runtime [3]. Hyracks splits each operator into multiple tasks that execute in parallel on a distributed set of machines. Similar to parallel database systems and Hadoop, each task operates on a single partition of the input data. In Section 4.1, we describe the structure of the physical plan template and discuss its tunable parameters. Section 4.2 then explores the space of choices that can be made when executing this physical plan on an arbitrary cluster with given resources and input data.

4.1 Iterative Map-Reduce-Update Physical Plan

Figure 2 depicts the physical plan for the Iterative Map-Reduce-Update programming model. The top dataflow loads the input data from HDFS, parses it into an internal representation (e.g., binary formated features), and partitions it over N cluster machines, each of which caches as much of the data as possible, spilling to disk if necessary. The bottom dataflow executes the computation associated with the iteration step. The **Driver** of the iteration is responsible for seeding the initial global model and scheduling each iteration. The `map` step is parallelized across some number of nodes in the cluster determined by the optimizer. [5] Each `map` task sends a data statistic (e.g., a loss and gradient in a BGD computation) to a task participating in the leaf-level of an aggregation tree. This aggregation tree is balanced with a parameterized fan-in f (e.g., $f = 2$ yields a binary

[5] Reflected in the partitioning strategy chosen for the data loading step.

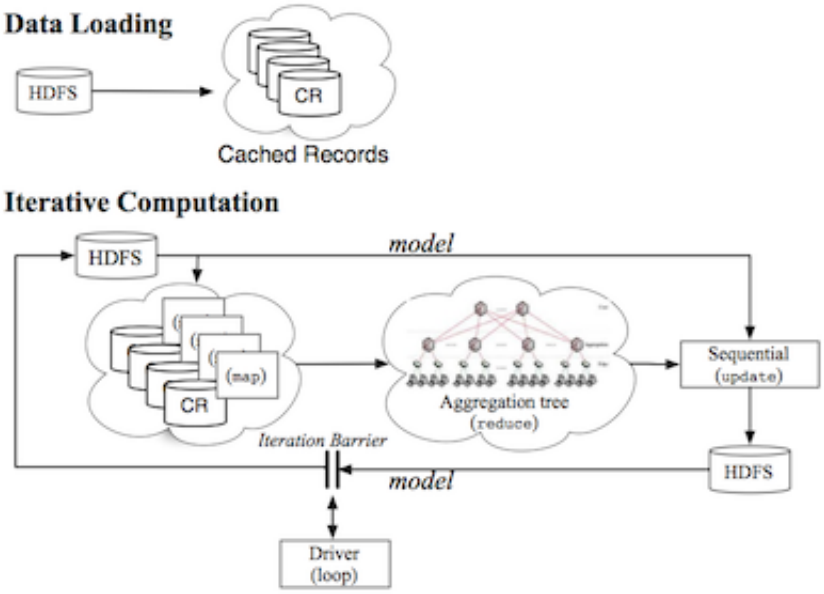

Figure 2: Hyracks physical plan for Iterative Map-Reduce-Update.

tree) determined by the optimizer. The final aggregate statistic is passed to the **Sequential** operator, which updates the global model and stores it in HDFS, along with some indicator that determines if another iteration is required based on evaluating the "loop" condition against the global model. The **Driver** detects this update and schedules another iteration on a positive indicator, terminating the computation otherwise.

This description of this physical plan highlights two choices to be determined by the optimizer—the number of nodes allocated for the map step, and the fan-in of the aggregation tree used in the reduce step.

4.2 The Plan Space

There are several considerations that must be taken into account when mapping the physical plan in Figure 2 to an actual cluster of machines. Many of these considerations are well-established techniques for executing data-parallel operators on a cluster of machines, and are largely independent of the resources available and data statistics. We begin by discussing these "universal" optimizations for arriving at an execution plan. Next, we highlight those choices that are dependent on the cluster configuration (i.e., amount of resources) and computation parameters (i.e., input data and aggregate value sizes).

4.2.1 Universal Optimizations

Data-local scheduling is generally considered an optimal choice for executing a dataflow of operators in a cluster environment: a map task is therefore scheduled on the machine that hosts its input data. **Loop-aware scheduling** ensures that the task state (i.e., the hash-table state associated with a join condition) is preserved across iterations. **Caching** of immutable data can offer significant speed-ups between iterations. However, careful consideration is required when the available resources do not allow for such caching. For example, Spark [15] assumes that sufficient main memory is always available to cache the data across iterations, but offers no assistance to the programmer for tuning this parameter, which may not even be feasible with the given resource allocation. **Efficient data representation** can offer significant performance improvements. For instance, we use a binary formated file, which has benefits in terms of space savings over simple Java objects.

4.2.2 Per-Program Optimizer Decisions

The optimizations discussed in Section 4.2.1 apply equally to all jobs and can be considered best practices inspired by the best-performing systems in the literature. This leaves us with two optimization decisions that depend on the cluster and computation parameters, which we highlight below. We have developed a theoretical foundation for an optimizer that can make these choices effectively, but elide the details due to space constraints.

Data partitioning determines the number of `map` tasks in an Iterative Map-Reduce-Update physical plan. For a given job and a maximum number N_{max} of machines available to it, the optimizer needs to decide which number $N \leq N_{max}$ of machines to request for the job. This decision is not trivial, even when ignoring the multi-job nature of today's clusters: more machines reduce the time in the map phase but increase the cost of the reduce phase, since more objects must be aggregated. The goal of data partitioning is to find the right trade-off between `map` and `reduce` costs: the `map` time *decreases* linearly with the number of machines and the `reduce` time *increases* logarithmically (see below). Using measurements of the actual times taken to process a single record in `map` and `reduce`, an optimal partitioning can be found in closed form.

