Declarative Data Analytics: a Survey

Nantia Makrynioti and Vasilis Vassalos

Abstract—The area of declarative data analytics explores the application of the declarative paradigm on data science and machine learning. It proposes declarative languages for expressing data analysis tasks and develops systems which optimize programs written in those languages. The execution engine can be either centralized or distributed, as the declarative paradigm advocates independence from particular physical implementations. The survey explores a wide range of declarative data analysis frameworks by examining both the programming model and the optimization techniques used, in order to provide conclusions on the current state of the art in the area and identify open challenges.

Index Terms—Declarative Programming, Data Science, Machine Learning, Large-scale Analytics

1 Introduction

With the rapid growth of world wide web (WWW) and the development of social networks, the available amount of data has exploded. This availability has encouraged many companies and organizations in recent years to collect and analyse data, in order to extract information and gain valuable knowledge. Topic classification, sentiment analysis, spam filtering, fraud and anomaly detection are only a few analytics tasks that have gained considerable popularity in recent years, along with more traditional warehouse queries which gather statistics from data.

Data mining and machine learning (ML) are core elements of data analysis tasks. However, developing such algorithms needs not only expertise in software engineering, but also a solid mathematical background in order to interpret correctly and efficiently the mathematical computations into a program. Even when experimenting with black box libraries, evaluating various algorithms for a task and tuning their parameters, in order to produce an effective model, is a time-consuming process. Things become more complicated when we want to leverage parallelization on clusters of independent computers for analysing big data. Details concerning load balancing, scheduling or fault tolerance can be overwhelming even for an experienced software engineer.

Research on the data management domain recently started tackling the above issues by developing systems that aim at providing high-level primitives for building data science tasks, which at the same time hide low-level details of the solution or distributed execution. MapReduce [1] and Dryad [2] were the first frameworks that paved the way of large-scale analytics. However, these initial efforts suffered from low usability, as they offered expressive but still low-level languages to program data analysis algorithms. Soon the need for higher-level declarative programming languages became apparent. Systems, such as Hive [3], Pig [4] and Scope [5], offer higher-level languages that enable developers to write entire programs or parts of them in declarative style. Then, these programs are automatically translated to MapReduce jobs or Dryad vertices that form a directed acyclic graph (DAG), which is optimized for efficient distributed execution.

Apart from the programming model, optimization techniques are another important issue that systems for declarative data analytics address. As the declarative paradigm implies that the user expresses the logical structure of a program, there may be many implementations that compute the same result, but differ at efficiency level. Rewritings of the logical structure and physical implementation of a program, which harness the properties of the operators of a language in a way similar to relational query optimization, as well as the use of techniques from the domain of compilers are explored in the context of declarative data science.

In this survey, we study systems for declarative analytics by focusing on the aforementioned aspects: programming model and optimization techniques. The rest of the paper is organized as follows. Section 2 describes the data analysis tasks that are considered when we present the capabilities of different systems and languages and presents seven properties serving as pillars for declarativity. In section 3 we discuss the example we will use to analyse the classes of programming models in section 4, where domain specific languages and libraries are classified into these classes. Section 5 examines optimization techniques that are employed by declarative data analysis systems, whereas section 6 presents a comparison between the surveyed systems based on specific properties and discusses future directions. Finally, section 7 concludes the survey.

2 Scope

In this work we focus on two popular classes of data analysis tasks: descriptive and predictive analytics. Descriptive analytics provide insights for the past. This usually involves complex queries on a database system to extract aggregated information, such as sum and average statistics for a collection of records. Data are stored in relations and relational algebra operators are used to form queries on them. On the other hand, predictive analytics study historical data in order to identify trends and produce

- *N. Makrynioti and V. Vassalos are with the Department of Informatics, Athens University of Economics and Business, Athens, 10434, Greece. Email: makriniotik@aueb.gr, vassalos@aueb.gr*

predictions for future events. ML algorithms for regression, classification and clustering hold a dominant position in this class. In this second category, we will address deterministic ML algorithms, either supervised or unsupervised. Many of the algorithms that fall into this category, e.g. linear / logistic regression, support vector machines (SVM) and k-means, are expressed using linear algebra operators on matrices and vectors, and employ an iterative refinement process to minimize/maximize a given objective function. The aforementioned algorithms are useful for a variety of tasks, including demand forecasting, market segmentation and spam filtering.

What is not covered: regarding the notable area of deep learning (DL), we consider the study of DL systems as a topic on its own. DL frameworks [6], [7], [8] provide infrastructure aiming at setting up neural networks easily. Their functionality focuses on building layers and choosing activation functions and optimizers and as such their design is fairly restrictive in describing other ML algorithms. The only exceptions that we make in this area regard TensorFlow [9] and PyTorch [10]. Although these frameworks are popular in the development of neural networks, their toolkits expand outside the area of deep learning, from linear algebra and distribution functions to out of the box algorithms for classification and regression. Furthermore, probabilistic graphical models, such as Bayesian or Markov networks, systems such as DeepDive [11], which allow the definition of inference rules in a declarative manner, as well as statistical relational learning are orthogonal to the scope of this survey and we will not study declarative languages for expressing algorithms in these categories [1]. Finally, although Model Management Selection Systems (MSMS) [14], [15], [16] are relevant to predictive analytics, we do not include them in this survey, as these systems work at the task level rather than the algorithm level and optimize for both accuracy and efficiency, instead of solely efficiency. For these reasons most of the properties we propose do not fit the design of MSMS and we find it unsuitable to proceed to a one-to-one comparison between them and the rest of the systems surveyed. The mission of MSMS poses interesting challenges, which constitute an important survey topic on its own.

In the scope of descriptive and predictive analytics described above, we will examine languages and systems from two perspectives: programming model and optimization techniques. In each of these perspectives we identify specific properties, some of which are also proposed in [17], that play an important role on the objectives of the declarative programming paradigm. These properties ensure necessary means to decouple the specification of data analytics programs from the underlying runtime engine and as a result provide the ability to design different implementations for a single specification, which is in alignment with the "describe what you want the program to achieve rather than how to achieve it" mantra of declarative programming. Along with its definition we also explain how each property contributes to these goals. The purpose of the seven properties presented below is to provide a way to measure declarativity under the specific scope of the survey. As we present different categories of data analytics systems in the following sections, we discuss how they stand in the context of the proposed declarativity properties.

1. Regarding TensorFlow and PyTorch, we only cover their core APIs and do not consider libraries and tools inside their ecosystem, such as TensorFlow Probability [12] and Pyro [13] which are used for graphical models.

- **Data abstractions**: matrices, vectors, tables and any other data types are exposed as abstractions and are independent from their physical implementation in the system, e.g. distributed or dense/sparse implementations of a matrix. If a system provides multiple data abstractions, this property requires that none of these abstractions exposes aspects of its physical implementation to the user. Abstract data types decouple an algorithm specification from the execution engine and allow a system to optimize internal storage of physical implementations.
- **Data processing operators**: basic data processing (DP) operators such as join or group by, which are common in relational databases, need to be supported. Relational primitives have well-defined semantics and let the user describe data processing tasks in a similar manner as with database query languages.
- **Advanced analytics operators**: support for primitives widely used in ML, such as linear algebra operators and probability distribution functions. Such operators have also well-defined semantics and fit the mathematical description of ML algorithms.
- **Plan Optimization**: users' programs are automatically optimized by the system based on properties and available implementations of the involved operators. Due to DP and ML operators with known semantics, a system can reason about equivalences via commutative/associative properties, and reorganization of operations. Well-defined semantics is the key here, in order to generate equivalent results through different implementations.
- **Limited dependence on UDFs**: the main operators should not serve as second-order functions that take as input user-defined code in imperative/functional languages with unknown semantics. This property goes hand-in-hand with the previous one, because code with unknown semantics, as it is the case with user-defined functions, is challenging to optimize [18], [19], [20].
- **Lack of control flow**: the user does not have access to control flow constructs, which specify a specific order for the execution of the program. This facilitates the specification of a problem as a set of data dependencies rather than a series of statements to follow.
- **Automatic computation of the solution**: the minimization/maximization of an objective function for finding the optimal parameters of a model is a shared process between a wide class of ML algorithms and is achieved using a handful of mathematical optimization routines. A usual example is gradient descent, which needs to be written by the user in many of the systems covered by this survey. Automating these processes by computing the derivatives of a

function with automatic differentiation mechanisms and templating mathematical optimization processes as a special kind of operation adds a level of abstraction and releases the user from coding repetitive processes based on non-trivial mathematical details.

The focus of the survey is conceptual and architectural. In principle the existence of optimization techniques facilitates better performance. However, conducting a straight out performance comparison in such a wide and diverse range of systems provides no meaningful information. We refer the reader to [21] for a benchmark involving a subset of the systems surveyed in this paper. We will also see in our analysis that some systems are more coupled with their execution engine than others. We consider the aspects of declarativity as orthogonal to parallel and distributed execution. That being said, especially in the area of data processing and descriptive analytics, the paradigm of SQL showed empirically that a declarative language with properties similar to the ones we discuss right above has been successfully and more easily parallelized [22], [23], [24] than a non-declarative one.

3 Running example

To demonstrate the programming model of the various categories of systems, we will use Linear Regression (LR) supplemented with a filtering process of training data at the beginning. In LR the dependent variable is computed as in equation 1. Features (x_i) are independent variables whose values are provided by training data, whereas the values of weights (w_i) are unknown and will be provided by optimizing the LR model.

$$\hat{y}(w) = \sum_{i=1}^{m} x_i w_i,\ m = number\ of\ features \tag{1}$$

To find the optimal values of weights, LR minimizes/-maximizes an objective function. In our examples we minimize Least Squared Error between predictions and actual values of training observations (see equation 2).

$$\min J(w) = \frac{1}{n}\sum_{i=1}^{n}(y_i - w x_i)^2,\ n = number\ of\ observations \tag{2}$$

The minimization of an objective function is achieved by a mathematical optimization algorithm. Whenever such an algorithm is needed in our coding examples, we implement gradient descent. Gradient descent finds the minimum of a function by iteratively moving towards the negative of its gradient as it is shown in equation 3.

$$w = w - \frac{\partial}{\partial w} J(w) \tag{3}$$

The particular task at hand is to predict the median value of houses based on a number of features about a suburb, such as crime rate, distance from employment centers, etc. To train the LR model we use the Boston housing dataset [2], which includes this kind of attributes for the area of Boston. In our example, before training, we preprocess the dataset to filter observations that are very close to Charles river.

The selected running example covers many of the characteristics typically seen in both supervised and unsupervised ML algorithms, as well as functionality that is useful in descriptive analytic queries. In order to implement the example at hand, we first need a suitable set of data structures and operators to express the LR function and Least Squared Error, which serves as the objective function. In order to minimize the objective function, we either need an implementation of a mathematical optimization algorithm or iteration constructs to be able to express the iterative process of minimization. Finally, for the preprocessing of training data, we need a filtering operator, similar to the one supported in relational algebra.

4 Programming Model

Systems for large-scale data science started as libraries of machine learning/data mining algorithms, but in the more recent years they have evolved into domain-specific languages that support primitives to code data science tasks and emphasize on declarative programming. The latter has become more popular based on the argument that the simplification of the development of such tasks offers more flexibility to the users to customize their solutions than black-box libraries targeted to a particular execution engine. A number of approaches is followed to fulfil this goal. In the following subsections we describe each approach in detail and categorize relevant systems according to their design choices on the programming model.

4.1 Libraries of Algorithms

In this subsection we present libraries of machine learning/data mining algorithms, such as SVM and k-means, implemented with the primitives of an execution engine. The user is able to write data analysis tasks by calling functions to load data from various data sources, transform data or use ML algorithms to analyse them. The programming paradigm is the same as that of a developer writing code for a single machine and calling functions from a third-party library or executing commands from an interactive shell. This code or set of commands is then executed on a specific, possibly distributed, platform. Below we present some examples of such libraries and analyze the programming model using a code snippet [3] that calls functions from the MLlib [25] library. There are also single node implementations of machine learning / data mining software in imperative languages, such as scikit-learn [26] and Weka [27], but in this section we focus on libraries that run on top of a platform.

