An Intermediate Representation for Optimizing Machine Learning Pipelines

Andreas Kunft[*] Asterios Katsifodimos[**] Sebastian Schelter[†]
Sebastian Breß[‡*] Tilmann Rabl[+] Volker Markl[‡*]

[*]TU Berlin [**]Delft University of Technology [†]New York University [‡]DFKI [+]HPI, Universität Potsdam

ABSTRACT

Machine learning (ML) pipelines for model training and validation typically include preprocessing, such as data cleaning and feature engineering, prior to training an ML model. Preprocessing combines relational algebra and user-defined functions (UDFs), while model training uses iterations and linear algebra. Current systems are tailored to either of the two. As a consequence, preprocessing and ML steps are optimized in isolation. To enable holistic optimization of ML training pipelines, we present Lara, a declarative domain-specific language for collections and matrices. Lara's intermediate representation (IR) reflects on the complete program, i.e., UDFs, control flow, and both data types. Two views on the IR enable diverse optimizations. Monads enable operator pushdown and fusion across type and loop boundaries. Combinators provide the semantics of domain-specific operators and optimize data access and cross-validation of ML algorithms. Our experiments on preprocessing pipelines and selected ML algorithms show the effects of our proposed optimizations on dense and sparse data, which achieve speedups of up to an order of magnitude.

PVLDB Reference Format:
Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. An Intermediate Representation for Optimizing Machine Learning Pipelines. *PVLDB*, 12(11): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/3342263.3342633

1. INTRODUCTION

Modern data analysis pipelines often include preprocessing steps, such as data cleaning and feature transformation, as well as feature engineering and selection [61, 62, 39, 15, 9]. Once data is in an appropriate shape, machine learning models are trained and evaluated. These training and model evaluation cycles are repeated several times to find the most suitable configuration of different features, machine learning (ML) algorithms, and hyperparameters. To build such training pipelines, data scientists can choose from a variety of tools and languages. Python and R offer popular libraries that are easy to use and provide fast development cycles. These libraries are embedded *shallowly* [29] in the host language, i.e., they are executed *as-is*, without any inter-library optimizations and support for large data [52]. General purpose dataflow systems [70, 2] provide second-order functions (e.g., *map* and *reduce*) to transform collections via user-defined functions (UDFs). They defer program execution by providing a type-based domain-specific language (DSL) that builds an associated operator graph. This operator graph is optimized and executed on a dedicated dataflow engine. In order to develop ML algorithms in such DSLs, linear algebra operations have to be retrofitted as UDFs on (distributed) collections. As a result, ML algorithms are hardcoded by experts and provided as library functions with fixed data representations and execution strategies. Thus, the semantics of linear algebra operations are concealed behind UDFs, which are treated as black boxes by optimizers [35]. In contrast, dedicated systems for ML, such as SystemML [12] and Tensorflow [1] provide linear algebra operations. However, it is difficult to express pipelines that include preprocessing and data transformation in these systems, as they lack dedicated types for collection processing. In summary, dedicated systems with type-based DSLs provide advantages over shallowly embedded libraries, but still suffer from three major problems in the context of end-to-end pipelines for model training: (*i*) Development, maintenance, and debugging of end-to-end pipelines is a tedious process in dedicated systems, and limits optimization potential and efficient execution. (*ii*) Preprocessing and ML are often executed in different systems in practice [59], which prevents optimizations across linear and relational algebra. (*iii*) Neither shallowly embedded libraries nor type-based DSLs can reason about native UDFs and control flow.

To address these issues, we propose Lara, a DSL that combines collection processing and machine learning.[1] Lara is based on *Emma* [3, 4], a quotation-based DSL [49] for (distributed) collection processing. Emma's `DataBag` algebraic data type enables declarative program specification based on *for-comprehensions*, a native language construct in Scala. In contrast to type-based DSLs, quotation provides access to the abstract syntax tree (AST) of the whole program, allowing us to inspect and rewrite UDFs and native control flow. Emma's intermediate representation (IR) is based on monad comprehensions [33] and enables operator fusion and implicit caching. Lara extends Emma with `Matrix` and `Vector` data types in its API and IR to execute pipelines for model training on single machines. It provides

[1] A preliminary short version has been published before [41].

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342633

two *views* on the IR to perform diverse optimizations: the *monadic view* represents operations on both types, `DataBag` and `Matrix`, as monad comprehensions in the IR. This common representation enables operator fusion and pushdown of UDFs across type boundaries, e.g., filter pushdown from a `Matrix` to a `DataBag`. Access to the control flow allows Lara to reason about operator fusion over loop boundaries, e.g., feature transformations that are iteratively applied over column ranges. The *combinator view* captures high-level semantics of relational and linear algebra operators as single entities in operator trees similar to relational algebra trees. It enables data dependent selection of specialized physical operators and implicit data layout conversions based on interesting properties [30]: similar to interesting properties of relational operators, e.g., sorted data for joins, linear algebra operators have preferred data access pattern, e.g., row-wise or column-wise.

In summary, we make the following contributions:

1. We propose Lara, a quotation-based DSL for end-to-end model training pipelines (Section 3).

2. We discuss our IR, which has access to the whole AST of the pipeline, and two views on top of it: a view based on monad comprehensions and a view based on the high-level semantics of operators (Section 3 and 4).

3. We discuss the extensibility of our approach by introducing a custom high-level operator and optimizations for *k-fold cross-validation*, a widely used technique to select hyperparameters for ML models (Section 4.4).

4. We conduct experiments on a typical preprocessing pipeline, and show the effects of data layout and cross-validation optimizations on selected ML algorithms for dense and sparse data. The experiments achieve speedups of up to an order of magnitude (Section 5).

2. BACKGROUND

In this section, we provide a brief overview of approaches to implement domain-specific languages. We refer to Alexandrov et. al [5] for an in-depth discussion and comparison.

Shallowly-Embedded Libraries. Many general-purpose programming languages (GPLs) provide dedicated libraries for different domains, such as collection processing and ML, e.g., Python's Pandas [47] and scikit-learn [55]. These libraries are embedded *shallowly* [29] in the host language (e.g., Python) and executed *as is*, i.e., without further optimizations except for those hard-coded in the algorithms and performed by the interpreter or compiler of the GPL.

Type-Based DSLs. Type-based or *symbolic* DSLs use the operations defined on a type (e.g., the `RDD` in Spark) to construct an operation graph, rather than directly executing the operations. Examples are the APIs of dataflow engines, such as Spark [70] and Flink [2], which mix the application of UDFs in second-order functions (e.g., `map`) and relational operators. More recent systems, such as MXNet [21] and Tensorflow [1], provide APIs based on tensors for ML. In contrast to shallowly embedded DSLs, type-based DSLs enable optimizations, such as choosing physical operator implementations and operator chaining. However, since the IR only reflects the operations defined on the type, host language UDFs are generally treated as black-boxes [35], and control flow is not visible. This can be partially addressed by specialized loop constructs [69, 26], but still prevents linguistic reuse [57] of the language's native control flow.

Standalone DSLs. Standalone DSLs, such as *SQL* and SystemML's *DML* [28] overcome the problems stated above. They provide a full-fledged compiler infrastructure for the DSL. However, a programmer has to develop this infrastructure (including libraries and development tools) from scratch, and support for complex UDFs written in GPLs is hard to achieve [54]. Another problem is user acceptance, i.e., to convince programmers to learn and adapt to the new language. Often, standalone DSLs try to ease the transition by staying close to the syntax of an existing language.

Quotation-Based DSLs. Quotation-based DSLs [49], such as LINQ [48], LMS [58], and Squid [53], reuse the syntax and type system of the host language and gives access to the program's AST during compilation and runtime. In contrast to type-based and shallowly-embedded DSLs, quotation gives access to the entire AST of the GPL, including types, white-box UDFs, and control flow. The AST can then be altered and optimized before execution. Thus, quotation-based DSLs overcome the limitations of the previously presented approaches but require careful design of domain-specific IRs.

3. LANGUAGE AND IR

In this section, we provide an overview of important design decisions for Lara, describe the IR of Lara, and introduce two views on top of the IR used to perform the diverse optimizations showcased in Section 4.

3.1 Language Design Decisions

We identified several shortcomings in current solutions, which led to the design of Lara. It extends the API and IR of Emma [3, 4], a quotation-based DSL for collection processing, with support for matrices and vectors. Users can express preprocessing and successive model training in the same program, which is reflected in a common IR. The following Lara code excerpt highlights these design decisions.

```
1  @lib def vectorizeComment(c: Comment) = { /* UDF */ }
2  @lib def vectorizeUser(u: User) = { /* UDF */ }                    ❶
3
4  optimize {                                                         ❷
5    // Join "Comments" and "Users" and vectorize the result
6    val features = for {
7      c <- Comments // DataBag[Comment]
8      u <- Users    // DataBag[User]
9      if u.user_id == c.user_id
10   } yield vectorizeComment(c) ++ vectorizeUser(u)
11   // Convert the DataBag "features" into matrix "X"
12   val X = Matrix(features)                                         ❸
13   // Filter rows that have values > 10 in the third column
14   val M = X.forRows(row => row(2) > 10)                            ❹
15   // Calculate the mean for each column
16   val means = M.forCols(col => mean(col))                          ❺
17   // Deviation of each cell of "M" to the cell's column mean
18   val U = M - Matrix.fill(M.nRows, M.nCols)((i,j) => means(j))
19   // Compute the covariance matrix
20   val C = 1 / (U.nRows - 1) * U ** U.t                             ❻
21 }
```

Lara enables the declarative specification of relational operators via Emma's `DataBag` data type with *for-comprehensions*. Line 6 – 9 illustrate a join between the datasets `Users` and `Comments`. ML pipelines are expressed as high-level linear algebra operators on the `Matrix` and `Vector` data type. For example, a matrix multiplication is specified using the `**` method in Line 20. Operators of both domains can

be interleaved with calls to user-defined (aggregate) functions: the tuples resulting from the join are converted to vectors in Line 10, followed by a filter predicate, which is applied to the rows of matrix **X** in Line 14. Separate types with dedicated syntax in the user-facing API reduce the impedance mismatch between relational and linear algebra, e.g., users do not have to specify linear algebra in terms of for-comprehensions over collections.

