Bridging the Gap: Towards Optimization Across Linear and Relational Algebra

Andreas Kunft Alexander Alexandrov Asterios Katsifodimos Volker Markl

TU Berlin
first.last@tu-berlin.de

ABSTRACT

Advanced data analysis typically requires some form of pre-processing in order to extract and transform data before processing it with machine learning and statistical analysis techniques. Pre-processing pipelines are naturally expressed in dataflow APIs (e.g., MapReduce, Flink, etc.), while machine learning is expressed in linear algebra with iterations. Programmers therefore perform end-to-end data analysis utilizing multiple programming paradigms and systems. This impedance mismatch not only hinders productivity but also prevents optimization opportunities, such as sharing of physical data layouts (e.g., partitioning) and data structures among different parts of a data analysis program.

The goal of this work is twofold. First, it aims to alleviate the impedance mismatch by allowing programmers to author complete end-to-end programs in one engine-independent language that is automatically parallelized. Second, it aims to enable joint optimizations over both relational and linear algebra. To achieve this goal, we present the design of Lara, a deeply embedded language in Scala which enables authoring scalable programs using two abstract data types (DataBag and Matrix) and control flow constructs. Programs written in Lara are compiled to an intermediate representation (IR) which enables optimizations across linear and relational algebra. The IR is finally used to compile code for different execution engines.

1. INTRODUCTION

Data analytics requirements have changed over the last decade. Traditionally confined to aggregation queries over relational data, modern analytics is focused on advanced in situ analysis of dirty and unstructured data at scale. Data sources such as log files, clickstreams, etc., are first cleansed using relational operators, and then mined using clustering and classification tasks based on statistical and machine learning (ML) methods. As a result, data cleaning and preprocessing is typically an initial step of advanced data analysis pipelines (e.g., product recommendations, statistical analysis). Moreover, the preprocessing logic and data representation often depends on the ML method that will be applied subsequently.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

BeyondMR'16, June 26-July 01 2016, San Francisco, CA, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4311-4/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2926534.2926540

While relational domain-specific languages (DSLs) such as Pig [14], Hive [15], or Spark SQL/DataFrame [3] are a good fit for ETL tasks, programming machine learning algorithms in those languages is cumbersome. To this end, R-like DSLs such as SystemML's DML [10] or Apache Mahout's Samsara were proposed. These DSLs offer linear algebra and control flow primitives suitable for expressing ML algorithms, but only provide limited, non-intuitive support for classic ETL tasks. This strict separation of programming paradigms introduces three fundamental problems.

Impedance Mismatch. Large analysis pipelines have to be authored in multiple DSLs, plumbed together by additional code, and possibly executed on different systems.

Inefficient Data Exchange. Pre-processing in a relational algebra (RA) DSL and subsequent learning in a linear algebra (LA) DSL enforces materialization of the intermediate results at the boundary. Moreover, unless the staging format offers meta-information about the physical layout, this information is lost when the boundary is crossed.

Loss of Optimization Potential. Separating RA and LA into distinct DSLs entails that different, albeit similar intermediate representations (IRs) and compilation pipelines are used for the two languages. For example, compare the compilation phases and DAG-based IRs for Hive [15] and SystemML [4]. As a result, optimizations that could be applied among the IRs (e.g., filter and projection push-down, sharing of physical layout) are currently not possible.

To overcome these problems, we argue for *the unification of relational and linear algebra* into a common theoretical foundation. To achieve this goal, first we need to *explore and reason about optimizations across the two algebras* in a suitable intermediate language representation (IR). Second, we need to showcase the added benefits of unification and the optimizations that come thereof, by *defining a common DSL* with high-level programming abstractions for both relational and linear algebra. In line of the benefits offered by other UDF-heavy dataflow API's, the proposed DSL should be *embedded* in a host-language like Scala (e.g. Spark RDDs, Samsara) rather than *external* (e.g., Pig, DML).