MLlib is a scalable machine learning library on top of Spark [28]. It includes algorithms for classification, regression, clustering, recommendations and other useful tasks, which the user can either run via interactive shells in Scala and Python or import in her code by calling functions from APIs in both the aforementioned programming languages, as well as Java. For historical reasons, we also mention

2. https://www.cs.toronto.edu/ delve/data/boston/bostonDetail.html

3. Similar examples using MLlib can be found on https://spark.apache.org/mllib/

here the Apache Mahout project [29], which started as a library of ML algorithms implemented on Hadoop. Later the project took a different direction and currently it develops and maintains a Scala DSL (Domain Specific Language) for linear algebra, which we will discuss further in section 4.4.

Moving away from distributed processing frameworks, MADlib [30] is a library of in-database methods for machine learning and data analysis. The broad know-how that has been developed over the years for database systems makes them a promising candidate to encompass data analytics. MADlib provides a library of SQL-based ML algorithms, which run on database engines. SQL operators are combined with user defined functions (UDF) in Python and C++ that implement iteration, linear algebra and matrix operations, which are prevalent in ML algorithms. Popular data analysis tasks, such as classification, clustering and regression, are already implemented in MADlib.

Table 1 summarizes the list of algorithms included in each library.

Let us illustrate the programming model of such libraries using the running example of LR described in section 3.

Listing 1: Linear Regression using MLlib library

```
// Features and Labels
JavaRDD<LabeledPoint> trainingData = data.map(line -> {
        String[] tokens = line.split(",");
        String[] features = tokens[1].split(" ");
        double[] feature_vector = new double[features.length];
        for (int i = 0; i < features.length; i++){
                feature_vector[i] = Double.parseDouble(features[i]);
        }
        return new LabeledPoint(Double.parseDouble(tokens[0]),
                Vectors.dense(feature_vector));
});

// Data Filtering
JavaRDD<LabeledPoint> filteredTrainingData = trainingData.filter(
    point -> {
        return (point.features().toArray()[3]==0);
});

filteredTrainingData.cache();

int iterations = 200
double learning_rate = 0.0000001

// Training using Linear Regression
final LinearRegressionModel model = LinearRegressionWithSGD.train
    (JavaRDD.toRDD(filteredTrainingData), iterations, learning_rate);

// Generating predictions
JavaRDD<Tuple2<Double, Double>> predictions_labels =
    filteredTrainingData.map(point -> {
        double prediction = model.predict(point.features());
        return new Tuple2<Double, Double>(prediction, point.label())
                ;
});
```

The workflow in listing 1 begins with creating a vector of features for each training observation. Both the features and the target value are stored in an object that represents a labeled point, i.e. a data point that is annotated with a target value. The set of labelled points are stored in a RDD (Resilient Distributed Dataset) [28], a Spark abstraction for a collection of elements partitioned across the nodes of a cluster. After filtering the points based on a specific feature, a new RDD is created with the same structure, which is used to train a LR model. The model is then loaded and can give predictions for new data. In this example we output

TABLE 1: List of Algorithms Provided by Each Library

Algorithm	**Mahout**	**MLlib**	**MADlib**
Logistic Regression	✓	✓	✓
Naive Bayes	✓	✓	✓
Complementaty Naive Bayes	✓		
Random Forest	✓	✓	✓
Hidden Markov Models	✓		
Multilayer Perceptron	✓		
Linear SVM		✓	✓
Decision Trees		✓	✓
Gradient Boosted Trees		✓	
Regression			
Linear Regression		✓	✓
Isotonic Regression		✓	
Ordinal Regression			✓
Clustering			
k-means	✓	✓	✓
Fuzzy k-means	✓		
Streaming k-Means	✓	✓	
Spectral Clustering	✓		
Canopy Clustering	✓		
Bisecting k-Means		✓	
Gaussian Mixture		✓	
Power Iteration Clustering (PIC)		✓	
Recommendations			
User-Based Collaboration Filtering	✓		
Item-Based Collaboration Filtering	✓		
Matrix Factorization	✓	✓	✓
Matrix Fact. ALS	✓	✓	
Matrix Fact. ALS (Impl. Feedback)	✓		
Weighted Matrix Factorization	✓		
SVD++	✓		
Topic Modelling			
LDA	✓	✓	✓
Dimensionality Reduction			
SVD	✓	✓	
PCA	✓	✓	✓
QR Decomposition	✓		
Lanczos Algorithm	✓		
Principal Component Projection			✓
Data Mining			
Apriori			✓
FP-Growth		✓	
Association Rules		✓	
Optimization Methods / Solvers			
Stohastic Gradient Descent		✓	
L-BFGS		✓	
Conjucate Gradient		✓	
Dense/Sparse Linear Systems		✓	
Time Series Analysis			
ARIMA			✓
Statistics			
Summary Statistics		✓	✓
Hypothesis Testing		✓	✓
Streaming Significance Testing		✓	
Pearson's Correlation			✓
Cardinality Estimators			✓
Text Analysis			
Term Frequency			✓
Sparse TF-IDF Vectors from Text	✓	✓	
Collocations	✓		
Email Archive Parsing	✓		
Utility Functions			
PMML Export		✓	✓
XML Parsing	✓		
RowSimilarityJob	✓		
Feature Transformations		✓	
Low-Rank Matrix Factorization			✓
Probabilistic Graphical Models			
Conditional Random Fields			✓

predictions again for training data just to showcase how this can be done using MLlib.

We can see that the MLlib library provides classes for ML algorithms, such as LinearRegressionWithSGD, and user defined code is needed mainly for preprocessing tasks, like transforming data to the format that is required by the algorithm. However, in case users want to run an ML algorithm that is not provided by the library, they have to implement it by using the programming model of the underlying platform.

Because libraries are developed using a specific programming model or language, it depends on this choice whether the seven declarativity properties described in section 2 are covered. For example, regarding the property of data processing operators, MLlib is built on Spark, which provides operators for joining or filtering RDDs, whereas in MADlib one can apply relational operators on tables. Also these platforms may or may not support optimization of programs. For these reasons declaritivity properties do not apply in the context of libraries and we will exclude them from the final comparison table at the end of the paper.

4.2 Hybrids of SQL and MapReduce

This class includes languages, such as Jaql [31], Stratosphere's Meteor [32], Pig Latin [4] and U-SQL [33] (based on Scope [5]), which aim to offer a programming model between SQL and MapReduce. The reasoning behind this approach is based on the fact that SQL is too rigid for some data analysis tasks, but on the other hand MapReduce is too low-level and needs custom code even for the simplest operations. Custom code not only requires effort to be written, but it also increases the time spent on debugging and maintenance. Hence, programs in these languages are sequences of steps as in procedural programming, but with each step performing a single high-level transformation, similar to SQL operations (e.g. filtering and grouping). Meteor also supports domain-specific operations, such as part-of-speech and sentence annotation, duplicate detection, record linkage and other common data analysis tasks.

In the aforementioned languages developers can define workflows incrementally by storing intermediate results in variables or using pipe syntax to forward them to the next expression instead of describing the final desired outcome as in SQL. Nevertheless, these intermediate results are generated via declarative operators and the user does not need to describe how these operators are implemented. Moreover, these languages usually provide a wider range of operators than the limited set of primitives used in MapReduce framework. In case custom code is needed, users can write user-defined functions. Programs written in those languages are translated automatically by the system into lower level code using the programming model of an execution engine, such as MapReduce and Nephele [34].

Concerning the data model, most of the languages in this class revolve around the concept of tuple known from databases or adopt a JSON-like model. Two important aspects of these models are schemaless records and arbitrary nesting. Schema may not be needed or is specified on the fly and fields of a tuple may store non-atomic values, i.e. tuples, bags and maps. The argument behind this feature is that developers of procedural programming are more familiar with storing multiple values in data structures, such as arrays, sets, maps, than with normalizing data to tables.

Let us now present the aforementioned features with more detail using the implementation of the Linear Regression algorithm in Pig Latin. As stated above, Pig Latin does not provide implementations of ML algorithms, so the user has to use its operations and structures to write the code. We use gradient descent for optimizing the weight values and our implementation in Pig Latin also includes code for this procedure.

In listing 2 we compute the prediction for each instance, the total error and a single iteration of gradient descent. Given that we have loaded training data from a file to a bag of tuples, i.e. `input_data`, we start with filtering housing instances based on their proximity to Charles river. Then, we create a new tuple for each instance consisting of the target value and a nested tuple of features, in order to transform data to a more convenient format for subsequent operations. These preprocessed tuples are stored in bag `input_data_preprocessed`. To combine each feature with its corresponding weight and compute products between them, we implement a UDF. The sum of the differences between predictions and actual target values gives us the total error, which we want to minimize using gradient descent. The derivatives and the weight updates for gradient descent are computed and stored in bags `gradients` and `weight_updates`. Finally, we store the new weight values in a file.

Listing 2: Linear Regression in Pig Latin

```
-- Filtering observations
filtered_input_data = FILTER input_data BY f3 == 0;

-- Initialize weights
weight_data = LOAD '$input_weights' USING PigStorage(',') AS (w0:
    double, w1:double, w2:double, w3:double, w4:double, w5:
    double, w6:double, w7:double, w8:double, w9:double, w10:
    double, w11:double, w12:double);

weight_data = FOREACH weight_data GENERATE TOTUPLE($0 ..
    $12) AS weight_vector;

-- features and labels
input_data_preprocessed = FOREACH filtered_input_data GENERATE
     $13 AS response, TOTUPLE($0 .. $12) AS feature_vector;

weight_feature_tuples = CROSS weight_data,
    input_data_preprocessed;
weight_feature_pairs = FOREACH weight_feature_tuples GENERATE
    response as response:double, WeightFeaturePair(weight_vector,
     feature_vector) as pairs:bag{t:tuple(weight:double, feature:
    double, dimension:int)};

-- Generate predictions
predictions = FOREACH weight_feature_pairs {products=foreach pairs
     generate weight*feature as product:double; prediction = (double
    )SUM(products.product); GENERATE FLATTEN(pairs) AS (
    weight, feature, dimension), response as response, prediction as
    prediction;};

-- Errors between prediction and actual target value
errors = FOREACH predictions {error = (prediction - response);
    GENERATE weight as weight, feature as feature, dimension as
    dimension, error as error;};

-- Gradient Descent steps
gradients = FOREACH errors GENERATE weight, dimension, feature*
    error as feature_error;
```

```
weight_updates = FOREACH (GROUP gradients BY (dimension,
        weight)) {learning_rate=0.0000001/(double)COUNT(gradients);
        total = SUM(gradients.feature_error); weight_update=
        learning_rate*total; new_weight=group.weight−weight_update;
        GENERATE group.dimension as dimension, new_weight as
        new_weight;};

new_weights = FOREACH (GROUP weight_updates ALL) {in_order =
        ORDER weight_updates BY dimension ASC; tuple_weights =
        BagToTuple(in_order.new_weight); GENERATE tuple_weights
        AS weights;};

final_weights = FOREACH new_weights GENERATE FLATTEN(
        weights);

STORE final_weights INTO 'pig_LR/weight_values' USING PigStorage(
        ',');
```

The reason we use UDFs in this code snippet is the lack of support for iteration over columns in Pig Latin. In our example, we have two vectors, one for features and one for weights, and we need to combine each feature with its corresponding weight. To do so, we need to dive into the contents of the vectors and process them element by element. Although iteration over tuples of a bag is provided by using the `foreach` operator, there is no obvious way to do the same for columns without explicitly stating column numbers or names, which is impossible when we have a large number of columns. It is important to note that UDFs are treated as a black box and their code is not optimized, as it happens with Pig Latin's operators.

Recall that gradient descent is an iterative algorithm. So far we have presented one iteration of it, but in practice we run gradient descent for many iterations or until error converges. However, Pig Latin does not support loop constructs and in order to run this Pig script iteratively, we need an external program, i.e. a driver, in any programming language supporting iteration. The driver program should initialize weights with zero and run the Pig script for a number of times passing as parameters the paths for feature and weight files. Due to the lack of iteration support we are forced to write and read data from files, and repeat preprocessing steps, which would be avoided if we were able to use variables for intermediate results after each iteration.