Next, we highlight key aspects of Lara's IR with respect to the code snippet:

❶ UDFs for the second-order functions of the `DataBag` (e.g., `map` or `fold`) and `Matrix` (e.g., `forRows` and `forCols`) are defined as closures (Line 14) or provided as library functions (Line 1 and 2). The body of library functions is inlined in case they are called within a pipeline (e.g., Line 10 and Line 16) and considered during optimization.

❷ `DataBag`, `Matrix`, and `Vector` types can be used without further optimization. This is useful to debug and test pipelines during development. To enable optimization, the very same pipeline is *quoted* by surrounding the code with an `optimize` macro [16] (Line 4).

❸ Type conversion methods (Line 12) in the API and the IR track data provenance, i.e., which field of a `DataBag` element corresponds to a given column in a `Matrix` and vice versa. It decouples the specification and execution of relational and linear algebra and enables joint optimization.

❹ Type and operator choices in the user facing API do not enforce a particular physical execution backend. A unified representation of both types in a common formal representation in the IR enables operator pushdown and fusion of UDFs over type boundaries. For instance, the filter UDF applied on each row of the matrix `X` in Line 14 can be pushed to the `DataBag` and fused with the `vectorize` UDFs applied in a `map` on the join result in Line 10.

❺ White-box UDFs in the IR enable reasoning about read and write accesses to the processed elements (e.g., fields, rows, and columns). In combination with access to the control flow in the IR, this provides opportunities to fuse UDF applications that are executed iteratively in a loop, if their read/write sets are disjoint. The iterative calculation of the mean in Line 16 can be optimized. Instead of executing the mean function on each column separately, it is executed for all columns at once.

❻ High-level linear algebra operators in the API (Line 20) and an IR that captures the domain-specific semantics of operators enable the selection of specialized operator implementations, e.g., BLAS [43] instructions for linear algebra.

3.2 Intermediate Representation

In this section, we describe how Lara's IR facilitates the design decisions described in the previous section. First, we introduce the low-level intermediate representation (LIR) provided by Emma. Then, we present two higher-level *views* on top of the LIR. The *monadic view* represents monad comprehensions [33] over the `DataBag` and `Matrix` types. The *combinator view* represents high-level operators (e.g., matrix multiplication) as single entities or *combinators* [32] in an operator tree.

Low-Level Intermediate Representation. Emma uses the meta-programming features of Scala [16] to access the AST of a quoted program. Emma transforms the original AST into let-normal form (LNF), a functional representation of static single assignment form (SSA) [8], to overcome several shortcomings of Scala's AST. LNF offers a normalized representation that encodes dataflow and control flow information directly. LNF guarantees that variables are defined only once, i.e., the *single static assignment* property. Thus, *def-use chains* [6] can be implemented efficiently. This property eases data dependency analysis, most notably the detection of dependencies across control constructs, e.g., between iterations of loops. The LIR is used as basis two *views*, which we introduce in the next paragraphs. The views combine expressions of the LIR that represent certain operations, e.g., a matrix multiplication, and make their semantics available for reasoning. The LIR augments the views by providing efficient data- and control flow analysis.

Monadic View. Monad comprehensions [33] on the *Bag* monad provide a concise and declarative way to specify collection transformations with first-class support for user-defined (aggregation) functions, as shown in the introductory example (Listing 3.1, Line 10). Matrices and vectors can also be represented as a *Set* monad $\{(\mathtt{i}, a)\}$ of index-value tuples, where `i` is a singular *index* in case of a vector and a tuple (*row-index*,*column-index*) in case of a matrix [27]. Writing linear algebra operators as monad comprehensions at the user-level is tedious and error prone. As a consequence, Lara offers high-level operators for linear algebra in its API. Monad comprehensions in the IR enable Lara to examine and optimize applications of UDFs, e.g., fusion and pushdown. These optimizations are not bound to the concrete monad instance, but rely on the general properties of monads. Lara extends the monad representation of the `DataBag` in Emma with the `Matrix` and `Vector` monad. A traversal over the LIR converts all explicit second-order functions on a `DataBag` to monad comprehensions (e.g., a `map` is converted to the corresponding comprehension). Calls to linear algebra methods and second-order functions of the `Matrix` type are replaced by their corresponding monad comprehensions. For instance, element-wise addition of two vectors is written as `x + y` in the API and represented as following AST nodes in the LIR:

```
Apply(Select(Ident("x"), "$plus"), Ident("y"))
```

In the monadic view, element-wise addition is represented as following monad comprehension [27]:

```
for {
  (idx_x, val_x) <- x          // generator
  (idx_y, val_y) <- y          // generator
  if idx_x == idx_y            // guard
} yield (idx_x, val_x + val_y) // head
```

The comprehension contains two *generators*, which bind each `(index, value)` pair of the two vectors `x` and `y`. The *head* expression is called for each combination of pairs that satisfies the *guard* expression. The values of all pairs that have the same index are added and form a new vector for the result of the addition.

Combinator View. The monadic view allows Lara to apply fusion over UDFs based on the properties of monads. In order to apply domain-specific optimizations and trace operator trees, data types and their respective high-level operations need to be represented explicitly in the IR [65]. Comprehension combinators [32] can be leveraged to represent high-level, logical operators whose semantics are not present in the monadic view. At the combinator view, relational operators, such as `join`, and linear algebra operations, such as

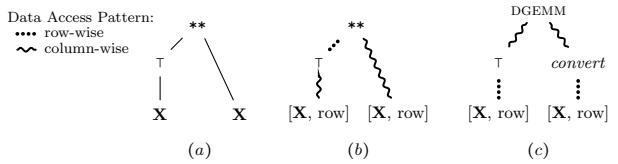


Figure 1: Combinator view for gram matrix $\mathbf{X}^\top\mathbf{X}$.

a matrix multiplication are captured as single entities and the program is viewed as a call or operator tree of these entities. This enables optimizations known from relational query processing (e.g., join reordering or the choice of different physical operator implementations). At the same time, it enables optimizations on the semantics of linear algebra. Logical operators can be replaced with specialized physical implementations, e.g., BLAS sub-routines [43]. Moreover, operators can propagate *interesting properties* to their child nodes, such as row- or column-wise access patterns to their operands. Based on these properties, different plan variants are generated. The combinator view for the computation of the gram matrix $\mathbf{X}^\top\mathbf{X}$ is illustrated in Figure 1a. Figure 1b depicts the operator tree with propagated access patterns – from the root expression to the sources. The matrix-multiplication (**) prefers fast access to the rows (dotted lines) of its left operand and the columns (snake lines) of its right operand. The physical row-wise data layout of the source matrix $\mathbf{X}$ is depicted in brackets in the Figure. Figure 1c represents a physical plan variant. The matrix-multiplication is replaced by a specialized BLAS instruction called DGEMM, which requests column-wise partitioned input. An *convert* enforcer [30] establishes the data layout desired by the DGEMM instruction for its second operand. We provide a detailed discussion in Section 4.3 and 4.4.

4. OPTIMIZING END-TO-END PIPELINES

In this section, we showcase how our IR enables different optimizations and present the LIR's interplay with the monadic and combinator view.

Running Example. Listing 1 depicts our running example of an end-to-end training pipeline, which leverages historical data about clicks on advertisements to predict the number of future clicks on other advertisements using a regression model. The pipeline showcases common user-defined feature transformations. We omit the implementation of the transformation UDFs for the sake of space. The categorical features in columns 11 to 15 are *dummy-encoded* [34] as sparse vectors in Line 4. The numerical features in columns 1 to 10 are *normalized* [31] to have zero mean and unit variance in Line 9. We concatenate the numerical features in Line 5 and combine them with the dummy-encoded features in Line 6, in order to end up with one vector per input record. After preprocessing, we evaluate different candidates for the hyperparameter `lambda` with cross-validation [37] on the normalized feature matrix $\mathbf{X}$. Lara provides cross-validation as utility method, similar to ML libraries such as scikit-learn. The learning algorithm supplied to the cross-validation is executed k times – for k different combinations of training sets and test sets obtained from the feature matrix $\mathbf{X}$ and target vector $\mathbf{y}$. This example trains a ridge regression model in Line 17 – 20, and calculates its test error in Line 22 – 23.

```
1  // Column 0 contains the target variable, columns 1-10 contain
2  // numerical and columns 11-15 contain categorical features
3  val dataset  = readAndClean("/path/to/data")
4  val encoded  = dummyEncode(dataset, 11 to 15)
5  val vectors  = concatNumericalFeatures(encoded, 1 to 10)
6  val features = concatVectors(vectors)
7  // y = 0: extract 1st column as target vector y
8  val (M, y)   = Matrix(features, y = 0)
9  val X        = Matrix.normalize(M, 1 to 10)
10 // Grid search over hyperparameter candidates
11 val regCandidates: Seq[Double] = // ...
12 for (lambda <- regCandidates) {
13   // 3-fold cross-validation for the hyperparameter lambda
14   val errors = ML.crossValidate(3, X, y) {
15     (X_train, X_test, y_train, y_test) =>
16     // Ridge regression
17     val reg = Matrix.eye(X_train.numCols) * lambda
18     val XtX = X_train.t ** X_train + reg
19     val Xty = X_train.t ** y_train
20     val w   = XtX \ Xty
21     // Calculate mean squared error on test set
22     val residuals = y_test - (X_test ** w)
23     residuals.map(r => r * r).agg(_ + _) / y_test.size
24   }
25   // Print mean error for chosen hyperparameter
26   println(errors.sum / k)
27 }
```

Listing 1: An ML training pipeline in Lara.

4.1 Operator Pushdown

Users can apply UDFs on Lara's `DataBag` and `Matrix` types. However, this does not enforce the concrete execution of the program, i.e., a UDF called in a `Matrix` operator can be rewritten to a `DataBag` operation and vice versa. For instance, consider the normalization of columns 1 – 10 in Listing 1, Line 9. Even though the user implements the normalization on the `Matrix` representation, it is beneficial to push the operation to the `DataBag` representation.