In this paper we propose Lara, an embedded DSL in Scala which offers abstract data types for both relational and linear algebra (i.e., *bags* and *matrices*). We build our work on Emma [1, 2], a DSL for scalable collection based processing, which we extend with linear algebra data types. We exploit the code rewriting facilities of the Scala programming language to lift a user program into a unified intermediate

```
// Read measurements into the DataBags A and B
val A = readCSV(...) //
val B = readCSV(...)
// SELECT a_1, ...,a_N, b_1, ..., b_M
// FROM   A, B
// WHERE  A.id = B.id
val X = for {
        a <- A
        b <- B
        if a.id == b.id
       } yield (a_1, ...,a_N, b_1, ..., b_M)

// Convert DataBag X into Matrix M
val M = X.toMatrix()
// Calculate the mean of each column c of the matrix
val means = for ( c <- M.cols() ) yield (mean(c))
// Compute the deviation of each cell of M
// to the cell's column mean.
val U = M - Matrix.fill(M.numRows, M.numCols)
                      ((i,j) => means(j))
// Compute the covariance matrix
val C = 1 / (U.numRows - 1) * U.t %*% U
// Compute singular value decomposition
// e.g. rescale M, reduce dimensions, etc.
```

Listing 1: Code snippet written in Lara

representation for joint optimization of linear and relational algebra. Given this IR, a just-in-time (JIT) compiler generates code for different execution engines (Spark and Flink).

A Motivating Example. Consider a set of machinery sensors found in industrial plants taking part in the production of home mixers. In the end of the production line, a percentage of those mixers is found to be defective. The goal of our analysis is to train a classifier which will predict whether a mixer has high chances of being defective, based on the given measurements. For this task, we have to gather data from various log files residing in different production plants and join them in order to get all measurements of a mixer throughout its production. Since there are thousands of measurements per mixer and millions of mixers, we first run a Principal Component Analysis (PCA) to prune the number of measurements used for classification.

The above process can be implemented in Lara as shown in Listing 1. Lines 2 and 3 read the data from two different industrial plants before joining them (lines 7-11) to gain a full view over all measurements for each of the mixers. Note that the join is expressed as a native Scala construct, called `for`-comprehension (see [2] for details). Next, DataBag `X` which contains all projected measurements (a_1, ...,a_N, b_1, ..., a_M) is converted into Matrix `M` (line 14). The next line computes the mean of each of the measurement columns before we compute matrix `U`, holding the deviation of each of `M`'s cells from their corresponding column's mean using the `fill` operator (called "filling function" in [13]). Finally, line 22 computes the covariance matrix `C`. In the next step we would feed matrix `C` to the PCA algorithm, which is omitted from the example.

Discussion. Observe that the ETL part of the pipeline is expressed as a *declarative* program of transformations over DataBags, whereas the ML part is expressed in linear algebra. Moreover, no physical execution strategy has been predetermined by the programmer. Our matrix abstraction is strongly influenced by R's matrix package and includes all common operations on matrices. In addition, there are two explicit operations to convert from a `DataBag` to a `Matrix` and vise versa. Finally, note that converting a `DataBag` into a `Matrix` does not necessarily mean that an operation is going to take place on a physical `Matrix` representation. For example, consider a scalar multiplication of a `Matrix`: the multiplication could be applied directly on a `DataBag`, since scalar multiplications do not rely on special, linear algebra-specific operators. Thus, we let the optimizer decide in which physical data representation operations apply.

2. LANGUAGE FOUNDATIONS

Types as First Class Citizens. Our DSL is based around the concept of generic types. We propose a set of elementary generic types that model different structural aspects and represent both user-facing (e.g, matrix, bag) and engine-facing (e.g, partitioning) abstractions. Each type implies (i) a set of structural constraints expressed as axioms, as well as (ii) a set of structure-preserving transformations, which operate element-wise and correspond to the functional notion of map. Moreover, the types can be suitably composed to define new, richer structure. This allows for reasoning about the optimization space in a systematic way.