Regarding the data model, we can observe that during the loading of the weights a schema is defined. However, this is not necessary and the LOAD command could have ended after the delimiter definition. Nested data structures are also used since the initial bag of tuples that stores scalar values is transformed to the bag called `input_data_preprocessed`, where each tuple consists of a scalar and a nested tuple.

The main limitation of these languages is the lack of iteration support. There are some operators for iteration over data, such as the FOREACH operator of Pig Latin, but the user is not able to declare that a given group of operators will be repeated for a number of times or as long as a certain condition holds. Iterative processes are common in ML algorithms, which are frequently used for more advanced data analysis tasks. Given the lack of control flow and automatic solving methods to provide the values of the model parameters, the approach of this class of systems is less frequently used in the context of machine learning, as different directions that we will discuss in the rest of the survey proposed more efficient architectures for the described limitations.

Overall, this class of systems supports five out of the seven declarativity properties in the context of data analytics, namely data abstractions, data processing operators, plan optimization, which will be discussed in section 5, lack of control flow and UDF free operators.

4.3 Extensions of MapReduce

In this section, programming models propose extensions to the MapReduce model. They employ the idea of passing first-order functions to operators that are integrated to an imperative/functional language, and extend it mainly at two directions. First, the set of operators provided by these models is enriched compared to MapReduce, which is limited to only two operators. For example, join, filter, union or variants of the map operator, such as flatmap, are implemented by these models, which try to offer an out of the box solution instead of the tedious fitting of these common operations to the MapReduce model. For example, Spark and Tupleware [35] integrate such operators into Scala, Java or Python, whereas DryadLINQ [36] extends LINQ, which is a set of constructs operating on datasets enabled in .NET languages. The second important aspect is that these programming models allow for arbitrary dataflows, which combine operators in any order and each logical plan can form a directed acyclic graph (DAG) instead of a sequence consisting of a single map and reduce function.

Programming models of this category also take iterative processes into consideration by allowing in memory processing. This enables the development of ML algorithms, whose optimization methods are mainly iterative processes. However, when it comes to distributed processing, supporting iteration can be quite challenging, as updates and propagation of global variables to every node of the cluster at the end of each iteration are not trivial. A few approaches are followed by the programming models of this category to address these issues. DryadLINQ has very limited support of shared state, as it allows read-only shared objects and computation results become undefined if any of them is modified. This is an important restriction of the DryadLINQ model with regard to ML algorithms, as they usually optimize a model by updating global parameters after each iteration. Spark and Tupleware use the concepts of Accumulator and Context respectively. Both objects can be updated only by associative and commutative operations. Spark also provides another type of shared variables, which are called Broadcast variables and are analogous to the read-only shared variables supported by DryadLINQ. Flink [37] (initially known as Stratosphere) has also Accumulators, which share similar features with those in Spark, but their partial results are only merged at the end of a Flink job. For computing simple statistics among iterations, Flink provides Aggregators, which can be used to check for convergence after each iteration step.

Although the design of MapReduce is not well-suited for iterative workloads [38], there has been effort on extending its original programming model to support loop constructs. Two of these efforts are HaLoop [39] and Twister

[40], which provide programming extensions to specify a termination condition and the input data of a loop, or define broadcast variables. To efficiently support these extensions, they developed mechanisms for caching loop-invariant data and scheduling map/reduce tasks that occur in different iterations but access the same data to run on the same machine, as well as a streaming-based runtime so that intermediate results between iterations are disseminated from their producing to their consuming map/reduce tasks without being written on disk. However, due to the wide adoption of Spark and Flink, which generalized in-memory processing for DAGs, these solutions did not prevail.

Regarding the data model, the core structures in this class of systems are immutable collections of records, which can be distributed and individually processed by a machine of the cluster. Data types of the elements in a record are based on the data types provided by the host language. Collections are also represented by objects of the host language. For example DryadLINQ datasets are managed via DryadTable objects in .NET, whereas Spark RDDs (Resilient Distributed Datasets) are Scala objects. Spark and Flink, which are still under active development, support higher level data representations too, such as SQL tables [41], [42] and dataframes [43]. In the following implementation of our running example in Spark we use RDDs to showcase the properties of the core data representation exposed to the users, as well as the expressivity capabilities of this class of frameworks.

In listing 3, training data are represented using a RDD of objects of class `LabeledPoint`. Each of these objects consists of a target value and a vector of features. First, we filter training observations based on the value of the third feature in each vector. Weights are also stored in a vector, which is transformed to a broadcast variable (read-only variable) in order to be available to every node of the cluster. Using a map operator, we then compute the error between the prediction and the target value and store it as an RDD of double values. A reduce function over the RDD of errors returns the total error. The gradient descent algorithm is also implemented in a similar manner. A map function computes the partial derivative of each error value, whereas the gradient is given by aggregating over all partial derivatives.

Listing 3: Linear Regression in Spark

```
import breeze.linalg.{DenseVector => BDV}
import breeze.linalg.{DenseMatrix => BDM}

def linearRegression (data: RDD[LabeledPoint], sc: SparkContext) {

        val trainingData = data.map { line =>
                val parts = line.split(',')
                val feature_vector = new Array[Double](parts.length
                    −1)
        for(i <− 0 to (parts.length −2)){
                feature_vector(i)=parts(i).toDouble
        }
        LabeledPoint(parts(parts.length−1).toDouble, Vectors.dense(
                feature_vector))

        }

        //Filtering observations
        val filteredTrainingData = trainingData.filter {point =>
                val features = point.features.toArray
                features(3) == 0
        }

        filteredTrainingData.cache();

        val weights = Vectors.zeros(13)

        val features = filteredTrainingData.map { point =>
                point.features
        }.cache()

        val numInstances = sc.broadcast(features.count())

        //Use of var to define a mutable reference to weights, as they
                are
        //reassigned after each iteration of gradient descent
        var broadcastWeights = sc.broadcast(BDV(weights.toArray))

        //Gradient descent
        for(i <− 1 to 200){
                val errors = filteredTrainingData.map { point =>
                        val features = BDV(point.features.toArray)
                        val features_transpose = features.t
                        val label = point.label
                        (label − (features dot broadcastWeights.value
                                )) * (label − (features dot
                                broadcastWeights.value))
                }
        }


        val totalError = (errors.reduce((a, b) => a+b))/(numInstances.
                value)

        val newWeights = computeGradients(filteredTrainingData,
                broadcastWeights, numInstances)
        broadcastWeights = sc.broadcast(BDV(newWeights.toArray))

}

def computeGradients (data: RDD[LabeledPoint], inputWeights:org.
        apache.spark.broadcast.Broadcast[breeze.linalg.DenseVector[
        Double]], numInstances:org.apache.spark.broadcast.Broadcast[
        Long]) : breeze.linalg.DenseVector[Double] = {

        val learning_rate = 0.0000001

        val gradients = data.map { point =>
                val features = BDV(point.features.toArray)
                val features_transpose = features.t
                val label = point.label
                −(2.0/numInstances.value)*(features * (label − (
                        features dot inputWeights.value)))
        }

        val totalGradient = gradients.reduce {case (a:(breeze.linalg.
                DenseVector[Double]), b:(breeze.linalg.DenseVector[
                Double])) =>
                BDV((a.toArray, b.toArray).zipped.map(_ + _))}

    val weights = (inputWeights.value − (learning_rate*totalGradient))
          .toDenseVector
        return(weights)
}
```

The same example can be also implemented with RDDs and array Scala objects without the use of the Breeze library [4]. Training data can be represented as a key-value RDD with key being the ID of an observation and value being an array consisting of the features and the label of an observation. Given such a representation, matrix-vector operations can be written using element-wise numeric operations, aggre-

4. https://github.com/scalanlp/breeze

gations and group by functions. For example, the computation of predictions in linear regression can be expressed as follows.

Listing 4: Computing predictions as a sum over feature-weight products

```
val predictions = filteredTrainingData.map {case (key, value) =>
        val w = broadcastWeights.value
        var sum=0.0
        for(i <− 0 to (numOfFeatures−1)){
          sum=sum+(value(i) * w(i))
        }
        (key, sum)
      }
```

Spark also supports a subset of linear algebra operations, but this support is currently incomplete and fragmented. Matrices and vectors are not exposed as data abstractions, but they rather depend on specific physical implementations. For example, distributed matrices are supported in four different formats, RowMatrix, IndexedRowMatrix, CoordinateMatrix and BlockMatrix. The details of these formats may not be intuitive to ML experts, who just need a logical abstraction of a matrix/vector to express linear algebra computations in their algorithms. Another important point is that the provided API does not include the same functionality among the different formats. For example, one can use element-wise addition and subtraction only in the BlockMatrix format. Such common operations are not supported in the other three distributed matrix implementations. These differences may make users to choose types based not only on format characteristics, but also on the available functionality. In the following code, we can observe such conversions from IndexedRowMatrix objects to BlockMatrix objects, in order to use subtractions.

Listing 5: Computing errors with Spark's primitives for linear algebra

```
// Assuming features and labels are RDDs of IndexedRow objects
      initialized earlier

val featuresMatrix = new IndexedRowMatrix(features)
val labelsMatrix = new IndexedRowMatrix(labels)
val weightsMatrix = Matrices.dense(1, numOfFeatures, Array.fill[Double
      ](numOfFeatures)(1.0))

val weightsMatrix_t = weightsMatrix.transpose
val predictions = featuresMatrix.multiply(weightsMatrix_t)
val predictions_block = predictions.toBlockMatrix()
val labels_block = labelsMatrix.toBlockMatrix()
val erros = labels_block.subtract(predictions_block)
```

Furthermore, summing over rows of a matrix, which is needed to compute the total error in LR, is not supported in any of the formats. One would convert a matrix to an RDD and implement the computation there. Hence, despite the combination of RDDs with the Breeze numerical processing library in listing 3 leads in conversion between Spark's and Breeze vectors, this issue also occurs between different types of matrices and RDDs when using Spark's built-in API.

At the declarativity front, systems in this category support three out of seven properties. Data processing and at least a subset of linear algebra operators is supported. DAGs of operators are also optimized based on a number of techniques. Regarding the rest of the properties, data abstractions are not fully independent, as the user has access to caching and distribution properties of the data. For example, in listing 3 we use the `cache` function to cache the filtered training observations and the `broadcast` function to define that the weights are broadcasted to every node. Hence, the user is required to have an understanding about the physical implementation of the data structures used. This shortcoming have also been recognized by the developers of Emma [44], a language for parallel data analysis that is embedded in Scala. Emma serves as a layer above systems, such as Spark and Flink, and provides a ubiquitous bag abstraction, which hides the low level details of the individual data structures in distributed execution engines. Moreover, because in this class of systems operators are essentially second-order functions, code that is given as argument to the functions is written in existing imperative or functional languages, which makes it quite challenging to optimize. In the following sections, we describe how Flink and Tupleware attempt to overcome this problem by doing static analysis of the code and applying optimizations from the domain of compilers. Also the integration of operators into imperative or functional languages exposes control flow constructs to the user. Finally, the algorithm that computes the parameters of a machine learning model is coded by the user and no out of the box solvers, like gradient descent, are provided.

4.4 Systems Targeted to Machine Learning

Machine learning algorithms usually involve linear algebra operations, probability distributions and computation of derivatives. Based on these characteristics, systems in this category provide domain specific languages (DSLs) and APIs that support data structures for matrices and vectors, as well as operations on them, probability distribution functions, and automatic differentation. Some representative systems of this class are SystemML [45], Mahout Samsara [29], TensorFlow [9], PyTorch [10], BUDS [46], MLI API [47] and Lara [48]. Not every one of them supports all of the aforementioned features and degree of declarativity also differs. We emphasize such differences using the example of linear regression written in DML, the declarative language used in SystemML.

In listing 6, almost every computation is expressed as a linear algebra operation between matrices and vectors. SystemML has more recently added frames, a data structure for tabular data with limited support for transformations, which however does not include filtering of rows. Most of the systems in this category does not provide any structure for relational or tabular data as well. Due to this, we omit filtering of training observations in listing 6 and load features and labels directly to matrices. We also create a one-column matrix for storing weights and initialize it with zeros. To implement gradient descent we use a loop, where we compute predictions and gradients, as well as update `weights` and `total_error` in each iteration.