Aggregation tree structure involves finding the optimal fan-in of a single `reduce` node in a balanced aggregation tree. Aggregation trees are commonly used to parallelize the `reduce` function. For example, Hadoop uses a `combiner` function to perform a single level aggregation tree, and Vowpal Wabbit [1] uses a binary aggregation tree. For the programming model we described above —with a blocking `reduce` operation— we have shown that the optimal fan-in for the aggregation tree is independent of the problem, both theoretically and empirically. In fact, its actual value is a per-cluster constant. The intuition behind this derivation lies in the difference between the optimal aggregation tree for a small versus a large number of leaf (`map`) nodes being sheer scaling, a process for which the fan-in of the tree does not change. Furthermore, there is an independence between the transfer time and aggregation time, in that time spent per aggregation tree level and the number of levels balance each other out.

5 Conclusion

The growing demand for machine learning is pushing both industry and academia to design new types of highly scalable iterative computing systems. Examples include Mahout, Pregel, Spark, Twister, HaLoop, and PrItr. However, today's specialized machine learning platforms all tend to mix logical representations and physical implementations. As a result, today's platforms 1) require their developers to rebuild critical components and to hardcode optimization strategies and 2) limit themselves to specific runtime implementations that usually only (naturally) fit a limited subset of the potential machine learning workloads. This leads to the current state of practice: implementing new scalable machine learning algorithms is very labor-intensive and the overall data processing pipeline involves multiple disparate tools hooked together with file- and workflow-based glue.

In contrast, we have advocated a declarative foundation on which specialized machine learning workflows can be easily constructed and readily tuned. We have verified our approach with Datalog implementations of two popular programming models from the machine learning domain: Iterative Map-Reduce-Update, for deriving linear models, and Pregel, for graphical algorithms (see [5]). The resulting Datalog programs are compact, tunable to a specific task (e.g., Batch Gradient Descent and PageRank), and translated to optimized physical plans. Our experimental results show that on a large real-world dataset and machine cluster, our optimized plans are very competitive with other systems that target the given class of ML tasks (see [5]). Furthermore, we demonstrated that our approach can offer a plan tailored to a given target task and data for a given machine resource allocation. In contrast, in our large experiments, Spark failed due to main-memory limitations and Hadoop succeeded but ran an order-of-magnitude less efficiently.

The work reported here is just a first step. We are currently developing the ScalOps query processing components required to automate the remaining translation steps; these include the Planner/Optimizer as well as a more general algebraic foundation based on extending the Algebricks query algebra and rewrite rule framework

of ASTERIX [4]. We also plan to investigate support for a wider range of machine learning tasks and for a more asynchronous, GraphLab-inspired programming model for encoding graphical algorithms.

References

[1] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *CoRR*, abs/1110.4198, 2011.

[2] Ashima Atul. Compact implementation of distributed inference algorithms for network[s]. Master's thesis, EECS Department, University of California, Berkeley, Mar 2009.

[3] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011.

[4] Vinayak R. Borkar, Michael J. Carey, and Chen Li. Inside "Big Data Management": Ogres, Onions, or Parfaits? In *EDBT*, 2012.

[5] Yingyi Bu, Vinayak Borkar, Michael J. Carey, Joshua Rosen, Neoklis Polyzotis, Tyson Condie, Markus Weimer, and Raghu Ramakrishnan. Scaling datalog for machine learning on big data. Technical report, CoRR. URL: http://arxiv.org/submit/427482 or http://isg.ics.uci.edu/techreport/TR2012-03.pdf, 2012.

[6] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, pages 281–288, 2006.

[7] Tyson Condie, David Chu, Joseph M. Hellerstein, and Petros Maniatis. Evita raced: metacompilation for declarative networks. *PVLDB*, 1(1):1153–1165, 2008.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[9] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[10] Jason Eisner and Nathaniel W. Filardo. Dyna: Extending Datalog for modern AI. In Tim Furche, Georg Gottlob, Giovanni Grasso, Oege de Moor, and Andrew Sellers, editors, *Datalog 2.0*, Lecture Notes in Computer Science. Springer, 2011. 40 pages.

[11] Michael Kearns. Efficient noise-tolerant learning from statistical queries. In *Journal of the ACM*, pages 392–401. ACM Press, 1993.

[12] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23:125–149, 1993.

[13] Markus Weimer, Tyson Condie, and Raghu Ramakrishnan. Machine learning in scalops, a higher order cloud computing language. In *BigLearn Workshop (NIPS 2011)*, Sierra Nevada, Spain, December 2011.

[14] Markus Weimer, Sriram Rao, and Martin Zinkevich. A convenient framework for efficient parallel multipass algorithms. In *LCCC : NIPS 2010 Workshop on Learning on Cores, Clusters and Clouds*, December 2010.

[15] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. HotCloud'10, page 10, Berkeley, CA, USA, 2010.

[16] Carlo Zaniolo, Natraj Arni, and KayLiang Ong. Negation and aggregates in recursive rules: the LDL++ approach. In *DOOD*, pages 204–221, 1993.