Analogous to many other common feature transformations (such as dummy or tf-idf [56] encoding), normalization is performed in two steps. These two steps are often called *fit* and *transform*, e.g., in scikit-learn and Spark MLlib. The fit step computes an aggregate over the feature column in a `fold`, e.g., the mean and variance in case of the normalize function. The successive transform step changes the values of the feature column based on the aggregation result of the fit step in a `map`, e.g., by subtracting the mean and dividing by the variance in case of the normalize function. These steps are then repeated for all columns 1 – 10. In our running example, the normalization is defined on the `Matrix` type – after all features are combined in a single sparse vector. Both functions, the `fold` and the `map`, call their UDF separately for each row. Thus, if the functions are executed on a `Matrix` in Compressed Sparse Row (CSR) format (as in our running example), the UDF performs the element-wise lookup of the column value on a sparse vector, which has logarithmic complexity in the number of non-zero values. However, the numerical features are still separate entries in the array element type (with constant time access) of the `DataBag`, before they are combined to a sparse vector with the categorical features in Line 5 and 6.

In the following, we detail how Lara can push UDF applications from one type to another in the IR. To this end, we first introduce *conversion methods*, which allow Lara to track data provenance across type conversions, i.e., how and where features are stored in both types. We then describe how a unified representation of `DataBag` and `Matrix` as monads in the IR enables the pushdown of UDFs.

Background: Conversion Methods. Generic type conversions correspond to a categorical concept called *natural transformations* [46]. Unfortunately, this concept can not be used directly when converting a `DataBag` to a `Matrix`. In this case, the container type changes from *Bag* to *Set* and the element type changes from `A` to $((i,j),A)$ to introduce the row- and column-index for the values. To overcome this problem, the conversion methods (Listing 1, Line 8) accept only `DataBag`s with instances of Lara's vector type as elements or expect an index function of form $idx : A \rightarrow ((i,j),A)$. We provide instances for `Product`, `Array`, and `Vector` in the moment. Therefore, Lara can track data provenance, i.e., how the access to a `Matrix` cell $((i,j),A)$ commutes with access to a $(i, Vector[A])$ element in a `DataBag`. The explicit representation of conversion methods and types as monad comprehensions, allows us to define mappings for the second-order functions that apply UDFs from the `Matrix` type to the `DataBag`.

Pushing down Bulk Operations. The unified representation as monads (and the corresponding conversions) enables us to reason about push downs of bulk operations (i.e., operations that apply UDFs to all the rows/columns of a `DataBag`/`Matrix`) in a sound way. For instance, consider the method `forRows(udf: Vector => Double)`, which applies an aggregation function to all rows of a `Matrix`. Intuitively, the UDF can be executed in a `map` on the `DataBag`, as its elements represent rows. The monad comprehensions for the `forRows` method exemplify this intuition:

```
1 for { rowCells <- M.groupBy(cell => cell.index.rowIndex) }
2 yield {
3   val rowVector = for { elem <- rowCells.values }
4                   yield (elem.index.colIndex, elem.value)
5   val aggregate = udf(rowVector)
6   (rowCells.key, aggregate)
7 }
```

The `Matrix` cells are grouped by their `rowIndex` in Line 1. Next, all values in a group (i.e., all cells of a row) are converted to a row vector $\{(colIndex, value)\}$ in Line 3 – 4 and then passed to the UDF in Line 5. The grouping and the conversion to a vector revert the conversion method. Thus, if we pushdown the UDF through the conversion method, the UDF can be executed in a `map` on the `DataBag`. Table 1 depicts mappings for all bulk-operations on rows of a `Matrix` after pushdown. Using the same mechanisms, we can execute bulk-operations defined over all columns, but need to convert to a `DataBag` of columns. For instance, executing the operation `forCols(a: Vector => Double)` on a `DataBag` requires the following comprehensions: `flatmap.groupBy.map.map(a)` as listed in Table 1.

Pushing down Row/Column Range Access. UDFs are often applied to particular row or column ranges. An example is the normalization in Line 9 of Listing 1, where the first 10 columns are normalized. Selection of a row in a `Matrix` is pushed to the `DataBag` as a `withFilter` method. A particular column is accessed via the `M.column(index)` method, which corresponds to the following monad comprehensions:

```
1 for { colCells <- M.groupBy(cell => cell.index.colIndex)
2       if colCells.key == index
3 } yield {
4   for { elem <- colCells.values }
5   yield (elem.index.colIndex, elem.value)
6 }
```

The guard (i.e., filter predicate) in Line 2 selects the group that matches the requested column index. Pushing the column selection to a `DataBag`, requires us to repartition its elements by their column index, as depicted in Table 1.

4.2 Operator Fusion

As discussed in Section 4.1, feature transformations apply two consecutive steps: the *fit* step aggregates column values (e.g., the mean and variance for normalization) in a `fold`. The *transform* step changes the column values based on the aggregate in a `map`. If the feature transformation is applied on multiple disjoint columns, Lara can fuse the consecutive `fold` and `map` applications. This allows us to share a given pass over the data, and only requires a single `fold` and `map` operation, independent of the number of transformed columns.

We briefly discuss operator fusion techniques, before we introduce the control flow and dependency analysis, which Lara applies to verify the applicability of operator fusion.

Background: Fold-Fusion. Fusion is based on two core operations of an algebraic data type `T` with element type `A`: the function application on each element

```
T[A].map[B](f: A => B): T[B]
```

and the generic structural recursion

```
T[A].fold[B](zero: B)(init: A => B, plus: (B, B) => B): B}
```

Function composition has been applied on several types [67, 22]. It fuses consecutive applications of UDFs in `map` second-order functions into a single, composed function call:

$$\texttt{T.map}(f_1)\texttt{.map(...).map}(f_N) = \texttt{T.map}(f_N \circ \ldots \circ f_1)$$

Fold-fusion combines multiple `fold` applications on a type to a single fold. In the following example code, the mean over a `DataBag[Int]` is calculated by computing the sum and the count of its elements:

```
val sum   = bag.fold(0)(e => e, (s1, s2) => s1 + s2)
val count = bag.fold(0)(e => 1, (c1, c2) => c1 + c2)
val mean  = sum / count
```

The *banana-split* [10] law states that pairs of folds that are applied on the same data type can be fused into a single fold, resulting in following code:

```
val (sum, count) = bag.fold((0,0))
  (e => (e, 1), ((s1, c1), (s2, c2)) => (s1 + s2, c1 + c2))
val mean = sum / count
```

The *cata-fusion* [10] law enables us to fuse `map` and `filter` operations into a consecutive `fold` application. Lara leverages the fusion capabilities of Emma [5], and applies them to the monad representations of `Matrix` and `Vector`.

Operator Fusion over Loops. We demonstrate operator fusion on the `dummyEncode` method of our running example (Listing 1, Line 4), which is implemented as follows. We hide the implementation details of the fit and transform UDFs and indicate their data dependencies with colored boxes.

```
1 def dummyEncode(bag: DataBag[Array[Any]], columns: Seq[Int]) = {
2   var encoded = bag
3   for ( columnIndex <- columns) {
4     // Fit: build a dictionary of column values
5     val dictionary = encoded.fold(fit-UDAF)
6     // Transform: create encoding in sparse vector
7     encoded = encoded.map(transform-UDF)
8   }
9   encoded
10 }
```

● def
○ use

Line 5 and 7 depict the dummy encoding for a single column, which is applied iteratively for all columns defined by the `columns` parameter. The `fold` creates a dictionary that maps each distinct column value to a unique index. The consecutive `map` replaces the categorical values by sparse vectors

containing a single non-zero entry at the index obtained by a dictionary lookup. A naïve execution of the code (i.e., independently on each column) is suboptimal, as it requires two passes over the data *per column*. In order to fuse the UDFs and save multiple passes over the data, Lara performs (*i*) a dependency analysis of the loop variable `columnIndex`, and (*ii*) an analysis of the UDFs to determine their access patterns to the elements of the `DataBag`.

Dependency Analysis. At first glance, it is not obvious that the operators can be fused. The `fold` in Line 5 appears to be a fusion barrier, as the previously built dictionary is required to perform a lookup in the `map` operator in Line 7. A closer lock reveals that the code accesses only a distinct feature column within each iteration. Thus, when we would *unroll* the loop, we could fuse the consecutive `fold` applications (Background: Fold-Fusion), *if* disjoint columns are accessed. Analogously, we could fuse all `map` applications. Lara analyses the loop based on the *direct style* control flow representation of the LIR. It validates that no *loop-carried dependencies* [6] exist and that the read and write accesses to the array elements inside the fit and transform UDFs are conducted with the `columnIndex` loop variable. Thus, Lara can verify that each consecutive iteration step reads and writes on disjoint columns, if the values of the loop variable are known at compile time (a constant sequence for instance, e.g., `11 to 15`) or the function semantics guarantee disjoint access (e.g., all Bulk-Operations in Table 1).

Fusion. After the dependencies have been evaluated successfully, Lara *unrolls* the loop to enable operator fusion. First, the banana-split [10] rule combines the UDFs executed in the `fold` operations, as they are applied on the same dataset; all dictionaries are created by executing a single combined `fold` only (Background: Fold-Fusion). Second, the successive transformations to sparse vectors in the `map` UDFs are fused, in order to apply all transformations in a single `map` operation. Thus, the optimized code executes a single `fold` and a single `map` only, independent of the number of transformed columns.

In general, operator fusion is always limited by pipeline breakers [50], i.e., (aggregated) data, which is required by an successive operator and thus, has to be materialized. While Lara can not overcome this inherent limitation, it can fuse multiple `folds` that are applied on the same data and thus, reduce the cost to a single pass over the data. Similar fusion techniques have been proposed in the *Stubby* [44] optimizer for Hadoop [7]. Lara leverages white-box UDFs and control flow analysis to ensure disjoint field access an thus enables these techniques over loop boundaries. In the moment, Lara requires direct access to the loop variable in its dependency analysis and does not support complex index expressions.