Listing 6: Linear Regression in DML, one of the languages supported by SystemML

```
features = data[,1:(ncol(data)−1)];
labels=data[,ncol(data)];
num_of_observations = nrow(labels);
```

```
learning_rate=0.0000001;
iterations = 200;
# Initialize weights to zero
weights = matrix(0,rows = ncol(features),cols=1);

total_error = matrix(0,rows = iterations,cols=1);

# Gradient descent
predictions = features%*%weights;
error = predictions − labels;
for( i in 1:iterations){
        gradients = (t(features)%*%error)/num_of_observations;
        weights = weights − learning_rate*rowSums(gradients);
        predictions = features%*%weights;
        error = predictions − labels;
        total_error[i,1] = t(error)%*%error;
        total_error[i,1] = total_error[i,1]/(2*num_of_observations);
}
```

Iteration is supported via the usual constructs "for" and "while", which make it possible to express the entire algorithm in DML, with no need for driver programs in other languages. In addition to this, linear algebra operators and matrices provide a more intuitive way to express computations and keep the code succinct. Similar code could be written in the rest of the systems in this category but with a few differences hidden in the details.

Mahout Samsara, which is a Scala DSL for linear algebra baring a R-like look and feel, allows users to decide about specific representations of data abstractions and data flow properties. For example, the user can choose between local and distributed or dense and sparse matrices, as well as specify caching and partitioning of the data. These options diverge from the notion of declarativity. Tensorflow and PyTorch APIs provide the expressivity of SystemML, but the user can also leverage automatic generation of gradients and avoid writing code for this computation. Regarding declarativity, Tensorflow and Pytorch also expose details of physical data representation to the user with sparse and dense tensors. Lara and MLI API support both collections and matrices, as well as common operations on them. This is an interesting approach, because they combine data processing and machine learning. However, MLI API is no longer under active development and the authors claim that many of the key ideas have been integrated into Spark MLlib and Keystone ML [49]. Last but not least, BUDS is fundamentally declarative in the sense that each computation is a list of dependencies among variables and variable updates happen recursively according to those dependencies. There are no control flow constructs and iteration is expressed via recursion. The "for" construct serves for parallel execution of variable expressions, instead of typical looping.

As far as declarativity properties are concerned, except from the research prototypes of Lara and MLI API, systems in this class focus on supporting ML-centric operators, such as linear algebra and distribution functions, and lack data processing operators. The main data abstractions are matrices and vectors, but in some of the systems they are not independent from their physical implementation. Also at least some basic plan optimization happens in most of the frameworks. As we will describe in Section 5, some of the languages presented here are accompanied by optimizers developed specifically for them, while others are translated to existing languages and rely on their optimizers or on optimizations performed by the execution engine. Regarding iterative processes, we see that some systems provide loop constructs, while others, e.g. TensorFlow, support automatic differentiation and mathematical solvers for encapsulating the training. Finally, the set of supported operators in this category does not heavily depend on UDFs.

4.5 Integration of machine learning to databases

In this section we study approaches that attempt to integrate the development of ML algorithms with a database system in a different manner from providing a library of algorithms as UDFs to the user. Approaches in this class follow three directions: array databases, mathematical optimization on relational data and extension of the SQL language with linear algebra data types and operations.

4.5.1 Array databases

Scientific data, such as astronomical images or DNA sequences, are commonly represented as arrays. Array databases propose a data model that fits ordered collections of data better than the relational model. In addition to this, arrays can be used to represent matrices and vectors widely involved in ML algorithms.

A prevalent system in this category, SciDB [50] uses the following data model: an array that consists of dimensions and attributes. Dimensions serve as indices/coordinates of each cell, whereas attributes are the actual contents of a cell. Cells may either contain tuples of attributes or may be empty (null), so that both dense and sparse arrays can be represented. Each attribute is of a primitive type (int, float, char). SciDB also supports the definition of complex types by the user in a similar way as PostgreSQL [5].

SciDB offers a functional language and a query language with similar syntax to SQL. It provides common operations on array data. Those include slicing an array along a dimension, subsampling a region of an array, filtering attribute values in an array, applying a computation on the cells of an array and combining cells from two arrays. Basic linear algebra operators, such as matrix multiplication and transpose, are also supported. The implementation of Linear Regression using SciDB in listing 7 gives more details about its programming model.

Listing 7: Linear Regression in SciDB

```
−− Loading data to arrays
create array features <instance_id:int64, feature_id:int64, val:double
    >[i=0:6577];
load(features, '/home/hduser/features.csv',−2,'CSV');
store(redimension(features, <val:double>[instance_id=0:505:0:1000;
    feature_id=0:12:0:1000]), features_2d);

create array labels <instance_id:int64, val:double>[i=0:505];
load(labels, '/home/hduser/labels.csv',−2,'CSV');
store(redimension(labels, <val:double>[instance_id=0:505:0:32, j
    =0:0]), labels_2d);

create array weights <feature_id:int64, val:double>[i=0:12];
load(weights, '/home/hduser/weights.csv',−2,'CSV');
store(redimension(weights, <val:double>[feature_id=0:12:0:1000, j
    =0:0:0:1000]), weights_2d);
```

5. https://www.postgresql.org/

```
−− Multiplying feature and weight arrays to produce predictions
store(build(<val:double>[row=0:505:0:1000; col=0:0:0:1000],0),C1);
store(gemm(features_2d, weights_2d,C1), predictions);

−− Compute errors between predictions and actual values
store(project(apply(join(predictions,labels_2d), e, predictions.gemm −
        labels_2d.val), e), errors);
store(build(<val:double>[row=0:12; col=0:0],0), C);

−− Compute gradients
store(gemm(transpose(features_2d), errors ,C), gradients);
store(apply(gradients, d, gradients.gemm/506), gradients_2);
aggregate(gradients_2,sum(d),col);

−− Compute new values for weights
store(apply(cross_join(weights_2d, aggregate(gradients_2, avg(d) as
        m)), val2, weights_2d.val−0.00001*m), new_weights);
store(redimension(new_weights, <val2:double>[feature_id=0:12]),
        new_weights_2);
```

The first query creates a one-dimensional array, where each cell is a tuple consisting of a feature id and a feature value. Feature values from a csv file are loaded to this array using the `load` command. We then move on to the implementation of the Linear Regression model, without filtering out training observations. SciDB supports filtering of cells based on boolean expressions by emptying their content, e.g. `filter(A,cell_value<100)`, but the dimensions of the array remain the same. Filtering of a matrix by dropping rows, as we need for observations in our example, or emptying all of their cells based on a condition is not supported on the specific system. This is justified by the fact that SciDB targets ordered data, such as images, where rows might be related to each other.

Back to the code in listing 7, in order to be able to use linear algebra operators for matrix multiplication or transpose, we need to transform the initial array to a new one that is two-dimensional and where each feature value is stored in a different cell as a tuple with a single attribute, whereas feature id has become the second dimension. The same process is also followed for creating arrays for weight values and labels. When creating an array, the user can also specify chunk size, which determines the maximum number of cells in each chunk. Although SciDB can automatically choose a chunk length based on schema, in the presented implementation of LR `gemm` operator complained about the chosen chunk size being small. This type of properties does not concern the logical structure of a data analysis problem and tend to mix logical with physical implementation, which contradicts the purpose of declarative programming.

The rest of the queries implement the steps of a gradient descent iteration, starting with multiplying weight and feature matrices to compute predictions using SciDB's function `gemm`. Despite matrix multiplication and transpose are supported by specific operators/functions, element-wise addition/subtraction/multiplication are implemented by joining tables and applying a transformation on their cells as it is shown in the computation of array `errors`. So, regarding linear algebra the design of SciDB follows two strategies: complicated linear algebra operators are provided by the database, whereas simpler ones are implemented/emulated using array-based operators. Code below implements a single iteration of gradient descent, as there are no constructs to express iterative processes in SciDB. Similarly to Pig Latin, SciDB operators can be embedded to an imperative language or use a driver program to execute SciDB queries multiple times, but of course this is not optimal performance-wise and it also mixes declarative with imperative programming.

For reasons of completeness, we also mention TileDB [51], which is another system relevant to array data management. It supports the same data model as SciDB, but is currently more of a storage manager for array data rather than an array database. Its storage manager module is accessed via a low level C API, which includes functions for initializing and finalizing an array (freeing its memory), loading and retrieving the schema of an array, reading and writing to an array, iterating over its elements, as well as synchronizing between files and merging of array updates.

Array databases have been successfully used for data science tasks on scientific data, as those are naturally represented with arrays. However, iterative processes in machine learning, such as gradient descent, cannot be natively supported using the primitives of these languages. Although workarounds in other languages can be developed for this and an extension to the original model [52] has been proposed, iteration is still not integrated into the SciDB query language and as a result optimized by a query optimizer. Systems in this category support five out of seven properties. They support data processing operators on arrays and a subset of linear algebra operators. Their SQL-like query language lacks control flow structures. Semantics for both types of operators are well-defined and do not depend on user defined functions. Also, despite not as advanced as in relational databases, some query optimization techniques are implemented in SciDB. However, data abstractions are not fully independent, since the user is exposed to array characteristics, such as chunk size. Finally, algorithms for automatic computation of the parameters of a model are not provided.

For relational data to be ported to the array data model, denormalization is necessary. In the next sections, we analyse other approaches for integrating ML into relational database systems that model data science tasks on relational data.

4.5.2 Mathematical Optimization on Relational Data

Mathematical optimization queries regard the expression of mathematical optimization problems on relational data. Relational databases are extensively used. The ability to integrate the expression of mathematical optimization problems with a database offers great value, as it would eliminate the need to manually export and import data to different systems. As many ML algorithms are deep down numerical optimization problems, modeling such problems inside a database would also allow modeling ML tasks.

One such system that allows modeling mathematical optimization problems on relational data is the LogicBlox database [53], [54]. The LogicBlox database supports the expression of linear programs using LogiQL, an extended variant of Datalog. The user needs only to define the objective function, the constraints and any other business logic in LogiQL. Then, a rewriting process, which is called grounding, transforms the relational form of the convex optimization program to a matrix format, that is consumed by an external solver. Grounding involves the automatic

creation of predicates that represent the LP instance in its canonical form, i.e. the matrix A and vectors c and b in $max\,\{c^Tx|x{\geq}0, Ax{\leq}b\}$, as well as the rules that populate these predicates during runtime. As soon as the data are marshaled to the data structures supported by the solver, the optimization process of the problem begins. The solver responds with the computed solution, which is finally stored back in the database and can be accessed via typical LogiQL queries.

The LogiQL code in listing 8 implements LR as a linear program [55], whose objective function is Least Absolute Error (see equation 4)

$$\min \sum_{i=1}^{n} |y_i - wx_i|,\ n = number\ of\ training\ observations \tag{4}$$

We assume that the type and arity of each predicate are defined earlier in the program and that the extensional database (EDB) predicates, i.e. `observation`, `feature_value` and `target`, are populated by user's data. The code snippet begins with defining the rules that populate the intensional database (IDB), i.e. the predicates that are not explicitly imported by the user. The predicate `totalError` is the objective function, which is denoted using the annotation `lang:solver:minimal(totalError)`. The absolute error between a target value and a prediction is expressed using two linear constraints, which state that the absolute error is greater or equal than both the value of the error and the negative value of the error. The weights of the model, i.e. the variables of the linear program, are defined using the annotation `lang:solver:variable`.

