Optimizing End-to-End Machine Learning Pipelines for Model Training

vorgelegt von

Andreas Kunft, M.Sc.
ORCID: 0000-0001-7557-1703

von der Fakulät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
– Dr. rer. nat. –

genehmigte Dissertation

Promotionsausschuss:
Prof. Dr. Ziawasch Abedjan, Vorsitzender
Prof. Dr. Volker Markl, Gutachter
Prof. Dr. Matthias Böhm, Gutachter
Prof. Dr. Asterios Katsifodimos, Gutachter

Tag der wissenschaftlichen Aussprache: 16. August 2019

Berlin 2019

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor Volker Markl, for the excellent work environment and the collaborations under which this thesis was created. Volker introduced me to the academic world and gave me valuable insights throughout my time as a Ph.D. student. He provided me with innumerable opportunities to meet other researchers and exciting people.

I would also like to express my gratitude to Matthias Böhm and Asterios Katsifodimos for serving as my committee members.

During my time as a Ph.D. student, I was accompanied by many stellar colleagues. I would like to express my sincere gratitude to Alexander Alexandrov, Asterios Katsifodimos, and Sebastian Schelter. Alexander introduced me to the beauty of monads, which ultimately influenced the choice of my topic. Asterios always had my back and pushed me in times of doubt. Sebastian expanded my research interests and helped me to validate my thoughts. I would like to thank Lauritz Thamsen and Thomas Renner, who I could always talk to and which became my dear friends. I would like to thank Tilmann Rabl and Sebastian Bress for their constant advice and patience. Furthermore, I want to thank all my co-authors and colleagues at DIMA for their help and advice.

Finally, I would like to thank my family and friends, my girlfriend Katharina, and especially my parents, Marlies and Werner, who always supported me in any possible way.

ABSTRACT

Modern data analysis programs often consist of complex operations. They combine multiple heterogeneous data sources, perform data cleaning and feature transformations, and apply machine learning algorithms to train models on the preprocessed data. Existing systems can execute such end-to-end training pipelines. However, they face unique challenges in their applicability to large scale data. In particular, current approaches either rely on in-memory execution (e.g., Python Pandas and scikit-learn) or they do not provide convenient programming abstractions for specifying data analysis programs (e.g., Apache Flink and Spark). Moreover, these systems do not support optimizations across operations in these pipelines, which also limits their efficient execution. In this thesis, we present our contributions towards a more efficient execution of end-to-end machine learning pipelines for model training. In particular, we discuss the following three contributions:

In our first contribution, we propose a programming abstraction in the dynamic language R on top of the distributed dataflow engine Apache Flink. Our abstraction scales to large amounts of data but also hides system specifics of the dataflow engine. We integrate R and the system language Java in a shared runtime to alleviate the performance overhead of current solutions for language integration in case of complex user-defined functions. In our second contribution, we introduce an intermediate representation for data analytics programs, which allows us to optimize the complete training pipeline. We base our approach on a low-level intermediate representation that provides access to the whole program and augments two abstractions on top of it: a layer that unifies the application of user-defined functions enables operator fusion and pushdown; a high-level representation of the operations enables plan variant selection. In our third contribution, we propose a new context-aware operator for dataflow engines, which directly creates an efficient partitioning schema for matrices from normalized data sources. The operator decouples the evaluation of the join predicate from the materialization of the result. This minimizes data shuffling in distributed settings and enables the operator to choose the materialization strategy depending on the shape of the input relations.

In summary, we conclude that we need a holistic system design that covers all tiers – programming abstraction, intermediate representation, and execution backend – to overcome the scalability challenges of large-scale data analysis programs.

ZUSAMMENFASSUNG

Programme zur Datenanalyse vereinigen häufig komplexe Operationen. Sie verknüpfen verschiedenste heterogene Datenquellen, entfernen fehlerhafte Datenpunkte und transformieren deren Darstellung, bevor Modelle mit Verfahren des maschinellen Lernens auf diesen vorverarbeiteten Daten trainiert werden. Obwohl existierende Systeme solche sogenannten Ende-zu-Ende Pipelines ausführen können, stellt sie deren Anwendbarkeit auf großen Datenmengen vor neue Probleme und Herausforderungen. So sind bisherige Ansätze entweder auf die Ausführung innerhalb des verfügbaren Speichers limitiert (z. B.: Python Pandas und scikit-learn), oder es fehlen geeignete Programmierabstraktionen, um vollständige Analyse Pipelines auszudrücken (z. B.: Apache Flink und Spark). Weiterhin bieten diese Systeme keine übergreifenden Optimierungen an, welche die verschiedenen Operationen innerhalb dieser Pipelines einheitlich betrachten. Ausgehend von diesen Herausforderungen, stellen wir im folgenden unsere Forschungsbeiträge zur effizienteren Ausführung von Ende-zu-Ende Pipelines für maschinelles Lernen vor:

Innerhalb des ersten Beitrags stellen wir eine Programmierabstraktion für die dynamische Sprache R vor, welche auf dem verteilten Datenfluss System Apache Flink ausgeführt wird. Unsere Implementierung stellt eine benutzerfreundliche Abstraktion bereit, welche die Skalierbarkeit auf große Datenmengen ermöglicht und gleichzeitig die Komplexität des Datenflusssystems verbirgt. Dabei integriert unsere Lösung R und die Systemsprache Java in einer gemeinsamen Laufzeitumgebung. Hierdurch ist eine effiziente Ausführung möglich, welche insbesondere bei komplexen, benutzerdefinierten Funktionen eine Herausforderung darstellt. Innerhalb des zweiten Beitrags führen wir einen Zwischencode für die Repräsentation von Datenanalyseprogrammen ein, der es ermöglicht, über die komplette Analyse Pipeline zu optimieren. Unser Ansatz verwendet dazu drei verschiedene Darstellungsschichten: eine feingranulare Schicht ermöglicht den Zugriff auf alle Details des Programms – inklusive Daten- und Kontrollfluss – und dient als Basis für die zwei darüber liegenden Schichten. Die einheitliche Darstellung von benutzerdefinierten Funktionen ermöglicht deren Zusammenführung und effiziente Ausführung. Die Darstellung der einzelnen Operationen auf hohem Abstraktionsniveau ermöglicht die Selektion von äquivalenten Planvarianten. In unserem dritten Forschungsbeitrag stellen wir einen neuen, kontextsensitiven Operator für Datenflusssysteme vor, welcher Matrizen auf effiziente Weise aus normalisierten Daten erstellt – ein häufiger Teilschritt innerhalb von Ende-zu-Ende Pipelines. Der Operator entkoppelt die Evaluation des Join-Prädikates von der Materialisierung des Ergebnisses. Dies führt zur Minimierung der notwendigen Datenkommunikation und erlaubt eine von der Beschaffenheit der Daten abhängige Wahl der Datenmaterialisierung.

CONTENTS

1

INTRODUCTION

Requirements for data analytics have changed drastically over the last decade. Traditionally, companies used Extract, Transform, and Load (ETL) procedures to move raw data into data warehouses, and analysts focused on aggregation queries over this structured data, e.g., for reporting and online analytical processing (OLAP). The ever growing size of data and the increasing number of domains in which practitioners apply machine learning (ML), e.g., business intelligence [CCS12], recommendations [SKKR01], natural language processing [MMS99], and speech recognition [HAS+14], have blurred the lines between data preprocessing and analytics. Today, data scientists do *in-situ* analysis on dirty and unstructured data [R+11]: they prepare diverse data sources, such as log files and clickstreams, with relational operators and user-defined functions (UDFs) before they mine the preprocessed data with statistical and ML methods [ZKR16, BFG+17]. Preprocessing is an essential requirement that includes data cleaning and de-normalization, but also feature engineering and selection. The applied preprocessing methods depend on the respective algorithm used to train and evaluate ML models in so called *end-to-end machine learning pipelines* [STH+15, KMNP15, BBC+17, ZKR14]. These pipelines are refined multiple times to obtain the most promising configuration of features, ML algorithms, and hyperparameters [SHG+15].