Type-based DSLs (e.g., in Spark and Flink) must execute the loop as-is, which is suboptimal as it prevents pipelining and fusion. Their IR can only reason about the operators, e.g., the `fold` and `map` higher-order functions in the example. Control flow and UDFs are not visible, which prevents the required dependency analysis.

4.3 Choosing a Data Layout

Choosing efficient physical operators for linear algebra operations, such as matrix-matrix multiplications, can have a huge impact on the runtime of ML pipelines [66]. We leverage the combinator view to choose appropriate physical implementations of operators based on the layout of the data.

Table 1: Operator pushdown between `DataBag` and `Matrix`.

	DataBag (row-wise)	**Matrix**
Bulk-Operations	`map(m:Vector => Vector)`	`forRows(m:Vector => Vector)`
	`map(a:Vector => Double)`	`forRows(a:Vector => Double)`
	`withFilter(f:Vector => Boolean)`	`forRows(f:Vector => Boolean)`
	`flatmap` : split in (colIdx, (rowIdx, value))	
	`.groupBy` : group by colIdx	`forCols(m:Vector => Vector)`
	`.map` : (rowIdx, value) : values as vector	`forCols(a:Vector => Double)`
	+ `.map(m)` or `.map(a)` or `.withFilter(f)`	`forCols(f:Vector => Boolean)`
Ranges	`withFilter(index:Int)`	`row(index:Int)`
	`flatmap` : split in (colIdx, (rowIdx, value))	
	`.groupBy` : group by colIdx	
	`.withFilter` : select column with `index`	
	`.map` : (rowIdx, value) : values as vector	`column(index:Int)`

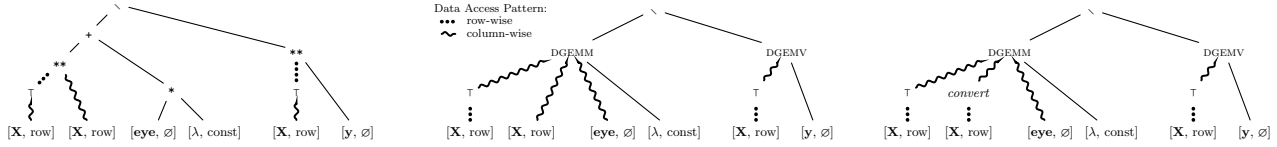
Figure 2 depicts three plan variants for the ridge regression algorithm (Listing 1, Line 17 – 20) in the combinator view. Operators are represented as single entities in an operator graph (e.g., `**` denotes matrix multiplication). Similarly to query optimization on relational algebra trees [30], Lara applies the following optimizations: (*i*) expressions (i.e., subgraphs) are transformed into equivalent expressions based on algebraic rules, (*ii*) logical operators are replaced by physical operator implementations and (*iii*) the desired physical data layouts are established by enforcers [30] based on *interesting properties*.

Transformation Rules. Lara provides an extensible set of rules to check for the applicability of backend specific operators. We define transformation rules to replace our default implementations of linear algebra operations in Scala. Lara applies these transformations for BLAS level 2 (i.e., matrix-vector) and level 3 (i.e., matrix-matrix) operations on dense data.[2] For instance Lara replaces the whole subtree for $\mathbf{X}^\top\mathbf{X} + \mathbf{I} * \lambda$ with a general level 3 BLAS matrix-matrix multiplication DGEMM (Figure 2b). Similarly, the general BLAS matrix-vector multiplication DGEMV is used to multiply $\mathbf{X}^\top\mathbf{y}$.

Physical Properties. Access patterns (row-, column-, or element-wise) of linear algebra operators and implementations can differ per operand. For example, a sparse matrix multiplication has fast access to the rows of the left operand and fast column-wise access to the right operand in the best case. Matching those access patterns has a large impact on the performance. For instance, suboptimal access pattern to Compressed Sparse Column (CSC) or CSR formats increases the asymptotic complexity from constant to logarithmic in the number of non-zero values.

To overcome this problem, we annotate the edges of plan variants with the access pattern of operators in a top-down traversal, similar to *interesting properties* in Volcano [30]. For instance, the default Scala implementation of a matrix-matrix multiplication (`**`) in Figure 2a yields the best performance in case of fast row-wise access to the left and fast column-wise access to the right operand. In contrast, BLAS sub-routines expect column-wise partitioned inputs, shown in Figure 2b for DGEMM and DGEMV. Lara considers the initial data format of matrices (depicted in brackets next to the matrices) to create plan variants by implicitly inserting conversion operators. Enforcers establish a certain data format, if the sources do not match the propagated access pattern. Furthermore, certain operator implementations can produce different output formats, which allows us to choose the for-

[2] `http://www.netlib.org/blas/#_blas_routines`



(a) (b) (c)

Figure 2: (a) the combinator view for ridge regression (Listing 1, Line 17 – 20) with default operators, (b) replaced sub-trees with equivalent BLAS instructions, and (c) an enforcer *convert* for the interesting properties of plan variant (b).

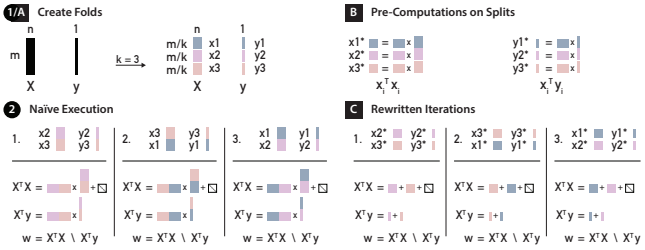


Figure 3: Cross-validation for ridge regression. Steps 1 and 2 depict the naïve execution with redundant computations. Steps A, B and C depict the optimized execution, which computes partial results outside the cross-validation loop.

mat depending on the properties of the parent node. Consider the plan variant in Figure 2b as an example. Remember that DGEMV and DGEMM expect column-oriented inputs, while the matrix $\mathbf{X}$ is in a row-wise format. The interesting properties of the DGEMV BLAS instruction match: the transposition $\top$ inverts the column-wise properties, matching the original row-wise format of $\mathbf{X}$. The same happens to the DGEMM BLAS instruction: the layout of matrix $\mathbf{X}$ matches the expectation of the first operand of DGEMM, but not the second (i.e., $\mathbf{X}$ has to be converted to a column-wise layout). Figure 2c depicts a plan variant with an enforcer to convert $\mathbf{X}$ to a column-wise data layout. The **eye** method, which creates an identity matrix, can produce the requested column-wise format seamlessly.

Rule-Based Plan Selection. Lara currently employs a heuristic plan selection strategy which is based on rules derived from the results of micro-benchmarks for operators. In case of dense data, we observe that BLAS instructions heavily outperform our self-implemented operators. Therefore, we greedily select the plan that replaces the highest number of operators with the lowest numbers of BLAS instructions, i.e., we promote the usage of BLAS instructions that cover the largest sub-trees. In case of sparse data, the layout choice has a major impact on the performance, and we choose plan variants that match the desired properties with enforcers. Note that we leave building a cost-model as well as a cost-based optimizer for future work.

4.4 Cross-Validation

Lara enables the integration of new high-level operators into its API and the definition of additional optimizations based on their semantics. To showcase this feature, we introduce an optimization for k-fold cross-validation [37], which is a common technique in ML pipelines to select well-working hyperparameters for models. Lara pre-computes linear algebra operations on the individual training set splits outside of the validation loop to avoid redundant computations.

Language Integration. We implement cross-validation as utility function (Listing 1, Line 14). In our running example, we use cross-validation to select a regularization constant `lambda` for the ridge regression model.[3] Its execution is illustrated on the left side of Figure 3. Step 1 is independent of the validation algorithm and partitions the specified feature matrix $\mathbf{X}$ and target vector $\mathbf{y}$ into k splits. In Step 2, the ridge regression algorithm is executed k times. For $k = 3$, the algorithm is executed once with $\mathbf{X}_1$ as test set and $\mathbf{X}_{2,3}$ as training set, next with $\mathbf{X}_2$ test set and $\mathbf{X}_{3,1}$ as training set, and finally with $\mathbf{X}_3$ as test set and $\mathbf{X}_{1,2}$ as training set.

Lara leverages the semantics of cross-validation to execute certain linear algebra operations more efficiently. We concentrate our detailed discussion on model training and, without loss of generality, leave the calculation of the test error in Line 22 – 23 out of the discussion.

Detecting Redundant Computations. If we look carefully at the execution of the learning algorithm in Figure 3, we observe that each feature matrix and target vector split is used twice ($k-1$ times in general) as part of the training set. Thus, the training algorithm is executed $k-1$ times on each split. This overlap poses potential for optimization: we can avoid redundant computations by (partially) *pre-computing* the algorithm on the individual splits outside of the validation loop.

The semantics of cross-validation guarantee that the individual splits do not overlap. Therefore, we can treat the splits as block-wise partitioning of the feature matrix $\mathbf{X}$ with size $m \times n$ into k matrix blocks $\mathbf{X_1} \dots \mathbf{X}_k$ with size $\frac{m}{k} \times n$. Block-partitioned matrices can be multiplied when they have *conformable partitions* [25], i.e., the block matrix itself and the individual blocks obey the rules of matrix multiplication. For example, $\mathbf{X}^\top\mathbf{X}$ can be calculated as sum over its k splits (i.e., matrix blocks $\mathbf{X_1} \dots \mathbf{X}_k$) using the distributive law: $\sum_{i=1}^{k} \mathbf{X}_i^\top \mathbf{X}_i$. This allows us to *pre-compute* the matrix multiplication of the individual splits $\mathbf{X}_i^\top\mathbf{X}_i$ once outside of the cross-validation loop (Figure 3, Step B). In each particular iteration, the pre-computed results of the test-set splits have to be added to calculate the final result (Figure 3, Step C). The time complexity for $\mathbf{X}^\top\mathbf{X}$ for a matrix with m rows and n columns is $\mathcal{O}(n * m * n)$. Under regular execution, the multiplication has to be executed for each training set with $m - \frac{m}{k}$ rows and n columns. Thus, the overall complexity for all k iterations in the regular execution is $\mathcal{O}(k[n*(m-\frac{m}{k})*n]) \equiv \mathcal{O}((k-1)mn^2)$. In the optimized execution, the multiplication is performed once for all *individual splits* in Step B. Each split has $\frac{m}{k}$ rows and n columns, resulting in the complexity of $\mathcal{O}(k[n\frac{m}{k}n]) \equiv \mathcal{O}(mn^2)$ for all k splits. In each iteration k we only have to add $(k-1)$ pre-

[3] Ridge regression is used for presentation reasons only – other algorithms, e.g., generalized linear models are supported as well.