Listing 8: Linear Regression in the LogicBlox database using LogiQL

```
//Weight is a LP variable
lang:solver:variable(`weight).

//Abserror is a LP variable
lang:solver:variable(`abserror).

//Data filtering
filtered_observation(i) <- observation(i), f="CHAR", feature_value[f, i
    ]=v, v=0.0f.

//IDB rule to generate predictions
prediction[i] = v <- agg <<v=total(z)>> filtered_observation(i),
    feature_value[f, i]=v1, weight[f]=v2, z=v1*v2.

//IDB rule to generate error between predicted and actual target values
error[i] = z <- prediction[i] = v1, target_actual[i] = v2, z = v1-v2.

//IDB rule to generate the sum of all errors
totalError[] = v <- agg << v=total(z) >> abserror[i]=z1, z=z1.
lang:solver:minimal(`totalError).

//Constraints to implement that abserror is the absolute value of error
filtered_observation(i), abserror[i]=v1, error[i]=v2 -> v1>=v2.
filtered_observation(i), abserror[i]=v1, error[i]=v2, w=0.0f-v2 -> v1>=
    w.
```

MLog [56] has a similar design to LogicBlox's framework. It offers a declarative language, whose data model is based on tensors. The user can create a tensor using a CREATE command, similar to SQL, and manipulate the tensor with a set of operations, including slicing and linear algebra operations. Each MLog program consists of a set of rules, which create TensorViews based on expressions involving operations over tensors. The concept is similar to relational views, which are the result of a query or intensional database rules in LogiQL / Datalog. The user can also run MLog queries, which correspond to mathematical optimization problems. These queries find the optimal instances for a number of tensors that minimize or maximize a TensorView defined earlier in the program. This is similar to the LogiQL annotation "lang:solver:minimal" that accompanies the predicate of the objective function. The solution to the MLog queries is computed by TensorFlow, which is also an external system to the database. MLog programs are automatically translated to Datalog programs by representing tensors as a special relation type. Based on this representation the authors claim that the MLog language can in principle be integrated with SQL, although they have not implemented this yet. Finally, Datalog programs are subsequently translated to TensorFlow code.

To summarize, systems in this category follow the model+solver paradigm, where the user defines the logical structure of the model, whose solution is delegated to a black box system. Regarding data abstractions, they use relations or attempt to integrate tensors with relations. As a consequence, they provide relational operators and either support natively or emulate linear algebra operators wherever it is possible. Plan optimization is implemented via standard techniques for SQL or Datalog queries. Moreover, because the supported languages are purely declarative, the user does not have control over the execution order of the commands inside a program. However, support for iteration is not necessary using the approach of this category, as the computation of the solution which is iterative is not implemented by the user, rather it is provided by the system. Finally, operators do not serve as second-order functions. Although code in imperative languages is supported via UDFs, their use is not as heavy as we see in the family of MapReduce extensions. Overall, six out of seven declarativity properties are covered by frameworks of this class.

4.5.3 In-database Linear Algebra

Following the same direction of modeling ML algorithms inside the database, a very recent approach [57] extends the relational model with three data types, matrix, vector and labelled scalar, as well as a set of operations over the aforementioned types. These types can be used for attributes when creating a table, i.e. a column in a table can store a matrix or a vector. Moreover, within this approach, common linear algebra operations are integrated into the SQL language. The user can perform matrix-matrix or matrix-vector multiplications as part of a query or use standard SQL aggregations on the elements of a vector. The new set of operations also includes functions for composing instances of the new data types, for example the function VECTORIZE creates a vector from a collection of labelled scalars and ROWMATRIX / COLMATRIX create a matrix from a set of vectors.

Both this approach and the systems for mathematical optimization in previous section share similar benefits. Moreover, they could also be combined and instead of having a new declarative language based on tensors or emulating linear algebra operators using relational operators, one could

implement the described extensions in the relational model, but still use a black-box solver to provide the solution.

This approach also scores high in declarativity as it extends an already declarative language, i.e. SQL, to support mode advanced analytics operators from linear algebra. The only property that is not covered is the automatic computation of the model parameters, which still needs to be written by the user in terms of the extended version of SQL and whose iterative nature is not well-suited to data querying languages.

4.6 Declarativity and Calculus

In previous sections, we showcased through the example of LR that users often need to implement challenging mechanics of ML algorithms. A common one is the definition of gradients. Apart from TensorFlow and systems in section 4.5.2, in the rest of the surveyed frameworks the user needs to define the gradients of the objective function manually, which requires a certain familiarity with differentiation in mathematics. A key component to automate this part that is currently missing from systems for declarative machine learning is either symbolic [58] or automatic differentiation [59]. Implementations of symbolic differentiation in computing environments like MATLAB [60] and Mathematica [61], and automatic differentiation in TensorFlow and a few frameworks for deep learning indicate the usefulness of this feature in the ML toolkit.

Despite the difference in the underlying mechanics and the output of these two differentiation methods, the programming interface to the user is pretty much the same in both approaches. The user calls a method to which it gives as input a numerical function. By combining techniques for computation of derivatives with mathematical optimization algorithms, systems for declarative machine learning can provide automatic solution of a large class of objective functions.

5 Optimization Techniques

Program/plan optimization is an important part of the declarative paradigm. Since the user defines the result and not how this is computed, the system can provide a number of possible implementations and choose one that is good enough, if not optimal, in terms of efficiency. Considering the systems presented in the previous section, libraries of algorithms are used as black boxes and may involve hard-coded optimizations. Nevertheless, the other categories use either rule or cost-based approaches and borrow their ideas mainly from two areas: database systems and compilers.

The purpose of this section is to cover optimizations that transform the structure or the implementation of a program. Parallelization, scheduling and low-level optimizations regarding write operations, unless specifically relevant to the nature of data science tasks, are out of scope. Also, systems that leverage optimizers of existing languages, such as BUDS which translates its programs to SimSQL [62], are not covered in these sections.

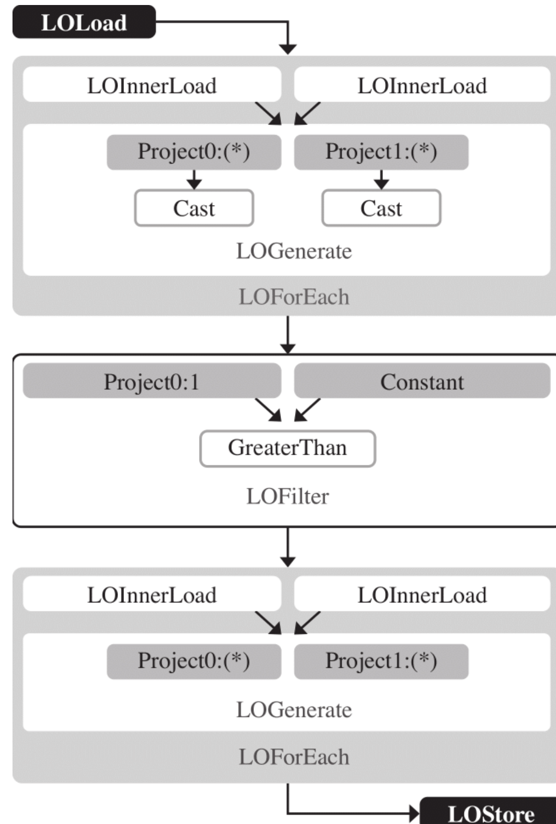

Fig. 1: Logical plan of the program in listing 9 generated by Pig

5.1 Rule-based optimization

This type of optimizations is based on rules. Every time certain conditions are met, these rules are triggered and perform specific transformations on the code. Languages that support relational operators, such as Pig Latin, U-SQL and Spark's SQL API [63], apply typical optimizations of database systems, e.g pushing filters closer to the data sources. In the following example in Pig Latin, we can notice that in the generated logical plan of Fig. 1 the filtering operation is performed before the `FOREACH` operator, whereas in the user code they are written in reversed order. Another optimization that takes place in the logical plan is the pruning of unused columns. The code states that four columns of the input data are loaded. However, as only the first two are used by the rest of the program, the optimizer transforms the logical plan so that only those two are eventually loaded.

Listing 9: A simple program in Pig Latin

```
data = LOAD 'pig/data.csv' USING PigStorage(',') AS (d0:double, d1:
        double, d2:double, d3:double);
data_projected = FOREACH data GENERATE $0, $1;
data_filtered = FILTER data_projected BY ($1)>5;
DUMP data_filtered;
STORE data_filtered INTO 'pig/data_filtered.csv' USING PigStorage(',')
        ;
```

Other optimizations include pushing `LIMIT` operators near data, which is also done with `FOREACH` when it is combined with `FLATTEN` operators. The main idea of these techniques is to decrease the number of tuples to be processed by

expensive operations, such as join. So, as `FLATTEN` unnests data and may increase the number of generated tuples, it is advised to be moved after the `FOREACH` operation. Finally, pipelining of operations on data is also applied when it is possible. For example two consecutive `FOREACH` statements can be merged, in order to avoid a second pass on data. The capability to pipeline operations and avoid materialization of intermediate results is possible due to the lazy evaluation of programs that is supported by many of the described systems, such as Pig Latin, Jaql and Spark. This means that logical plans represent just a sequence of steps and are not physically executed until an output operation is requested, e.g. a print or store command, allowing the program to be optimized as a whole. More sophisticated optimizations, such as reordering of joins, are not easily achieved with rule-based methods and depend on the computation of some cost metric, which is the topic of the next section.

Rule-based rewrites are also used by systems that provide linear algebra operators. For example, SystemML generates High-level Operator (HOP) plans that represent the data flow between linear algebra operators, such as cell-wise multiplication, matrix multiplication or matrix transpose. Those plans are similar to the logical plans used in databases. An example of a HOP DAG, which corresponds to the code snippet in listing 10 is displayed in listing 11.

Listing 10: Code sample in DML

```
gradients = (t(features)%*%error)/m;
```

At the moment, SystemML does not provide a visualization tool for plans, but we can analyse this representation by identifying a few key components. The first number of each line is the HOP id and right next to it there is the operation code. Operation codes are divided into six categories: binary, unary, aggregate unary, aggregate binary, reorganize and data. So, b(-) is a binary operator that performs a subtraction between two cells of a matrix, or ba(+*) is an aggregate binary operator that first multiplies cells of a matrix and then sums the individual results. Numbers in parenthesis explain data dependencies between HOPs. For example, HOP 63 depends on the output of HOP 55. The first pair of brackets include characteristics of the output matrix, e.g. number of rows and columns, whereas the second pair of brackets provide memory estimates for input data, intermediate results and output matrices. Finally, CP (Single Node Control Program), SP (Spark) and MR (MapReduce) denote the execution type.

Listing 11: HOP plan generated by SystemML for code in listing 10

```
(55) TRead features [506,13,−1,−1,−1] [0,0,0 −> 0MB], CP
(63) r(t) (55) [13,506,−1,−1,−1] [0,0,0 −> 0MB]
(58) TRead error [506,1,−1,−1,−1] [0,0,0 −> 0MB], CP
(64) ba(+*) (63,58) [13,1,−1,−1,−1] [0,0,0 −> 0MB], CP
(66) b(/) (64) [13,1,−1,−1,−1] [0,0,0 −> 0MB], CP
```

This particular HOP plan begins with reading the features that will be used to compute gradients. The computation of gradients starts at HOP 63, which outputs the transpose of matrix `features`. The result is read by the operator with id 64, which multiplies it with the `error` vector. The next operator divides the product with scalar `m`. The size of the matrices is recorded in the first pair of brackets. For example, features is a 506x13 matrix, but after applying a transpose operator it becomes a 13x506 matrix. It is important to note that HOPs are logical operators, placeholders in a sense, that will be replaced by specific implementations during the Low-level Operator (LOP) planning phase.

SystemML begins optimizing a HOP DAG by applying a number of static algebraic simplifications and removal of branches. These algebraic simplifications regard cost-independent rewrites that are always beneficial no matter the dimensions of the matrices/vectors involved. For example, operations with one or zero, i.e $X/1$ or $X - 0$, leave matrices unchanged and can be replaced by the matrix itself (X). Another example is the use of binary operations that can in fact be transformed to unary operations, such as X+X which can be rewritten to $2 * X$. Furthermore, if-else blocks that depend on constant conditions are replaced with the body of the corresponding branch. This rewrite makes propagation of unconditional matrix sizes easier, which is important for applying cost-based optimization techniques in a latter phase. Mahout Samsara's optimizer applies similar rule-based rewritings on the logical plan. The choice of physical operators is also based on heuristics regarding partitioning and key values, as well as types of distributed row matrices. However, overall Samsara provides a less advanced optimizer than SystemML, without support for cost-based optimization. In addition to this, the optimization methods that are used are much less documented and inspection of the source code seems to be the only available source of information at the moment.