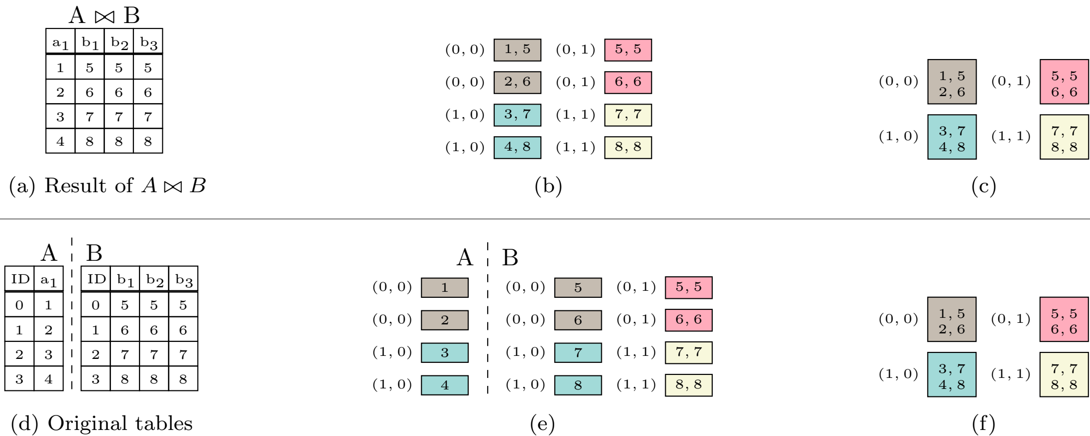
User Types. The core types included in our model are $\mathsf{Bag}\,A$, $\mathsf{Matrix}\,A$, and $\mathsf{Vector}\,A$. These types are polymorphic (generic) with a type argument A and represent different containers for data (i.e., elements) of type A. As such, their implementations should be independent on A.

Generic types can be modeled using algebraic data types (ADTs) using algebraic specifications [8]. An algebraic specification defines a set of functions that produce values of the specified type (constructors), as well as a set of axiomatic equations that relate constructor terms. This approach gives rise to categorical semantics of each specification – a free functor corresponds to the *classical* (or loose) semantics, and the initial object in the target category to the *initial* semantics. For instance, [1] advocates conceptually treating bags as types specified by the so-called union representation algebra:

$$\textbf{data}\ \mathsf{Bag}A = \mathit{empty} \mid \mathit{sng}\ A \mid \mathit{union}\ \mathsf{Bag}A \times \mathsf{Bag}A$$

Bag values can be constructed by a nullary constructor (*empty*), an unary constructor (*sng*), or a binary constructor (*union*), and the associated axioms state that `empty` is a unit and *union* is associative and commutative. LA types with fixed dimensions, such as $\mathsf{Vector}^n A$ and $\mathsf{Matrix}^{n\times m} A$, as well as the monoids used in [9] naturally fit this framework. The signature and dependencies between the constructors in an ADT definition impose certain *type structure*. Mappings that preserve this structure are called *homomorphisms*. In the case of generic types, the structure represents an abstract model for the shape of the container, and homomorphisms are characterized by a second-order function called map. It is important to note that map is predominantly associated with collections nowadays (as in MapReduce), while the concept of map is pervasive to *all* generic types.

System Types. Reasoning about optimizations that affect physical aspects such as partitioning or blocking means that those should be included in our model. Crucially, this type of structure can also be represented by generic types. For example, we can use $\mathsf{Par}\,\mathsf{T}\,A$ to represent a partitioned container of type $\mathsf{T}\,A$ (e.g., $\mathsf{Bag}\,A$, $\mathsf{Matrix}\,A$), and $\mathsf{Block}^n\,A$ to represent square-shaped blocks with dimension n. Homomorphisms (maps) over those, model partition- or block-preserving function applications (e.g., corresponding to mapValues in Spark's RDD API) respectively.



(a) Result of $A \bowtie B$ (b) (c)

(d) Original tables (e) (f)

Figure 1: Two execution strategies for *toMatrix*$(A \bowtie B)$ (colors represent different partitions). Above (a – c) Naïve Approach. Below (d – f) Partition Sharing.

Type Conversions. An obvious candidate to formalize generic type conversions in a categorical setting are *natural transformations* – polymorphic functions $t : \mathsf{T}\,A \rightarrow \mathsf{U}\,A$ which change the container type from T to U (e.g., from Matrix to Bag) irrespective of the element type A. Their characteristic property states that application of t commutes with application of $\mathsf{map}\,f$ for all f. This formalism, however, cannot be directly applied in all cases. For example *toMatrix* : $\mathsf{Bag}\,(\mathbb{N}, A) \rightarrow \mathsf{Matrix}\,A$ preserves the element value A but relies on an extra index $\mathbb{N}$ to determine the element position in the resulting matrix. Extending or adapting the theory of natural transformations in order to fit our needs is an open research question.