Table 2: Linear algebra operations and their respective cost in cross-validation. $\oplus$ stands for element-wise operations.

Operation			**Time Complexity**			
Baseline	Pre-Computation	Per-Iteration	Baseline	Static	Loop	Ratio
$\mathbf{X}^\top\mathbf{X}$	$\forall i, 0 < i \leqq k : \mathbf{X}_i^* = \mathbf{X}_i^\top \mathbf{X}_i$	$\sum_{i=1}^{k-1} \mathbf{X}_i^*$	$(k-1)mn^2$	mn^2	$(k-1)kn^2$	$1:(k-1)$
$\mathbf{X}\mathbf{X}^\top$	$\forall i, 0 < i \leqq k : \mathbf{X}_i^* = \mathbf{X}_i \mathbf{X}^\top$	concat $\mathbf{X}_{[i..k]}^*$	$kn(m-\frac{m}{k})^2$	m^2n	$(k-1)m^2$	$1:(k-2)$
$\mathbf{X}^\top\mathbf{y}$	$\forall i, 0 < i \leqq k : \mathbf{y}_i^* = \mathbf{X}_i^\top \mathbf{y}_i$	$\sum_{i=1}^{k-1} \mathbf{y}_i^*$	$(k-1)mn$	mn	$(k-1)kn$	$1:(k-1)$
$\mathbf{X}\mathbf{y}$	$\forall i, 0 < i \leqq k : \mathbf{y}_i^* = \mathbf{X}_i \mathbf{y}$	concat $\mathbf{y}_{[i..k]}^*$	$kn(m-\frac{m}{k})$	mn	$(k-1)m$	$1:(k-1)$
$\mathbf{X} \oplus \mathbf{X}$	$\forall i, 0 < i \leqq k : \mathbf{y}_i^* = \mathbf{X}_i \oplus \mathbf{X}_i$	concat $\mathbf{X}_{[i..k]}^*$	$(k-1)mn$	mn	$(k-1)mn$	$k:(k-1)$

calculated partitions of size $n \times n$, resulting in the overall complexity of $\mathcal{O}(k[(k-1)nn])$ for all iterations.

Cost Improvements. ML algorithms are composed of several operators for which we also exploit the data redundancy introduced by cross-validation. Table 2 (Time Complexity) shows a cost comparison for matrix-matrix, matrix-vector and element-wise operations. The calculation depicts the overall cost of executing the operator k times during cross-validation on a feature matrix $\mathbf{X}$ with m rows and n columns. $\mathbf{X}$ is divided into k splits and each split has $\frac{m}{k}$ rows and n columns. This results in $m - \frac{m}{k}$ rows per training set. *Baseline* shows the combined cost to execute the operator for all training sets, i.e., the overall cost for using the operator in standard k-fold cross-validation. *Static* is the one-time cost for the part of the operator that can be pre-computed statically on each split. *Loop* is the combined cost to calculate the final result based on the statically pre-computed values for all cross-validation iterations for the particular operator. *Ratio* depicts the ratio between the complexity of the baseline and the optimized version, e.g., $1:(k-1)$ means the baseline has $(k-1)$ × more time complexity. Such cost computations can also be applied to estimate the additional memory required to store pre-computed results.

Eliminating Redundant Computations. We introduced the cross-validation function in Lara's IR. Lara pattern matches calls to the cross-validation function and uses the combinator view to represent the linear algebra operations of the cross-validation UDF. It then traverses the combinator view of the UDF in post-order (i.e., from the sources) and checks if rewrites can be applied based on the rules of the currently traversed operator node. Matching operator nodes are split and the algorithm extracts operator trees for the Pre-Computation. The former sub-trees in the original tree are replaced with calls to their results, as depicted in Figure 4 for the ridge regression example. Two trees for the Pre-Computation are created, which calculate the matrix-matrix multiplication for $\mathbf{X}^\top\mathbf{X}$ and matrix-vector multiplication $\mathbf{X}^\top\mathbf{y}$ for the individual splits (Table 2 (Operation)). The extracted sub-trees from the Per-Iteration tree are replaced with the operation of the corresponding sub-tree root node (Table 2 (Per-Iteration)). An additional optimization pass eliminates remaining shortcomings once the optimization for the cross-validation body is finished, e.g., dead code elimination and common subexpression elimination (CSE) [6]. For instance, after the two trees for the Pre-Computation are created, the transposition ($\top$) of the splits $\mathbf{X}_{[1..k]}$ would be calculated for each tree separately. After CSE, the transposition is executed only once.

Often, cross-validation is also part of an outer loop (e.g., when different hyperparameter candidates are validated). In such cases, Lara moves loop-invariant Pre-Computations out of the loop. As a result, the Pre-Computations are computed only once in our example and we improve the performance by a factor of $1:(k-1)*h$ compared to naive execution, where h denotes the number of hyperparameter candidates. Thus, larger hyperparameter spaces can be explored in the same time, which potentially finds better models.

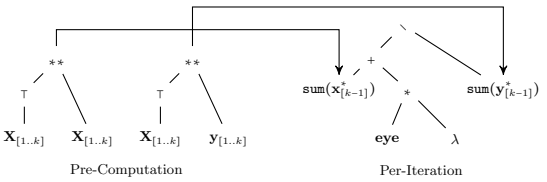


Figure 4: Optimized operator tree for ridge regression.

5. EVALUATION

We implemented Lara and the outlined optimizations in Scala, based on Emma v0.2.3. Operations on collections are backed by Scala Streams. The matrix/vector types apply netlib-java v1.1.2 for native BLAS routines in case of dense data, and ScalaNLP Breeze v0.13.1 for sparse data.

Experiment Setup. We conducted our experiments on a server node with a Intel E5530 processor (2.4GHz, 8 cores), and 48GB main memory. We run on Oracle Java 8 VM (build 1.8.0_72-b15, `-Xmx40g`) and use Scala version 2.11.11.

Datasets. We conducted our experiments on synthetic data and two real world datasets: (i) the Criteo[4] dataset contains click feedback of display ads. (ii) The *Reddit*[5] dataset contains comments of the news aggregator website *reddit.com*.

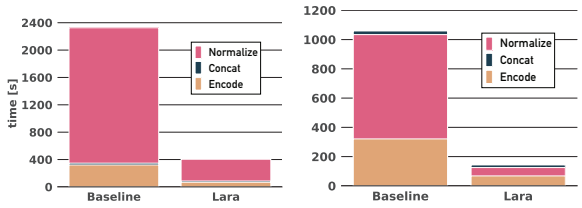
Overview. We evaluate the optimizations for the preprocessing pipelines based on the introductory example (Section 3.1) and our running example (Listing 1) and validate the importance of operator pushdown in Section 5.1 and 5.2. We measure the impact of data layout optimizations on ML algorithms for dense and sparse data in Section 5.3. We benchmark the optimizations for cross-validation in Section 5.4 and hyperparameter tuning in Section 5.5.

5.1 Preprocessing

In this experiment, we evaluate the presented optimizations on the preprocessing pipeline of our running example depicted in Listing 1, Line 3 – 9. We conducted the benchmark on differently sized versions of the Criteo dataset. We evaluate the impact of operator fusion and pushdown on each of the three preprocessing steps in Figure 5. Figure 5a depicts the runtime without pushing the normalization to the array representation of the data, while the pushdown is performed in the experiment represented in Figure 5b. *Encode* includes reading and converting the lines of the raw datafile to an array representation, as well as dummy encoding the categorical features to sparse vectors (Listing 1, Line 4). *Normalize* transforms the numerical features to have zero mean and unit variance (Listing 1, Line 9). *Concat* combines the numerical and the dummy encoded features in a single sparse vector (Listing 1, Line 5 – 6). In Figure 6, we compare Lara to scikit-learn, Spark, and Tensorflow Transform on different data sizes.

Discussion. *Baseline* shows the runtime for the pipeline executed without any optimizations. *Lara* depicts the results

[4] `http://labs.criteo.com/2013/12/download-terabyte-click-logs-2/`
[5] `http://files.pushshift.io/reddit/comments/`



(a) without pushdown (b) with pushdown

Figure 5: Preprocessing steps in detail on a 5GB sample.

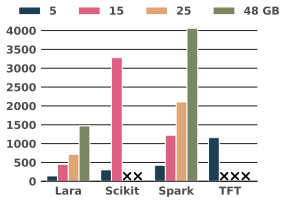


Figure 6: Preprocessing on different data sizes and systems.

with operator fusion and function composition (Section 4.2). Figure 5a depicts the results without operator pushdown for the normalization. Thus, the normalization is executed on sparse vectors. Under baseline execution, (*i*) the encoding is conducted for each column separately, requiring $5 * 2$ passes over the data. (*ii*) the value concatenation is executed in two separate map operators, and (*iii*) the normalization is applied to each column separately, requiring $10 * 2$ passes over the data. Lara enables the following optimizations: (*i*) Instead of separately encoding each column, a single fold creates all column dictionaries and then leverages these for encoding the column values in a single map operation. This reduces the complexity to two passes, independent of the number of encoded columns. (*ii*) Lara fuses the two map UDFs for concatenating the vector into a single map, which reduces the number of functions calls. (*iii*) The normalization benefits in the same way as the encoding, and reduces the number of passes to two. As the normalization is not pushed to the array representation, access to the numerical features in the UDFs suffers from the slow element-wise access of the sparse vectors: element-wise access requires a binary search with a cost that is logarithmic in the number of non-zero values of the row. Figure 5b depicts the results with operator pushdown in the baseline and Lara. As the elements are still stored in an array, read and write access to the features has constant cost. The scaling benchmarks in Figure 6 show that Lara and Spark scale linear with the increasing data size. Both execute in a streaming fashion and thus, are not affected by growing data sizes. Scikit-learn loads the whole dataset in memory, which leads to degrading performance for larger data sizes.