In the intersection of machine learning and database systems, a novel research area that factorizes linear algebra computations into a series of relational operators has emerged [64], [65], [66]. These rewrites aim at reducing computational redundancy caused by joins. For example, when working on relational data, features are usually split across different tables, let us say S and R. Hence, the product $w^T x$ between features and weights can be decomposed to inner products over the base tables S and R. This decomposition reduces the redundancy that is caused by computing the inner product when the same tuple from table R is joined with multiple tuples from table S. The partial inner products from R can be stored in a relation and reused for every joined tuple in S. Automating such rewrites remains a challenge, though. Recent work [67] proposes heuristic rules for automatically translating a set of linear algebra operators, which are common in ML, into operations over normalized data. These rules are based on two simple metrics, whose role is important on the speedups yielded by the rewritings: tuple ratio (#tuples of table1/#tuples of table2) and feature ratio (#columns of table1/#columns of table2) of two joined tables. Whenever tuple ratio and feature ratio are below an experimentally tuned threshold, rewriting rules are not fired. Thresholds are conservative in the sense that they may wrongly disallow rewritings that could improve runtime.

Apart from the rule-based rewritings described above, there are also a few simple rules that come from the domain of compilers. These include techniques like function and variable inlining, constant folding, common subexpression elimination and SIMD vectorization. Jaql, SystemML, Tupleware, Spark and TensorFlow already exploit ideas from this area. Function inlining replaces a function call with

the code of the function, whereas parameters become local variables. Similarly in variable inlining when a variable is used in an expression, it is replaced by its definition, which is either an expression or a value. Inlining reduces the overhead caused by functions calls and variable references, but at the same time increases memory cost and is usually avoided for large functions. Constant folding recognizes expressions over constants and evaluates them at compile time rather than at runtime. Common subexpression elimination aims at computing subexpressions that are repeated inside code once, then all of its redundant appearances would be removed. For example, TensorFlow accumulates repeated computations to a single node of the graph, which gets connected with every other node that needs this computation. Finally, SIMD vectorization achieves data level parallelism by using the registers of a processor to apply a single instruction on multiple data simultaneously. This technique has been recently added to Spark through the Tungsten project [6]. Tupleware also applies SIMD vectorization on UDFs' code, which is described in more detail in section 5.3.

Finally, at the physical level many of the systems for large-scale analytics need rules to translate programs written in higher-level languages into the programming model of the preferred execution engine efficiently. For example, Jaql and Pig that translate their programs to the MapReduce model, need to create as few MapReduce jobs as possible in order to avoid materialization of data between jobs. To do so, they identify steps of the logical plan which are mappable, i.e. can be executed independently over partitions of data (for example `FILTER` and `FOREACH` operators), and group them into a single Map function. Expressions that are encountered after a group by operator are executed inside a Reduce function. They also employ ways to translate group operators with multiple inputs, such as `COGROUP`, either by creating separate map functions for each input and aggregating them in a single reduce, or by adding a field which registers each tuple to the dataset it belongs to. The MapReduce plan for the Pig Latin program in listing 9 is displayed in Fig. 2. We can see that all operations are grouped inside a single map function, as Pig takes advantage of the fact that `FILTER` and `FOREACH` operations can be run in parallel over partitions of data. The gain from such optimizations depends highly on the properties of the execution engine that each system uses. For example, if we switch from Hadoop to Spark we can avoid many of the drawbacks that come with materialization of data between jobs.

In a similar way of thinking, SciDB's optimization at the physical level focuses on minimization of data movement and efficient parallelization. It follows an adaptive approach by identifying subtrees of the physical plan where operators can be pipelined and as a consequence parallelized over a cluster of machines. For the rest of the plan that cannot be parallelized, a scatter/gather operator is used, which gathers the local chunks of the nodes in a buffer and pushes it to the node where the data need to be transferred. MLog also employs a textbook static analysis technique, called the pivoting method [68], to recognize batching opportunities on the Datalog programs, which have been generated from translating the initial MLog user programs. The end goal is again to achieve parallelization by identifying components in a Datalog program that can be evaluated independently. After discovering such components, MLog expands every tensor in the generated TensorFlow code by an extra dimension denoting its batch. Probably due to space limitations, the paper includes limited details of this optimization technique and no examples of the final generated TensorFlow program are provided.

6. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

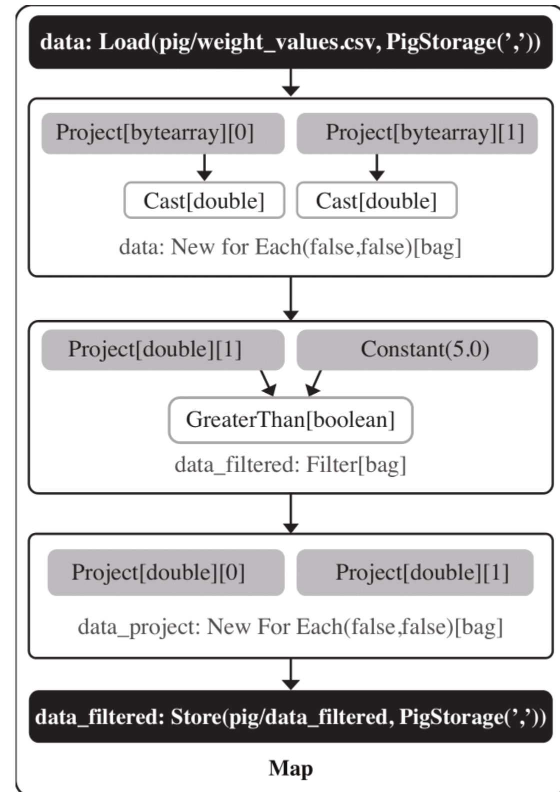

Fig. 2: MapReduce plan of the program in listing 9 generated by Pig

In order to avoid unnecessary operations in parallel query plans, U-SQL integrates reasoning about data partitioning into its optimizer [69], It derives the partitioning properties of physical operators and their operands in a plan and tries different partitioning schemes that satisfy these requirements. The authors define a data exchange operator which repartitions data, i.e. in its physical implementation consists of a partition and/or merge step. Each partitioning scheme is implemented via this operator and is determined using a set of rules generating a valid physical query plan. For example, if the result of a filter operator needs to be partitioned over a set of columns, the optimizer can add a partition operator before the filtering or propagate partitioning to its operands. The optimizer evaluates all generated equivalent plans and chooses the best based on estimated costs as it is typically done. In a similar manner, Lara exploits access patterns, e.g. column-wise or row-wise data layouts, to choose physical implementations for linear algebra operators.

5.2 Cost-based optimization

Another type of optimization techniques is based on the computation of cost metrics. That means that different rewritings of a plan are evaluated using a cost model and it depends on the evaluation which of the rewritings will take place. On the contrary, rule-based optimization always triggers a rule, when conditions apply, even if the rewriting does not improve the generated plan. Systems, such as Tupleware and Spark, support cost-based optimization on relational operators either at logical or at physical level, using well-studied ideas from database optimization. The cost model for relational operators is based on statistics of tables that are collected periodically, e.g. number of tuples, selectivity of columns. For example, both Tupleware and Spark's SQL Catalyst optimizer perform join reordering based on data statistics, whereas the latter also chooses implementations for join operators at the physical level by estimating I/O operations.

A more novel area of cost-based optimization addresses linear algebra operations. SystemML seems to be the most advanced system in this direction. Cost-dependent rewrites are applied on expensive operations of the logical plan, i.e. HOP DAG, and the cost is based on a number of metrics, including matrix dimensions, floating point operations, I/O operations and shuffle cost, depending on each optimization phase. Some of the first rewritings of a DAG consisting of linear algebra operators include the removal of operators when one of the matrices contains only zeros, the replacement of row and column sums with table sums whenever the matrix consists of a single row or column, or the removal of indexing whenever the dimensions of the matrices are the same. These transformations are size-dependent, as the optimizer needs to be aware of the dimensions and the sparsity of the input matrices in order to perform them. That being said, there is still no cost function that is minimized/-maximized on this phase. We can observe some of these rewritings in the HOP plan of the code snippet in listing 12.

Listing 12: Code sample in DML

```
gradients = (t(features)%*%error)/m;
weights = weights − learning_rate*rowSums(gradients);
predictions = features%*%weights;
error = predictions − labels;
total_error[i,1] = t(error)%*%error;
```

The generated plan for this part is in listing 13.

Listing 13: HOP plan generated by SystemML for code in listing 12

```
(171) TRead weights [13,1,1000,1000,−1] [0,0,0 −> 0MB], CP
(167) TRead features [506,13,1000,1000,−1] [0,0,0 −> 0MB], CP
(175) r(t) (167) [13,506,1000,1000,−1] [0,0,0 −> 0MB]
(170) TRead error [506,1,1000,1000,−1] [0,0,0 −> 0MB], CP
(176) ba(+*) (175,170) [13,1,1000,1000,−1] [0,0,0 −> 0MB], CP
(178) b(/) (176) [13,1,1000,1000,−1] [0,0,0 −> 0MB], CP
(238) t(−*) (171,178) [13,1,1000,1000,−1] [0,0,0 −> 0MB], CP
(183) TWrite weights (238) [13,1,1000,1000,−1] [0,0,0 −> 0MB], CP
(184) ba(+*) (167,238) [506,1,1000,1000,−1] [0,0,0 −> 0MB], CP
(174) TRead labels [506,1,1000,1000,−1] [0,0,0 −> 0MB], CP
(185) b(−) (184,174) [506,1,1000,1000,−1] [0,0,0 −> 0MB], CP
(186) TWrite error (185) [506,1,1000,1000,−1] [0,0,0 −> 0MB], CP
(168) TRead error_history [200,1,1000,1000,−1] [0,0,0 −> 0MB], CP
(189) r(t) (185) [1,506,1000,1000,−1] [0,0,0 −> 0MB]
(190) ba(+*) (189,185) [1,1,1000,1000,−1] [0,0,0 −> 0MB], CP
```

In listing 13 after the computation of vector `gradients` at HOP 178, we can observe that the unary aggregation of row sums of `gradients` is removed, as the number of columns is equal to one and this makes it the same with multiplying with the single cell of each row. So, the expression

```
weights = weights − learning\_rate*rowSums(gradients)
```

is transformed to just

```
weights = weights−learning\_rate*gradients
```

Another example is the expression in listing 14.

Listing 14: Code sample in DML

```
error = colSums(features%*%weights − labels)/nrow(features);
```

We can observe in the HOP plan of listing 15, that the optimizer has pushed a summation over columns of `features` (HOP id 128) before computing the product with `weights`, in order to reduce the size of the multiplication.

Listing 15: HOP plan generated by SystemML for code in listing 14

```
(92) TRead features [506,13,−1,−1,−1] [0,0,0 −> 0MB], CP
(128) ua(+C) (92) [1,13,−1,−1,−1] [0,0,0 −> 0MB], CP
(93) TRead weights [13,1,−1,−1,−1] [0,0,0 −> 0MB], CP
(133) ba(+*) (128,93) [1,1,−1,−1,−1] [0,0,0 −> 0MB], CP
(134) u(cast_as_scalar) (133) [0,0,0,0,−1] [0,0,0 −> 0MB]
(94) TRead labels [506,1,−1,−1,−1] [0,0,0 −> 0MB], CP
(126) ua(+RC) (94) [−1,−1,−1,−1,−1] [0,0,0 −> 0MB], CP
(127) b(−) (134,126) [0,0,−1,−1,−1] [0,0,0 −> 0MB], CP
(124) u(cast_as_matrix) (127) [1,1,1000,1000,−1] [0,0,0 −> 0MB]
(103) b(/) (124) [1,1,−1,−1,−1] [0,0,0 −> 0MB], CP
```

SystemML's optimizer examines the dimensions of the matrices involved and notes that the initial code will multiply two matrices of size (506x13) and (13x1), whereas pushing the summation first will result in a multiplication of size (1x13)*(13x1). The same optimization is also applied on `labels` by fully aggregating over both rows and columns (HOP id 126), which results in executing a subtraction between scalars instead of matrices. After these two rewritings, the initial expression is finally transformed to

```
error = ((colSums(features)\%*\%weights) − sum(label))/nrow(
        features)
```

SystemML's optimizer is able to propagate matrix sizes to the whole program, using a bottom-up procedural analysis inside and across DAGs. Starting from read operators, for which input sizes can be inferred or are known due to metadata, dimensions and sparsity properties are propagated to the operators of a DAG and finally to its result variables. Based on the semantics of linear algebra operators, computing the output matrix size of an operator is possible. In case of if or while conditions, the propagation procedure takes into account whether variable sizes change inside the loop and need to be re-propagated or whether both if and else conditions will result to the same output size.