Control-Flow. To enable rewrite-based optimizations, our proposed language has to be *referentially transparent* (i.e., purely functional). Moreover, in order to facilitate efficient and concise implementations of optimizations, the language IR should satisfy the following requirements. (R1) Both elementary and compound expressions should be addressable by-name. (R2) Each use-def and def-use information should be efficient, and easy to compute and maintain. (R3) Control and data flow should be handled in a unified way.

All of the above requirements could be satisfied by an IR in *static single assignment (SSA)* form. Graph-based SSA IRs (e.g., sea of nodes) are nowadays used in compiler backends like LLVM and Java HotSpot. We plan to use a purely functional IR which conforms to the SSA constraints. It enforces R1 through a restriction on the allowed terms called *A-normal form*, and R2-R3 by modeling control-flow through function calls in the so-called *direct style*.

3. OPTIMIZATIONS

A number of holistic optimizations can be derived from the unified formal model and implemented under the assumption of a full view of the algorithm code. Examples are projection push down (based on knowledge that fields are never accessed), as well as filters that are e.g., applied on matrices whereas they can be pushed to the DataBags from which the matrices originate. In the sequel we present more sophisticated optimizations that come from a deeper analysis of a program's code.

Matrix Blocking Through Joins. Distributed operations over matrices are commonly done over *block-partitioned* matrices [10, 5]. This representation differs a lot from the unordered, non-indexed bag representation, commonly used in dataflow APIs.

Consider again the example in Listing 1. Lines 7-11 perform a join, producing a bag which is converted to a matrix and processed in lines 16 and 22. Note that the subsequent linear algebra operations (filling as well as computing the covariance matrix) can be executed over a block-partitioned matrix. A naïve execution of this program would require to: shuffle the data once in order to execute the join, and then shuffle once again to block-partition the matrix to perform the linear algebra operations. In the sequel we use an example to show (i) how the naïve approach would perform the join and subsequently the block partitioning, and (ii) how we can avoid one partitioning step (for the join).

Naïve Approach. Assume we execute the join of A and B as shown in Figure 1d on 4 nodes, using *hash-partitioning* with $h(k) = k \mod 4$, where k is the product id. To block partition the matrix for the subsequent linear algebra operations, systems typically introduce a surrogate key $rowIdx$ for each tuple of the join result, to assign subsets of the rows and columns to blocks. Therefore, we assign the following key to each tuple:

$$k = \left(\frac{rowIdx}{rowBlkSize}, \frac{colIdx}{colBlkSize}\right)$$

The result of this key assignment for the join result is depicted in Figure 1b. Note that the blocks are partial. A grouping operation can bring the partial blocks on the respective machines and construct the final blocks as shown in Figure 1f.

Partition Sharing. A full view over the code in Listing 1 allows us to see both, the RA part in lines 1-11, the LA part in lines 16-22, and to holistically reason about the type conversion in line 14. Ideally, the join operation and the linear algebra operations can share the same partitioning. We can achieve that by range-partitioning the input tables A and B separately and then combine them. More specifically, we use a different key to partition the inputs, taking into account both the (unique, and sequential[1]) product id and

[1]Similar optimizations apply on joins over non-unique keys (e.g. normalized data [12]). Moreover, the assumption of sequential primary keys can be relaxed in the expense of an extra pass over the data that is negligible in complex analysis programs. For the lack of space, we omit this discussion.

the column index:

$$k = (\frac{ID}{rowBlkSize}, \frac{resultColIdx}{colBlkSize})$$

where *resultColIdx* is the index of the column in the (now virtual) bag `X` and *ID* is the primary key of the inputs. As the schema of the join result is explicitly provided in Lara (Listing 1 line 11), we can easily obtain the column indexes of the join result (`X`). The partitioning of the tables is shown in Figure 1e. Observe that the blocks with column index 0 are split across the tables, thus, we also have column-wise partial blocks. To create the final partitioning with complete blocks, we `union` `A` and `B`, before we aggregate the blocks sharing the same block id.