Results. The baseline without pushdown of the feature normalization takes 5.75× longer than Lara without pushdown and 16.1× than the completely optimized version. Overall, the baseline with pushdown is 7.3× slower than Lara. Lara improves the runtime for encoding by 4.7× compared to the baseline as the number of passes over the data is independent from the number of encoded features. Normalization with pushdown is 12.5× faster. This is roughly twice as much improvement compared to the encoding. This is expected as twice as much columns are normalized. Even though the access to the columns in logarithmic time degrades the overall runtime of Lara without pushdown, it is still 6.2× faster than the baseline without pushdown. Concatenation of the features to a single sparse vector requires no data materialization, as only `map` operators are used. Thus, the baseline and Lara can both stream data, and the function composition applied by Lara does not yield significant benefits. Figure 6 shows that scikit-learn initially outperforms single core Spark but degrades heavily and fails to execute for the 25GB sample due to out-of-memory errors. It already uses 15GB of memory for the smallest sample. For the 15GB sample, scikit-learn uses the whole 48GB main-memory available on the cluster node, which leads to 10.7× worse performance compared to the initial data sample. Lara outperforms scikit-learn by 2.1 and 7.3×. Lara consistently executes around 3× faster than single threaded execution in Spark. Spark with 8 parallel executors outperforms Lara (on a single core) by a factor of 2.2×. Tensorflow runs on Apache Beam, but only supports Google Dataflow and the unoptimized DirectRunner as backends in the moment. We ran Tensorflow Transform (TFT) on our cluster node with the DirectRunner, which failed with an *realloc* error after successfully applying the preprocessing for the 5GB sample. It fails to execute on the larger samples.

5.2 Operator Pushdown

In this experiment series, we evaluate the effects of operator pushdown based on Line 6 – 14 in the introductory example in Section 3.1, which applies a filter on the third column of the vectorized join result. The baseline executes the pipeline as specified, applying the filter on the matrix type. Lara pushes the filter application to the `DataBag` representation, before it is converted to a matrix. We conduct the experiments on a normalized version of the Reddit dataset with 1.4 million users and 31 million comments. The vectorize UDFs extract the `id`, `down-votes`, `up-votes`, and perform feature-hashing [68] of the n-grams obtained from the `user-name` ($n = 2$) and `comment-text` ($n = 10$) to a fixed, sparse vector space of 10000 and 50000.

Discussion. As described in Section 5.1, the element type of the `DataBag` representation (`product` types for user and comment) provides constant time access. The filter UDF of the `forRows` method is called for each row of the CSR matrix. Element-wise access to a particular value in the sparse row vector has logarithmic complexity. In a CSC matrix, the filter UDF could be evaluated for all non-zero values of the vector that represents the filtered column, but would require a conversion beforehand. It is important to note that the pushdown is only possible, because the filter is applied on the numerical feature `down-votes`. An inherent barrier for the pushdown of a function f, applied on columns c, is any previously applied function g, which is applied on the same columns c and has no inverse function. The feature hashing transformation has no inverse and thus, prevents operator pushdown.

Results. Lara takes 120.60 seconds to create the matrix representation of the filtered join result. Without filter pushdown, the execution is 7.3× slower and takes 881.41 seconds.

5.3 Data Layout

In this experiment series, we benchmark the impact of the matrix data layout on the performance of ML algorithms. We first evaluate ridge-regression as shown in Listing 1, Line 17 – 20. It calculates the solution directly using a

Table 3: Benchmarks on data access patterns.

Algorithm	Variant	Feature Layout	
		column-wise	row-wise
Ridge Regression	Scala	22.81 ± 0.155 s	56.57 ± 0.346 s
	BLAS	0.44 ± 0.272 s	0.46 ± 0.072 s
	BLAS+Convert	0.63 ± 0.003 s	0.64 ± 0.097 s
Logistic Regression w/ BGD 1 Iteration	Breeze	1.36 ± 0.074 s	1.53 ± 0.075 s
	Breeze+Convert	0.09 ± 0.002 s	0.07 ± 0.003 s
Logistic Regression w/ BGD 100 Iterations	Breeze	130.37 ± 2.579 s	168.99 ± 8.080 s
	Breeze+Convert	0.92 ± 0.143 s	0.87 ± 0.041 s

```
1 var weights = Vector(...) // initialize weight vector
2 for (_ <- 0 until Iterations) {
3     val hyp  = X ** w
4     val exp  = hyp.map(value => 1 / (1 + math.exp(-1 * value)))
5     val loss = exp - y
6     weights  = weights - alpha * ((X.t ** loss) / X.numRows)
7 }
```

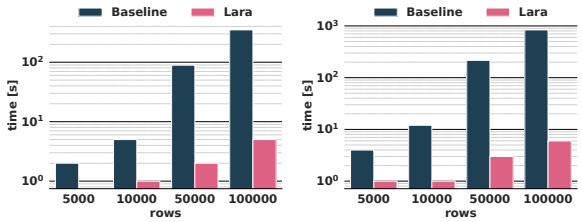
Listing 2: Logistic regression with BGD in Lara.

solver. Next, we evaluate logistic regression with batch gradient descent (BGD). The algorithm calculates the model iteratively over a fixed number of iterations. An implementation in Lara is depicted in Listing 2. We use a synthetic dataset with 10000 rows and 1000 columns.

Discussion. All results are depicted in Table 3. Ridge regression is conducted on dense data with row- and column-wise formats of feature matrix $\mathbf{X}$. *Scala* depicts the result for our own Scala implementations of dense linear algebra operators. *BLAS* depicts the results with BLAS instructions. *BLAS+Convert* depicts the results with BLAS and enforcers that establish the desired data layout for the `dgemm` instruction. *Scala* executes the plan depicted in Figure 2a for row-wise and column-wise features. *BLAS* executes the variant shown in Figure 2b for both layouts. For row-wise features, *BLAS+Convert* executes the plan shown in Figure 2c. For column-wise features, *BLAS+Convert* executes a plan variant that converts the input to the transpose ($\top$), both for the `dgemm` and `dgemv` instruction, to achieve a compliant data layout. The results for logistic regression are conducted on sparse data (10 percent non-zero values). Analogous to the ridge regression example, we conducted the experiments on row- and column-wise features $\mathbf{X}$. *Breeze* depicts the results for Lara, which internally uses the Breeze library to execute sparse linear algebra on matrices in compressed sparse row (CSR) and column (CSC) layout. *Breeze+Convert* depicts the results when an enforcer establishes the desired data layouts of the operators (Listing 2): for row-wise features, *Breeze+ Convert* converts the feature matrix $\mathbf{X}$ used in the multiplication with the loss vector `X.t ** loss`; for column-wise features, *Breeze+Convert* introduces an enforcer for $\mathbf{X}$ read in the multiplication with the weight vector `X ** w`.

Results. The benchmarks for ridge regression on dense data show the importance of specialized physical operators. BLAS is 51.8× faster for column-wise and 122× faster for row-wise features compared to the Scala implementation. For the Scala implementation, the column-wise feature layout matches the properties of the operators (Figure 2a) and is 2.4× faster than the row-wise feature layout. Converting the matrix for the BLAS instructions (*BLAS+Convert*) introduces a performance overhead of 1.43× for column and 1.39× for row-wise features: faster execution of the BLAS instruction can not overcome the overhead of the conversion.

The experiments for sparse data show the importance of choosing the best-suited data format. The initial feature layout only satisfies the access pattern of one of the two



(a) 5 folds, dense data (b) 10 folds, dense data

Figure 7: Cross-validation with ridge regression.

matrix-vector multiplications. Column-wise partitioned features are slightly faster, as the loss vector used in `X.t ** loss` is larger than the weight vector (by factor 10 in our experiments). The variants that introduce an enforcer to satisfy the desired access pattern of the operators increase the performance by a factor of 15.1× for 1 and up to 141.7× for 100 iterations in case of column-wise partitioned features. A enforcer for row-wise partitioned features brings the execution time on par with the column-wise features and achieves an up to 194× performance improvement for 100 iterations. This is due to the asymptotic access cost, which changes from logarithmic to constant.

5.4 Cross-Validation

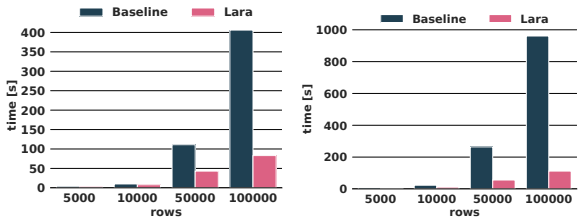
In this experiment series, we benchmark the impact of the proposed rewrites for cross-validation. We evaluate ridge regression as shown in Listing 1, Line 17 – 20 and logistic regression with batch gradient descent (BGD) (Listing 2). Each figure depicts the results for synthetic data with a fixed number of columns (1000) and folds (5 and 10). The number of rows in the dataset is scaled on the x-axis.

Ridge Regression Discussion. Figure 7 depicts the results for dense data. The *Baseline* implementation executes the algorithm without the proposed optimizations for cross-validation described in Section 4.4. *Lara* executes the matrix-matrix and matrix-vector multiplications in a Pre-Computation step on each split before the cross-validation iterations are executed.