Although the rewritings presented above are cost-based, there is no search algorithm that evaluates different plans in order to find the optimal one. However, when it comes to matrix multiplication chains, SystemML uses dynamic programming in order to find the multiplication order that

minimizes the number of operations needed to compute the final product.

The rewritings described above can be implemented in any system that supports linear algebra operations. In general though, linear algebra operators are not widely commutable and do not offer many opportunities for reorderings at the logical level, as it applies to relational operators. Apart from altering the order of matrix multiplication chains, most of the rewritings at the logical level are based on simple mathematical properties.

Cost-based optimization techniques are also applied on physical plans of linear algebra operators. The most advanced system at this front is again SystemML, which chooses different implementations for linear algebra operators based on cost functions. Its physical plans, LOP DAGs, consist of the physical operators for each high-level (logical) operator of the HOP DAG. The LOP DAG that corresponds to the HOP DAG of listing 13 is displayed in listing 16 . There may be implementations of LOPs for various runtime engines (single node, MapReduce and Spark), which is indicated by the first token of each runtime instruction in the LOP DAG, and more than one implementations of an operator for a specific execution engine.

The choice between single-node and distributed execution is based on memory estimates for each HOP and the available budget of a single machine. Memory estimates assume single-threaded execution and are computed recursively from the leafs of a HOP DAG to its internal nodes using a precise model of dense and sparse matrices. The memory for internal HOPS is the sum of the estimates from their child nodes, the memory for intermediate results and the output memory of the HOP. Hence, HOPS that need less memory than the available budget in a single machine, will be executed locally, since local in-memory execution is assumed to be always cheaper than distributing data over the network. In case CP mode is not possible, the optimizer chooses a distributed LOP implementation based on a different cost function that takes into account I/O cost, shuffle cost and the degree of parallelism. Shuffle cost is the cost to redistribute data from mappers to the appropriate reducers on a distributed framework. It involves the time to write and read data from the file system divided by the number of mappers and reducers. For example, in case of a multiplication between a matrix and a small vector, the optimizer can choose to send a copy of the vector in every machine and perform local partial aggregations of the results, in order to avoid shuffle costs.

In the LOP plan of listing 16 all operators are executed on a single node. This is because memory estimates for these operators as computed in HOP plan of listing 13 are very small and rounded to 0MB. Also, two different physical implementations are used for the HOP of matrix multiplication. Lines 4 and 16 use the default algorithm for an expression of type `X%*%Y`, whereas the last line calls a different implementation named `tsmm`, which is suitable when a matrix is multiplied with its transpose, i.e. `t(X)%*%X`, and is used for the expression `t(error)%*%error`.

Listing 16: Sample LOP plan corresponding to listing 13

```
CP createvar _mVar65 true MATRIX binaryblock 1 506 1000 1000 −1
        copy
CP r' error.MATRIX.DOUBLE _mVar65.MATRIX.DOUBLE 1
CP createvar _mVar66 true MATRIX binaryblock 1 13 1000 1000 −1
        copy
CP ba+* _mVar65.MATRIX.DOUBLE features.MATRIX.DOUBLE
        _mVar66.MATRIX.DOUBLE 1
CP createvar _mVar67 true MATRIX binaryblock 13 1 1000 1000 −1
        copy
CP r' _mVar66.MATRIX.DOUBLE _mVar67.MATRIX.DOUBLE 1
CP createvar _mVar68 true MATRIX binaryblock 13 1 1000 1000 −1
        copy
CP / _mVar67.MATRIX.DOUBLE 506.SCALAR.INT.true _mVar68.
        MATRIX.DOUBLE
CP createvar _mVar69 true MATRIX binaryblock 13 1 1000 1000 −1
        copy
CP −* weights.MATRIX.DOUBLE 1.0E−7.SCALAR.DOUBLE.true
        _mVar68.MATRIX.DOUBLE _mVar69.MATRIX.DOUBLE
CP createvar _mVar70 true MATRIX binaryblock 506 1 1000 1000 −1
        copy
CP ba+* features.MATRIX.DOUBLE _mVar69.MATRIX.DOUBLE
        _mVar70.MATRIX.DOUBLE 1
CP createvar _mVar71 true MATRIX binaryblock 506 1 1000 1000 −1
        copy
CP − _mVar70.MATRIX.DOUBLE labels.MATRIX.DOUBLE _mVar71.
        MATRIX.DOUBLE
CP createvar _mVar72 true MATRIX binaryblock 1 1 1000 1000 −1
        copy
CP tsmm _mVar71.MATRIX.DOUBLE _mVar72.MATRIX.DOUBLE
        LEFT 1
```

Finally, in the context of work for in-database linear algebra [57] described in section 4.5.3, preliminary size-dependent ideas on executing plans which involve both relational and linear algebra operators more efficiently are explored. For example, when combining matrix multiplication with join operators, an optimizer aware of the dimensions of the involved matrices can choose a better plan. It could choose to perform matrix multiplication before joining, in order to reduce the size of the matrices moving up the plan, instead of performing a series a joins and leave matrix multiplication for the end.

5.3 Optimization and UDFs

Systems that adhere to the class of extensions of the MapReduce model depend heavily on user-defined functions, as their operators are second-order functions. These UDFs are written in functional or imperative languages, such as Scala or Java, and the semantics of the code is unknown. As a result optimization of UDFs present specific challenges [70]. In this section, we describe methods proposed by Flink and Tupleware, which analyze the code of UDFs to some extent to identify particular types of optimization opportunities. Spark, despite making heavy use of second-order functions, does not provide optimization techniques for their content.

Flink uses static code analysis to determine which data are read and written from each operator and separate them into read and write sets. By checking whether read and write sets overlap, we are able to know whether a reordering between two operators would result to a semantically equivalent plan. This idea is quite similar to the concurrency control techniques used in databases. So when only the read sets of two operators overlap, reordering of them will not break the semantics of the program. On the other hand, when read and write sets overlap, we are not able to ensure semantic equivalence. In case of group by operators, the optimizer can also determine whether the cardinality of the grouping attribute will remain unchanged after a reordering. That

ensures that the input size of the grouping operator will also be the same. It is clear that this method still does not understand the semantics of the code and as a consequence it is conservative in the sense that it probably forbids valid reorderings and misses optimization opportunities.

Apart from the logical phase, Flink exploits read and write sets of UDFs in optimization of physical plans. By knowing whether a UDF modifies a partitioning or sorting key, it is possible to determine whether physical data properties change. For example, if a grouping operator has already partitioned data based on the same key that a join operator will be applied on, there is no need to reshuffle data and increase network I/O. However, this technique is not enough to provide an accurate I/O cost for each plan and Flink at the moment bases its cost estimations on hints for UDF selectivity provided by the user or derived from earlier phases.

Tupleware also introspects UDFs by examining their LLVM intermediate representation, but focuses on other types of information. The purpose of its analysis is to determine vectorizability and estimate CPU and memory requirements of the UDF code. Vectorizability achieves data level parallelism by using the registers of a processor to apply a single instruction on multiple data simultaneously. CPU cycles can provide an estimation of compute time, whereas memory bandwidth can be used to predict load time of operands. Therefore, these two metrics can estimate whether a problem is compute-bound or memory-bound by comparing compute to load time. These statistics allow Tupleware to employ an adaptive strategy that switches between pipelining and bulk processing.

As an example consider map operators. It is common to group consecutive maps to a single pipeline, in order to leverage data locality. In addition to this, Tupleware can identify which of the map UDFs are vectorizable, exploit SIMD vectorization for them and cache intermediate results to avoid delays. In case there is one or more vectorizable UDFs at the beginning of the pipeline, Tupleware's optimizer can examine statistics of load time to evaluate whether fetching UDF operands would be faster than UDF computations. Based on this evaluation, it will then determine whether it should apply SIMD vectorization or stick to the initial operator pipelining.

Concerning reduce operators, the construction of the hash table can also be parallelized using SIMD vectorization. Moreover, in case reduce is based on a single key and the aggregation function is commutative and associative, Tupleware again computes partial aggregates in parallel using vectorization and combines them at the end to derive the final result.

6 Comparison and Discussion

After presenting a broad range of systems for declarative data analytics, we summarize how the described systems align with the properties of Section 2. TileDB is excluded, as it currently provides only storage management. Table 2 summarizes the results of this analysis. The last column of the table concerns the scope of the systems with regard to the following data science areas: data processing (DP - relational operators and other data transformations), machine learning (ML - linear algebra operators), convex optimization (LP - linear programming). The term "machine learning" refers to operators for building ML algorithms, not black box ML libraries. Table 3 also explains the types of plan optimizations that are developed by each system. We do not consider optimizations that are provided by existing optimizers and are used as they are by the surveyed systems, such as BUDS and MLog leveraging SimSQL's and Datalog's optimizers respectively. Moreover, as described in section 5.1, MLog supports an optimization technique that identifies components of a program, which can be evaluated in parallel. However, neither further details are provided in the paper [56] describing how the synchronization of results is achieved, nor experiments using this technique are reported. Therefore, we cannot assume that automatic parallelization is supported at the scale that it is by systems like Spark and SciDB. For this reason we do not include MLog in Table 3.

Some of the systems compared in table 2 are early research prototypes or do not have any follow ups, so it is understood that they may not be credible options for large-scale development. Nevertheless, we believe that the study of their architecture and features may benefit projects with larger adoption, which may want to consider adding a more declarative approach as an extra layer.

Based on these properties in table 2, we can see that the most declarative approaches include: SciDB, MLog, LogicBlox and the extension of SQL with linear algebra operators. These systems provide or emulate both data processing and commonly used operations in ML, offer plan optimization and the programs in the languages they support lack control flow. Moreover, user defined functions are only necessary when a task cannot be expressed using the primitives of the language, since operators in these languages do not serve as second-order functions. It is important to note that SystemML, despite not fully declarative according to the properties of the table, is a strong player in the area as an Apache project focused on large-scale machine learning with advanced optimization techniques and increasing community adoption.

Regarding most declarative systems, there are still differences in their design choices. SciDB and MLog support an array-based data model, instead of a relational one. LogicBlox and the extension of SQL with linear algebra operators both work directly on relational data. The LogicBlox database follows the model+solver paradigm, where models are defined as Datalog programs and the solution is provided by convex optimization solvers, whereas an extension of SQL would natively support linear algebra operators, but the mathematical optimization algorithm, e.g. gradient descent need also be coded by the user.