Row-wise Aggregation Pushdown. In our example Listing 1, line 16 calculates the mean for each column in the matrix. Now, let us consider calculating the mean for each *row*, as shown in the following snippet:

```
// Convert DataBag X into Matrix M
val M = X.toMatrix()
// Calculate the mean of each row r of the matrix
val means = for ( r <- M.rows() ) yield (mean(r))
```

This would require a full pass over the data, and in fact, as the matrix is partitioned block-wise, we have to merge the partial aggregates of the blocks, to compute the full aggregate over each row. On the other hand, this could be executed in the DataBag representation in a single pass, as we have tuple/row at-a-time processing. In a typical hash-partitioned join, the aggregate could be calculated while executing the join with a simple `mapper` after the join.

Caching Throughout Iterations. A holistic view over the program allows us to reason about the structure of the control flow. For instance, caching data which is used repeatedly within an iteration or along multiple control-flow branches, can result in great performance benefits. This becomes even more interesting when the data originates from pre-processing which would otherwise be re-computed for each iteration. The decision to cache is not always evident and forces the programmer to consider system specifics – Lara currently employs a greedy strategy which implicitly caches data used repeatedly in iterations.

4. RELATED WORK

ML Libraries & Languages. SystemML's DML [4] and Mahout's Samsara provide R-like linear algebra abstractions and execute locally or distributed on Hadoop and Spark. While Samsara has fixed implementations for its linear algebra operations, SystemML applies inter-operator optimizations like operator fusion and decides execution strategies based on cost estimates. As there is no support for relational operators, ETL has to be executed in a different system and there is no potential for holistic optimization. The Delite project [6] provides a compiler framework for domain-specific languages targeting parallel execution. Delite's IR is based on higher-order parallelizable functions (e.g., reduce, map, zipWith). However, Delite's IR does not allow to reason holistically about linear and relational algebra. In this work, we base our reasoning on types and on the holistic view of the AST. Finally, we believe that monad comprehensions provide a better formal ground for reasoning and applying algebraic optimizations.

Algebra Unifying Approaches. Kumar et al. [12] introduce learning over linear models on data residing in a relational database. They push parts of the computation of the ML model into joins over normalized data, similar to [7]. These works focus on generalized linear models, as we focus on more generic optimizations that can be derived directly from the common intermediate representation of linear and relational algebras . MLBase [11] provides high-level abstractions for ML tasks with basic support for relational operators. Their DSL allows the optimizer to choose different ML algorithm implementations, but does not take relational operators into account nor does it apply any optimization among algebras.

Acknowledgments. The authors thank Matthias Boehm (IBM Almaden) for his constructive feedback and discussions. This work has been supported by grants from the German Science Foundation (DFG) as FOR1306 Stratosphere, the German Ministry for Education and Research as Berlin Big Data Center BBDC (ref. 01IS14013A), the European Commission through Proteus (ref. 687691) and Streamline (ref. 688191), and by Oracle Labs.

5. REFERENCES

[1] A. Alexandrov, A. Kunft, A. Katsifodimos, F. Schüler, L. Thamsen, O. Kao, T. Herb, and V. Markl. Implicit parallelism through deep language embedding. In *ACM SIGMOD*, 2015.

[2] A. Alexandrov, A. Salzmann, , G. Krastev, A. Katsifodimos, and V. Markl. Emma in action: Declarative dataflows for scalable data analysis. In *ACM SIGMOD*, 2016.

[3] 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 *ACM SIGMOD*, 2015.

[4] M. Boehm, D. R. Burdick, A. V. Evfimievski, B. Reinwald, F. R. Reiss, P. Sen, S. Tatikonda, and Y. Tian. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *IEEE Data Engineering Bulletin*, 2014.

[5] P. G. Brown. Overview of scidb: Large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[6] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. *ACM SIGPLAN Notices*, 2011.

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

[8] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations und Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[9] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM TODS*, 2000.

[10] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on MapReduce. *ICDE*, 2011.

[11] T. Kraska, A. Talwalkar, J. Duchi, R. Griffith, M. J. Franklin, and M. Jordan. MLbase: A Distributed Machine-learning System. In *CIDR*, 2013.

[12] A. Kumar, J. Naughton, and J. M. Patel. Learning Generalized Linear Models Over Normalized Data. *ACM SIMGOD*, 2015.

[13] D. Maier and B. Vance. A call to order. In *ACM PODS*, 1993.

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *ACM SIGMOD*, 2008.

[15] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*. IEEE, 2010.