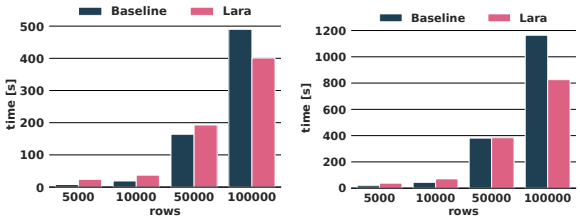
Ridge Regression Results. Lara is up to 65× faster than the Baseline for five folds and up 136× faster for ten folds. This heavily exceeds the expected ratio from the cost estimation in Table 2. We relate this to the very small intermediate result for the Pre-Computation of $\mathbf{X}^\top\mathbf{X}$. The intermediate results for the individual splits have the size $n \times n$, where n is the number of columns in the training matrix $\mathbf{X}$.

Logistic Regression Discussion. Figure 8a and 8b depict the results on dense data, while Figure 8c and 8d depict the results for sparse data. The experiment setup matches the previous experiment series on ridge regression. In the *Baseline*, the two most expensive operations are the multiplication $\mathbf{X}w$ of the feature matrix $\mathbf{X}$ with the weight vector w, and the multiplication $\mathbf{X}^\top\mathbf{loss}$ of the transposed feature matrix with the loss vector **loss** to calculate the gradient. Lara is able to extract these operations to lower the computational complexity, as described in Section 4.4. The Pre-Computation of the hypothesis *hyp* can now be calculated for all k weight vectors w_k in a single matrix multiplication $\mathbf{X}_k\mathbf{W}$, by stacking all weight vectors into a matrix $\mathbf{W}$.

Logistic Regression Results. On dense data, Lara is up to 4.8× faster for 5 folds and 8.6× faster for 10 folds than



(a) 5 folds, dense data (b) 10 folds, dense data



(c) 5 folds, sparse data (d) 10 folds, sparse data

Figure 8: Cross-validation with logistic regression.

the baseline (Figure 8a and 8b). The impact of the redundant computations in the baseline grows with the number of rows in the training set. Additionally, Lara benefits from the more efficient execution that leverages a single matrix-matrix multiplication instead of multiple matrix-vector multiplications in the baseline. On sparse data, Lara achieves a speedup of up to 1.2× for 5 folds and 1.4× for 10 folds compared to the baseline (Figure 8c and 8d). The cross-validation optimization is only beneficial once the number of rows is larger than 50000 rows. In contrast to the dense implementation, the Pre-Computations for sparse data cannot leverage more efficient instructions, and the speedup is solely based on the cross-validation optimization.

5.5 Hyperparameter Tuning

In this experiment series, we benchmark the performance impact of our proposed rewrites for the cross-validation utility function with hyperparameter tuning. We evaluate ridge regression with different `lambda` values for the regularization matrix, as shown in the running example. Next, we evaluate logistic regression with BGD with different initializations of the weight vector w. The feature matrix has 1000 columns in both experiment series, while we scale the number of rows. We tune for 5 different hyperparameters and validate them with 5-fold cross validation. The logistic regression with BGD runs for a fixed amount of 100 iterations.

Ridge Regression Discussion. Figure 9 depicts the results of the benchmark. The *Baseline* executes the cross validation and hyperparameter loop without rewrites, but uses BLAS instructions. We provide experiments for two optimization variants: *Lara (CV only)* depicts the results of optimized cross-validation without removing loop invariant code. *Lara* depicts the results after the loop invariant code is pulled out of the hyperparameter loop.

Ridge Regression Results. As expected, the baseline implementation takes 5× longer than the single cross validation (Section 5.4). Lara (CV only) is up to 141× faster than the baseline, which is analogous to the improvements for a single cross-validation. Lara with all optimizations achieves up to 800× speedups compared to the baseline and is up to 8× faster than Lara (CV only).

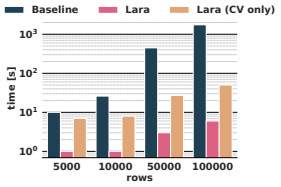
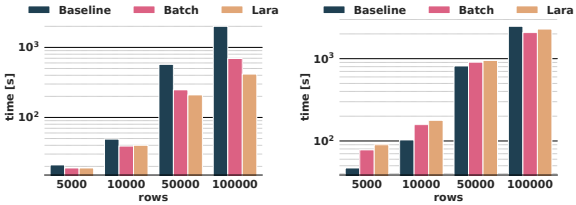


Figure 9: Hyperparameter tuning for ridge regression.



(a) Dense data (b) Sparse data

Figure 10: Hyperparameter tuning for logistic regression.

Logistic Regression Discussion. We evaluate the hyperparameter tuning for logistic regression on dense and sparse matrix representation with 10 percent non-zero values. Figure 10a depicts the results for dense matrix representation and Figure 10b for sparse matrix representation. We provide results for an implementation that uses *batching* as additional baseline (similar to the approach presented in TuPaQ [62]) to provide a reference point for our optimization. Batching reduces the number of times the dataset has to be read from n times (i.e., for each hyperparameter individually) to one time. It *batches* the model training by combining the weight vectors w (i.e., the hyperparameters) into a matrix $\mathbf{W}$, and thereby replaces n matrix-vector multiplications $h = \mathbf{X}w$ with a single matrix-matrix multiplication $Y = \mathbf{XW}$. Lara applies the optimizations presented in Section 4.4. The hyperparameter loop adds an additional nesting layer: we do not use a single weight vector w per split k, but a matrix $\mathbf{W}$, which contains all the weight candidates as columns, thus $hyp = [\mathbf{W}_0, ..., \mathbf{W}_k]$. Therefore, the resulting Pre-Computation for the individual splits involves k^2 iterations, as shown in the following code snippet:

```
for (i <- 0 until k) {
  for (j <- 0 until k) { // Wj from hyp-list
    val h    = X_train_i ** W_j
    val exp  = h.map(value => 1 / (1 + math.exp(-1 * value)))
    val loss = exp - y_train_i
    val s_i,j  = X_train_i.t ** loss
  }
}
```

Logistic Regression Results. Batching outperforms the baseline by up 5× for 10 hyperparameters. Lara achieves speedups of up to 8× compared to the baseline. Up to 10000 rows, Lara and batching provide comparable performance. For larger number of rows, Lara outperforms batching by up to 1.8×. For the sparse matrix representation we can observe that the baseline outperforms batching and Lara until a scaling factor of 50000 rows. For small number of rows, the baseline is up to 1.5× faster. Batching outperforms Lara by 1.1× and the baseline by 1.2×. We account the loss in Laras performance for smaller data sizes to the management and access cost of the weight matrices for the different hyperparameters. In future work, we plan to integrate batching as a rewrite rule for sparse data.

6. RELATED WORK

ML Libraries & Languages. SystemML [11] and Mahout Samsara [60] have R-like linear algebra abstractions and execute locally or distributed on Hadoop and Spark. They apply pattern-based rewrites and inter-operator optimizations such as operator fusion, and SystemML's execution strategy is based on cost estimates. Mahout Samsara does not provide substantial relational algebra capabilities. SystemML provides a transform function to apply pre-defined feature engineering methods, such as dummy encoding, binning, and missing value imputation, to raw datasets. SystemML can fuse the specified transformations, as their semantics guarantee disjoint column access. In contrast to Lara, users can not specify their own transformation UDFs in SystemML. OptiML [64] is a DSL for machine learning based on the Delite [18] framework. It shares a lot of ideas with Lara, as it provides pattern-based rewrites for linear algebra operations and operator fusion to avoid intermediate results. OptiML does not provide optimizations based on control flow analysis and is restricted to linear algebra operations. KeystoneML [63] executes ML pipelines on Apache Spark, automatically chooses solvers, and selects data materialization strategies. Due to its type-based DSL, KeystoneML can not apply operator re-ordering and fusion. To the best of our knowledge, Lara is the first language abstraction to combine linear and relational algebra, which is at the same time able to reason and optimize across the two algebraic abstractions, control flow and UDFs.

ML Specific Optimizations. Kumar et al. [40] propose learning of linear models on data in relational databases, which was later extended to linear algebra operators [20]. In this work, linear algebra operations can be pushed down to relations in databases, similar to [19]. Control flow, UDFs and general preprocessing pipelines are not considered. SystemML provides a *ParFOR* [14] primitive that executes the body of the loop in parallel or distributed. The operator fusion over loops, presented in Section 4.2, detects independent tasks (e.g., encoding of distinct columns), but fuses them instead of executing them in parallel. SystemML also performs operator fusion [23, 13] and generates linear algebra kernels based on skeleton classes. During a cost-based selection, the best plan with regards to fusion and caching for pipeline breakers is chosen. While the fusion techniques used in SystemML are superior to those presented in this work, SystemML does not consider collection processing for fusion. Yuan Yu et al. [69] extend TensorFlow with support for dynamic control flow, but, to the best of our knowledge, do not perform control flow and UDFs analysis to apply rewrites such as operator fusion. TuPaQ [62] is a framework for automated model training and supports custom optimizations such as *batching* to train multiple hyperparameters for linear models in parallel, which can be integrated in Lara. MLBase [38] provides high-level abstractions for ML tasks and basic support for relational operators. Its optimizer can choose between different ML algorithm implementations. In contrast to Lara, it does not consider relational operators during optimization and thus provides no capabilities for holistic optimizations.

Execution Engines. Weld [51] focuses on efficient data movement of data-parallel operators between different libraries. Domain-specific optimizations such as reordering linear algebra and operator pushdown are not supported. Scalable linear algebra on a relational database system [45] proposes a system to efficiently execute and optimize linear algebra over a parallel relational DBMS. It uses the foreign function interface of the DBMS to execute UDFs and complex linear algebra operations, which prohibits holistic optimizations and requires data movement in case of end-to-end pipelines. Meta-Dataflows [17] proposes a framework for *exploratory* execution of dataflows. It provides a high-level API with ML algorithms as function calls and does not focus on optimizing pipelines including UDFs.

7. CONCLUSION

In this paper, we present Lara, a DSL and IR for ML training pipelines. We based Lara on three key requirements that a DSL design should adhere to, in order to enable holistic optimizations: (*i*) The user-facing API should be declarative and provide dedicated types for both domains – the execution order and operator implementation is independent of the program specification. (*ii*) The complete pipeline should be visible by the optimizer – next to the data types and operations, UDFs and control flow have to be analyzed to perform certain optimizations. (*iii*) The IR should provide different levels of abstraction for diverse optimizations – a unified representation of types is required to reason about operator fusion and pushdown, while domain-specific optimizations require a high-level representation of operator semantics. We showcase such a DSL and IR and presented concrete optimizations for ML training pipelines. Our evaluation shows that the proposed optimizations can yield speedups of up to an order of magnitude.