These last two approaches emphasize the benefits of the relational model. Despite the limitations of relational algebra when it comes to looping, its operators and the concept of relation serve preprocessing and feature engineering tasks very well and therefore cannot be dropped from the ML toolkit. Based on this observation, two main directions are forming in the area of declarative machine learning. One direction claims that we should give up on the database paradigm and use different platforms for machine learning and data storage. Advocates of this direction seem to believe

TABLE 2: Declarativity of systems based on the seven properties

System/Language	Independence of Data Abstractions	DP Ops	ML Ops	Plan Optimization	Lack of Control Flow	Automatic Computation of Solution	Limited dependence on UDFs	Scope
Pig Latin	✓	✓		✓	✓		✓	DP
Jaql	✓	✓		✓	✓		✓	DP
U-SQL	✓	✓		✓	✓		✓	DP
Spark		✓	✓	✓				DP, ML
Flink (Stratosphere)		✓		✓				DP
DryadLINQ		✓		✓				DP
Tupleware		✓		✓				DP
SystemML	✓		✓	✓			✓	ML
Mahout Samsara			✓	✓			✓	ML
BUDS	✓		✓		✓		✓	ML
TensorFlow			✓	✓		✓	✓	ML
PyTorch			✓			✓	✓	ML
Lara	✓	✓	✓	✓				DP, ML
SciDB		✓	✓	✓	✓		✓	DP, ML
LogicBlox	✓	✓		✓	✓	✓	✓	DP, LP
MLog	✓		✓	✓	✓	✓	✓	ML
An extension of SQL with linear algebra [57]	✓	✓	✓	✓	✓		✓	DP, ML

TABLE 3: Optimization capabilities per system

System or Language	Relational Algebra	Linear Algebra	User Defined Functions	Compiler-Based Optimizations	Automatic Parallelization
Pig Latin	✓				
Jaql	✓			✓	
U-SQL	✓				
Spark	✓			✓	✓
Flink Stratosphere	✓		✓		✓
DryadLINQ					✓
Tupleware	✓		✓	✓	
SystemML		✓		✓	
Mahout Samsara		✓			
TensorFlow				✓	
Lara	✓	✓	✓	✓	
SciDB					✓
LogicBlox	✓				✓
An extension of SQL with linear algebra [57]	✓	✓			

that the current situation will become the status quo and the idea of integrating machine learning into the database will eventually fade away due to usability, performance and expressivity challenges that would be not addressed in an efficient manner. In this case however, an open question remains: how can we get closer to a declarative specification of machine learning algorithms when we program them in systems that use imperative languages? Notice that none of the systems in this category scores high in the seven properties of table 2. The other direction estimates that history will repeat itself. As it happened with other processing needs in the past that were ultimately integrated with database architectures, such as stream processing or full text search, at some point it will come down to adding a suitable set of operators to the relational algebra or merging relational with linear algebra, in order to provide a unified environment for data science. Although the approaches that show high declarativity according to our framework set a foundation for this direction, the problem of declarative data science and machine learning cannot be considered solved.

7 Conclusion

We presented an extensive survey over systems in the area of declarative data analytics. The area can be divided to a number of categories, each one following a different programming model and degree of declarativity. Today, data scientists use a hairball of these systems, as each of them fits best different scenarios. It is thus frequent to build pipelines of various frameworks to support all the components of a machine learning solution. Building such pipelines involves lots of glue code, as well as tedious ETL processes, and requires at least a basic level of familiarity with a number of systems and languages. As a result, those approaches are not

easily maintained. We argue that the challenges described above and the solutions that will surround them, form the driving force that will determine which of the two main directions will prevail, either database management systems or general-purpose ML systems operating on denormalized data. Based on our analysis, it seems that despite early trends in the area favouring Map-Reduce based frameworks, the database ecosystem strikes back and proposes more sophisticated approaches for analytic tasks than the early black-box libraries in the same environment where data live.

Acknowledgments

We thank Panagiotis-Ioannis Betchavas for the implementation of Linear Regression using DML in section 4.4.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, 2004, pp. 137–150.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 59–72.

[3] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009.

[4] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 1099–1110.

[5] R. Chaiken, B. Jenkins, P.-r. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "Scope: Easy and efficient parallel processing of massive data sets," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1265–1276, Aug. 2008.

[6] F. Chollet *et al.* (2015) Keras. https://github.com/fchollet/keras.

[7] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 2135–2135.

[8] S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: a next-generation open source framework for deep learning," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015.

[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[10] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[11] C. De Sa, A. Ratner, C. Ré, J. Shin, F. Wang, S. Wu, and C. Zhang, "Deepdive: Declarative knowledge base construction," *SIGMOD Rec.*, vol. 45, no. 1, pp. 60–67, Jun. 2016.

[12] Tensorflow probability. https://www.tensorflow.org/probability/overview.

[13] Pyro. https://pyro.ai/.

[14] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan, "Mlbase: A distributed machine-learning system." in *CIDR*. www.cidrdb.org, 2013.

[15] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms," in *Proc. of KDD-2013*, 2013, pp. 847–855.

[16] A. Kumar, R. McCann, J. Naughton, and J. M. Patel, "Model selection management systems: The next frontier of advanced analytics," *SIGMOD Rec.*, vol. 44, no. 4, pp. 17–22, May 2016.

[17] M. Boehm, A. V. Evfimievski, N. Pansare, and B. Reinwald, "Declarative machine learning - A classification of basic properties and types," *CoRR*, vol. abs/1605.05826, 2016.

[18] S. Chaudhuri and K. Shim, "Optimization of queries with user-defined predicates," *ACM Trans. Database Syst.*, vol. 24, no. 2, pp. 177–228, Jun. 1999.

[19] F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas, "Opening the black boxes in data flow optimization," *PVLDB*, vol. 5, no. 11, pp. 1256–1267, 2012.

[20] F. Hueske, M. Peters, A. Krettek, M. Ringwald, K. Tzoumas, V. Markl, and J. Freytag, "Peeking into the optimization of data flow programs with mapreduce-style udfs," in *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, 2013, pp. 1292–1295.

[21] A. Thomas and A. Kumar, "A comparative evaluation of systems for scalable linear algebra-based analytics," *Proc. VLDB Endow.*, vol. 11, no. 13, pp. 2168–2182, Sep. 2018. [Online]. Available: https://doi.org/10.14778/3275366.3284963

[22] G. Graefe, "Query evaluation techniques for large databases," *ACM Comput. Surv.*, vol. 25, no. 2, pp. 73–169, Jun. 1993.

[23] ——, "Volcano— an extensible and parallel query evaluation system," *IEEE Trans. on Knowl. and Data Eng.*, vol. 6, no. 1, pp. 120–135, Feb. 1994.

[24] M. T. zsu and P. Valduriez, *Principles of Distributed Database Systems*, 3rd ed. Springer Publishing Company, Incorporated, 2011.

[25] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "Mllib: Machine learning in apache spark," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 1235–1241, Jan. 2016.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[27] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009.

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[29] Apache mahout. http://mahout.apache.org/.

[30] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar, "The madlib analytics library: Or mad skills, the sql," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1700–1711, Aug. 2012.

[31] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C. Kanne, F. Özcan, and E. J. Shekita, "Jaql: A scripting language for large scale semistructured data analysis," *PVLDB*, vol. 4, no. 12, pp. 1272–1283, 2011.

[32] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The stratosphere platform for big data analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, Dec. 2014.

[33] U-sql. http://usql.io/.

[34] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/pacts: A programming model and execution framework for web-scale analytical processing," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 119–130.

[35] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik, "Tupleware: "big" data, big analytics, small clusters." in *CIDR*, 2015.

[36] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–14.

[37] Apache flink. https://flink.apache.org/.

[38] C. Doulkeridis and K. Nørvåg, "A survey of large-scale analytical query processing in mapreduce," *The VLDB Journal*, vol. 23, no. 3, pp. 355–380, Jun. 2014.

[39] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient iterative data processing on large clusters," in *36th International Conference on Very Large Data Bases*, Singapore, September 14–16, 2010.

[40] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 810–818.

[41] Spark sql. https://spark.apache.org/sql/.

[42] Flink table api. https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/table/index.html.

[43] Spark dataframe api. https://spark.apache.org/docs/latest/sql-programming-guide.html.

[44] A. Alexandrov, A. Katsifodimos, G. Krastev, and V. Markl, "Implicit parallelism through deep language embedding," *SIGMOD Rec.*, vol. 45, no. 1, pp. 51–58, Jun. 2016.

[45] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "Systemml: Declarative machine learning on mapreduce," in *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ser. ICDE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 231–242.

[46] Z. J. Gao, S. Luo, L. L. Perez, and C. Jermaine, "The buds language for distributed bayesian machine learning," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: ACM, 2017, pp. 961–976.

[47] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. E. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, "Mli: An api for distributed machine learning." in *ICDM*, H. Xiong, G. Karypis, B. M. Thuraisingham, D. J. Cook, and X. Wu, Eds. IEEE Computer Society, 2013, pp. 1187–1192.

[48] A. Kunft, A. Katsifodimos, S. Schelter, S. Breß, T. Rabl, and V. Markl, "An intermediate representation for optimizing machine learning pipelines," in *Proceedings of the 45th International Conference on Very Large Data Bases, VLDB 2019, August 26-30, 2019, Los Angeles, California, USA*, 2019.

[49] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht, "Keystoneml: Optimizing pipelines for large-scale advanced analytics," in *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, 2017, pp. 535–546.

[50] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman, "The architecture of scidb," in *Proceedings of the 23rd International Conference on Scientific and Statistical Database Management*, ser. SSDBM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 1–16.

[51] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson, "The tiledb array data storage manager," *Proc. VLDB Endow.*, vol. 10, no. 4, pp. 349–360, Nov. 2016.

[52] E. Soroush, M. Balazinska, S. Krughoff, and A. Connolly, "Efficient iterative processing in the scidb parallel array engine," in *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, ser. SSDBM '15. New York, NY, USA: ACM, 2015, pp. 39:1–39:6.

[53] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, "Design and implementation of the logicblox system," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1371–1382.

[54] M. Kifer and Y. A. Liu, Eds., *Declarative Logic Programming: Theory, Systems, and Applications*. New York, NY, USA: Association for Computing Machinery and Morgan & Claypool, 2018.

[55] N. Makrynioti, N. Vasiloglou, E. Pasalic, and V. Vassalos, "Modelling machine learning algorithms on relational data with datalog," in *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, ser. DEEM'18. New York, NY, USA: ACM, 2018, pp. 5:1–5:4.

[56] X. Li, B. Cui, Y. Chen, W. Wu, and C. Zhang, "Mlog: Towards declarative in-database machine learning," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1933–1936, Aug. 2017.

[57] S. Luo, Z. J. Gao, M. N. Gubanov, L. L. Perez, and C. M. Jermaine, "Scalable linear algebra on a relational database system," in *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, 2017, pp. 523–534.

[58] J. H. Davenport, Y. Siret, and E. Tournier, *Computer Algebra: Systems and Algorithms for Algebraic Computation*. London, UK, UK: Academic Press Ltd., 1988.

[59] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *Journal of Machine Learning Research*, vol. 18, pp. 153:1–153:43, 2017.

[60] Matlab. https://www.mathworks.com/products/matlab.html/.

[61] Mathematica,. https://www.wolfram.com/mathematica/.

[62] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine, "Simulation of database-valued markov chains using simsql," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 637–648.

[63] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1383–1394.

[64] A. Kumar, J. Naughton, and J. M. Patel, "Learning generalized linear models over normalized data," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1969–1984.

[65] M. Schleich, D. Olteanu, and R. Ciucanu, "Learning linear regression models over factorized joins," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016, pp. 3–18.

[66] M. Abo Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich, "In-database learning with sparse tensors," in *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. SIGMOD/PODS '18. New York, NY, USA: ACM, 2018, pp. 325–340.

[67] L. Chen, A. Kumar, J. Naughton, and J. M. Patel, "Towards linear algebra over normalized data," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1214–1225, Aug. 2017.

[68] J. Seib and G. Lausen, "Parallelizing datalog programs by generalized pivoting," in *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '91. New York, NY, USA: ACM, 1991, pp. 241–251.

[69] J. Zhou, P. Larson, and R. Chaiken, "Incorporating partitioning and parallel plans into the SCOPE optimizer," in *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, 2010, pp. 1060–1071.

[70] A. Rheinländer, U. Leser, and G. Graefe, "Optimization of complex dataflows with user-defined functions," *ACM Comput. Surv.*, vol. 50, no. 3, pp. 38:1–38:39, May 2017.

Nantia Makrynioti is a PhD candidate in the Department of Informatics at the Athens University of Economics and Business. Her research interests lie in the intersection of database and machine learning systems. More specifically, her work focuses on integrating machine learning functionality with data query languages used in relational databases. She holds a BSc in Computer Science from the University of Ioannina and a MSc in Information Systems from her current University.

Vasilis Vassalos is a Professor in the Department of Informatics at the Athens University of Economics and Business. He has been working on data science challenges, including data integration, data cleaning, query optimization, and heterogeneous data processing since 1996. He has published more than 60 research papers in international conferences and journals and holds 2 US patents for work on information integration. His current work is on ML systems and biomedical and clinical data integration.