Data scientists can choose from a variety of tools and languages to implement such end-to-end pipelines. The formerly dominant relational database management systems (RDBMSs) have limited support for complete pipelines: complex UDFs and iterative ML algorithms are hard to express in SQL [PPR+09]. Unstructured data must be loaded and transformed before analysis to benefit from efficient execution. Dynamic general-purpose programming languages (GPLs), such as Python and R, are a popular choice for exploratory data analysis [Smi17]. They are user-friendly and provide fast development cycles due to dedicated libraries for collection processing [McK12] and machine learning [PVG+11, T+15]. However, dynamic languages are designed for in-memory processing, which prevents out-of-the-box scaling to large amounts of data [HNP09]. The dedicated libraries optimize domains in isolation, which requires materialization of intermediate results [PTS+17]. General-purpose dataflow engines, such as MapReduce [DG04] and more recently Apache Spark [ZCF+10] and Flink [ABE+14], scale seamlessly to large numbers of homogeneous compute nodes. Their flexible application programming interfaces (APIs) are based on second-order functions to transform collections with arbitrary types using UDFs. The separation of program specification and execution enables out-of-core processing on dedicated execution engines. However, the implementation of ML algorithms in this collection-centric APIs is cumbersome and requires domain and system expertise. Therefore, ML algorithms are hard-coded by experts and provided as library functions [MBY$^+$16, DDGR07, LYF$^+$10]. Dedicated systems and libraries for distributed machine learning [GKP$^+$11] and more recently deep learning [CLL$^+$15, ABC$^+$16] provide convenient programming abstractions and efficient execution for ML. In contrast, they do not support general collection processing and thereby lack generality, e.g., to implement user-defined feature transformations.

In summary, all presented approaches are best suited to a particular area in practice. However, none of them addresses all requirements we deem crucial to implement and execute end-to-end machine learning pipelines for model training in an efficient manner: despite their close relationship, none of them examine preprocessing and ML holistically. Dedicated systems and APIs do not incorporate data exchange and format requirements of subsequent systems in their optimization decisions. This separation ultimately complicates the implementation of such training pipelines and prevents holistic optimizations over both domains.

1.1 Problem Description

End-to-end pipelines ML pipelines blur the lines between data preprocessing and analytics. Data sources and prerequisite transformations are determined by the particular feature set required by the ML algorithm and often need to be changed *ad hoc*. Thus, end-to-end pipelines interleave (*i*) relational operators to join data sources, (*ii*) UDFs for feature extraction and *vectorization*, and (*iii*) linear algebra operations for model training and evaluation.

Dynamic languages, such as Python and R, have gained much traction in recent years [IEE17, Smi17] and are the tools of choice for prototyping such pipelines. On the one hand, this is due to their language features: dynamic languages are easy to learn and require less *boilerplate* code (e.g., no typing and straight forward I/O) compared to typed system languages, such as C++ and Java. On the other hand, Python and R offer popular libraries for dataset manipulations (e.g., Pandas [McK12] and `dataframes` [Lan13]) and machine learning (e.g., scikit-learn [PVG$^+$11] and `matrices` [Lan13]). On the downside, dynamic languages and their libraries operate in-memory and cannot scale to large amounts of data out-of-the-box [PTN$^+$18]. The transition to systems that can cope with large amounts of data requires a system expert. Data scientists are often unfamiliar with the systems' native language and programming abstraction, which is crucial for achieving reasonable performance [AKK$^+$15].
To overcome this problem, databases and dataflow engines started to provide *guest language* extensions for dynamic languages, which build on familiar library abstractions [Smi17, VYL$^+$16]. While these approaches work well for predefined library functions and relational operators, custom transformations specified as UDFs in second-order functions, such as `map` and `reduce`, are not handled ideally: (*i*) UDFs are executed via inter-process communication (IPC) between

the system and an external guest language process. IPC introduces large performance overheads compared to the native API; (*ii*) source-to-source translation (STS) to the native API overcomes the performance disadvantages of IPC. STS only supports a subset of the guest language and thus, does not support general UDFs. Preprocessing in end-to-end ML pipelines for model training relies on non-trivial functions to transform data into a suitable form for model training. Examples are bag-of-words representations, such as *n-grams* [CT$^+$94], word-embeddings, such as *word2vec* [MCCD13], or conversion of categorical features by *one-hot-encoding* [HH10].

Systems should provide convenient APIs that hide system specifics and scale seamlessly to large amounts of data. The APIs should also support a rich set of language features and complex UDFs that do not degrade the systems' performance.