Limitations & Future Work. Our prototype is not integrated in a dedicated runtime nor uses code-generation in the moment. This would alleviate several shortcomings of our current implementation: we did not yet implement a robust caching mechanism, e.g., to test different models on the same feature set. Memory-safe caching requires runtime support; simply caching data in the Java Virtual Machine (JVM) heap is subject to out-of-memory errors. Emma supports caching for the `DataBag` type, but Lara misses a robust implementation for matrices in the moment. The common view as monads enables fusion of linear algebra operators with applications of UDFs. Lara currently does not apply fusion of linear algebra operators and UDFs applications, as our current dense (BLAS) and sparse (Breeze) backends do not support fused operators. Future work could extend our optimizations on data layout access patterns to generate kernels for sparse linear algebra operations with UDF support and hardware-efficient code by integrating ideas from recent work [36, 13, 42]. Furthermore, one could extend the combinator view by integrating more data representations (e.g., block-wise or compressed [24]). Finally, to support layout optimizations for intermediate results, we plan to extend Lara with runtime code analysis and an cost-based optimizer. Layout decisions that are dependent on dynamic control flow also require analysis at runtime, e.g., for conditional operations that depend on predicates that have to be evaluated at runtime.

Acknowledgments. This work has been supported by the EU project E2Data (ref. 780245), the German Ministry for Education and Research as BBDC 2 (ref. 01IS18025A) and BZML (ref. 01IS18037A), and the Moore-Sloan Data Science Environment at New York University.

8. REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] A. Alexandrov et al. The stratosphere platform for big data analytics. *VLDB Journal*, 23(6):939–964, 2014.

[3] A. Alexandrov et al. Implicit parallelism through deep language embedding. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 47–61. ACM, 2015.

[4] A. Alexandrov et al. Emma in action: Declarative dataflows for scalable data analysis. In *SIGMOD*, 2016.

[5] A. Alexandrov, G. Krastev, and V. Markl. Representations and optimizations for embedded parallel dataflow languages. *ACM Transactions on Database Systems (TODS)*, 44(1):4, 2019.

[6] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*, volume 1. Morgan Kaufmann San Francisco, 2002.

[7] Apache Hadoop, http://hadoop.apache.org.

[8] A. W. Appel. Ssa is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.

[9] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395. ACM, 2017.

[10] R. Bird and O. De Moor. The algebra of programming. In *NATO ASI DPD*, pages 167–203, 1996.

[11] M. Boehm et al. SystemML's optimizer: Plan generation for large-scale machine learning programs. *IEEE Data Eng. Bull.*, 37(3):52–62, 2014.

[12] M. Boehm et al. SystemML: Declarative machine learning on spark. *VLDB*, 9(13):1425–1436, 2016.

[13] M. Boehm, B. Reinwald, D. Hutchison, P. Sen, A. V. Evfimievski, and N. Pansare. On optimizing operator fusion plans for large-scale machine learning in systemml. *Proceedings of the VLDB Endowment*, 11(12):1755–1768, 2018.

[14] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. R. Burdick, and S. Vaithyanathan. Hybrid parallelization strategies for large-scale machine learning in systemml. *Proceedings of the VLDB Endowment*, 7(7):553–564, 2014.

[15] J.-H. Böse, V. Flunkert, J. Gasthaus, T. Januschowski, D. Lange, D. Salinas, S. Schelter, M. Seeger, and Y. Wang. Probabilistic demand forecasting at scale. *Proceedings of the VLDB Endowment*, 10(12):1694–1705, 2017.

[16] E. Burmako. Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, page 3. ACM, 2013.

[17] R. Castro Fernandez, W. Culhane, P. Watcharapichat, M. Weidlich, V. Lopez Morales, and P. Pietzuch. Meta-dataflows: Efficient exploratory dataflow jobs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1157–1172. ACM, 2018.

[18] H. Chafi et al. A domain-specific approach to heterogeneous parallelism. *ACM SIGPLAN Notices*, 2011.

[19] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *VLDB*, 1994.

[20] L. Chen, A. Kumar, J. Naughton, and J. M. Patel. Towards linear algebra over normalized data. *Proceedings of the VLDB Endowment*, 10(11):1214–1225, 2017.

[21] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[22] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *ACM SIGPLAN Notices*, volume 42, pages 315–326. ACM, 2007.

[23] T. Elgamal, S. Luo, M. Boehm, A. V. Evfimievski, S. Tatikonda, B. Reinwald, and P. Sen. Spoof: Sum-product optimization and operator fusion for large-scale machine learning. In *CIDR*, 2017.

[24] A. Elgohary et al. Compressed linear algebra for large-scale machine learning. *PVLDB*, 9(12):960–971, 2016.

[25] H. W. Eves. *Elementary matrix theory*. Courier Corporation, 1980.

[26] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *Proc. VLDB Endow.*, 5(11):1268–1279, July 2012.

[27] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems (TODS)*, 25(4):457–516, 2000.

[28] A. Ghoting et al. SystemML: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242. IEEE, 2011.

[29] J. Gibbons and N. Wu. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In *ACM SIGPLAN Notices*, volume 49, pages 339–347. ACM, 2014.

[30] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218. IEEE, 1993.

[31] J. Grus. *Data science from scratch: first principles with python*. " O'Reilly Media, Inc.", 2015.

[32] T. Grust. *Comprehending queries*. 2000.

[33] T. Grust and M. Scholl. How to comprehend queries functionally. *Journal of Intelligent Information Systems*, 12(2):191–218, 1999.

[34] D. Harris and S. Harris. *Digital design and computer architecture*. Morgan Kaufmann, 2010.

[35] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the black boxes in data flow optimization. *Proceedings of the VLDB Endowment*, 5(11):1256–1267, 2012.

[36] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, Oct. 2017.

[37] R. Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model

selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[38] T. Kraska et al. Mlbase: A distributed machine-learning system. In *CIDR*, volume 1, pages 2–1, 2013.

[39] A. Kumar, R. McCann, J. Naughton, and J. M. Patel. Model selection management systems: The next frontier of advanced analytics. *ACM SIGMOD Record*, 44(4):17–22, 2016.

[40] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, pages 1969–1984. ACM, 2015.

[41] A. Kunft, A. Alexandrov, A. Katsifodimos, and V. Markl. Bridging the gap: towards optimization across linear and relational algebra. In *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, page 1. ACM, 2016.

[42] A. Kunft, A. Katsifodimos, S. Schelter, T. Rabl, and V. Markl. Blockjoin: efficient matrix partitioning through joins. *Proceedings of the VLDB Endowment*, 10(13):2061–2072, 2017.

[43] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[44] H. Lim, H. Herodotou, and S. Babu. Stubby: A transformation-based optimizer for mapreduce workflows. *Proceedings of the VLDB Endowment*, 5(11):1196–1207, 2012.

[45] S. Luo, Z. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. Scalable linear algebra on a relational database system. *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[46] S. Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

[47] W. McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython.* " O'Reilly Media, Inc.", 2012.

[48] E. Meijer. The world according to linq. *Commun. ACM*, 54(10):45–51, Oct. 2011.

[49] S. Najd, S. Lindley, J. Svenningsson, and P. Wadler. Everything old is new again: Quoted domain-specific languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, pages 25–36, New York, NY, USA, 2016. ACM.

[50] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[51] S. Palkar et al. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

[52] S. Palkar, J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. Amarasinghe, et al. Evaluating end-to-end optimization for data analytics applications in weld. *Proceedings of the VLDB Endowment*, 11(9):1002–1015, 2018.

[53] L. Parreaux, A. Voizard, A. Shaikhha, and C. E. Koch. Unifying analytic and statically-typed quasiquotes. *Proceedings of the ACM on Programming Languages*, 2(POPL):13, 2017.

[54] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM, 2009.

[55] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[56] A. Rajaraman and J. D. Ullman. *Mining of massive datasets*. Cambridge University Press, 2011.

[57] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation*, 25(1):165–207, Mar 2012.

[58] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *ACM SIGPLAN Notices*, 2010.

[59] S. Schelter, F. Biessmann, T. Januschowski, D. Salinas, S. Seufert, G. Szarvas, M. Vartak, S. Madden, H. Miao, A. Deshpande, et al. On challenges in machine learning model management. *Data Engineering*, page 5, 2018.

[60] S. Schelter, A. Palumbo, S. Quinn, S. Marthi, and A. Musselman. Samsara: Declarative machine learning on distributed dataflow systems. In *Machine Learning Systems workshop at NeurIPS*, 2016.

[61] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems*, pages 2503–2511, 2015.

[62] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 368–380. ACM, 2015.

[63] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 535–546. IEEE, 2017.

[64] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun. Optiml: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616, 2011.

[65] R. Y. Tahboub, G. M. Essertel, and T. Rompf. How to architect a query compiler, revisited. In *Proceedings of the 2018 International Conference on Management of Data*, pages 307–322. ACM, 2018.

[66] A. Thomas and A. Kumar. A comparative evaluation of systems for scalable linear algebra-based analytics. *Proceedings of the VLDB Endowment*,

11(13):2168–2182, 2018.

[67] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, pages 344–358. Springer, 1988.

[68] K. Weinberger, A. Dasgupta, J. Attenberg, J. Langford, and A. Smola. Feature hashing for large scale multitask learning. *arXiv preprint arXiv:0902.2206*, 2009.

[69] Y. Yu, M. Abadi, P. Barham, E. Brevdo, M. Burrows, A. Davis, J. Dean, S. Ghemawat, T. Harley, P. Hawkins, M. Isard, M. Kudlur, R. Monga, D. Murray, and X. Zheng. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 18:1–18:15, New York, NY, USA, 2018. ACM.

[70] M. Zaharia et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.