Dedicated libraries for collection processing and ML in dynamic languages provide a high abstraction level but are executed *as is*, without any further optimizations [GW14]. As described above, language integrations on dedicated systems mitigate some of the shortcomings, e.g., out-of-core processing and optimizations. End-to-end pipelines are composed of relational operators, UDFs, and linear algebra. Using multiple systems or libraries to express end-to-end pipelines enforces *staging* of intermediate results [PTS$^+$17]. Composing the whole pipeline within a single system is challenging. Typically, systems tightly couple their domain-specific language (DSL) and intermediate representation (IR) with their domain. RDBMSs enable the declarative specification of analytics over normalized data. They can integrate UDFs but their IR is based on relational algebra, and they do not consider external UDFs during optimization. Dataflow systems provide efficient collection processing and rich support for UDFs. Their type-based DSLs defer execution to build an operator graph as IR, which they use to parallelize the execution of second-order functions. This IR enables specific physical optimizations, such as choosing the implementation for join operators, but black-box UDFs prevent optimizations that require schema information, e.g., operator reorder and pushdown. In contrast to RDBMSs, the flexible *schema-on-the-fly* model of dataflow engines allows to retrofit ML algorithms as UDFs over collections, but again, black-box UDF prevent optimizations. Dedicated systems for ML optimize for efficient execution of linear algebra but only provide limited support for collection processing with UDFs, e.g., *transform* methods in SystemML[1] and Tensorflow.[2]

To enable holistic optimizations over end-to-end pipelines, an IR has to provide (*i*) *types for relational and linear algebra, e.g., to reason about physical operator implementations,* (*ii*) *white-box UDFs to enable operator fusion and pushdown, and* (*iii*) *control flow to optimize over iterative algorithms.*

Preprocessing, including de-normalization, data cleaning, and feature engineering, is commonly executed on row-wise partitioned data. Linear algebra operations can also be expressed over row-wise partitioned data [KNP15, CKNP17], but block-partitioned matrices allow to execute linear algebra operations more efficiently [GKP$^+$11, KTF09, HBY13, LHKK79]. Thus, in the

---

[1] https://apache.github.io/systemml/dml-language-reference.html#data-pre-processing-built-in-functions

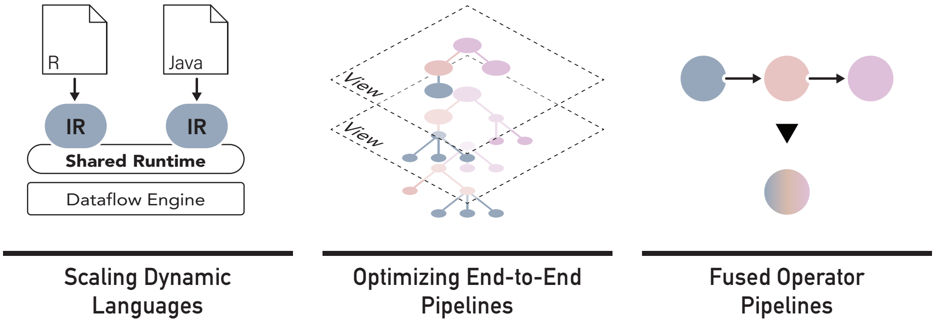
[2] https://www.tensorflow.org/tfx/transform

context of end-to-end pipelines for model training, systems have to combine both partitioning schemata and provide efficient conversion methods between them.

An execution backend should not only provide dedicated operators for both schemata, but also efficient conversions between them. Context-aware operators at the intersection of relational and linear algebra can prevent unnecessary staging and re-partitioning of intermediate results.

1.2 Contributions

End-to-end machine learning pipelines for model training change the requirements in all layers of systems design. Diverse data formats, in-situ processing, and the strong connection between preprocessing and model training introduce new challenges for the specification, optimization, and execution of such pipelines. In the following, we give an overview of our research contributions to each particular area.



Scaling Dynamic Languages – In Chapter 3, we introduce a `Dataframe` API in R, which is translated to dataflow programs that run on Apache Flink. The process is transparent to the user and combines a well-known API and language with efficient distributed execution on large scale data. In contrast to previous approaches, we focus on the efficient execution of arbitrary, complex UDFs where we achieve comparable performance to the systems native API. The content described in this chapter was published in [KSB+18].

Optimizing ML Training Pipelines – In Chapter 4, we propose an embedded DSL for the declarative specification of end-to-end training pipelines based on domain-specific types for collections and matrices. Our IR allows the system to examine the types for collections and matrices, control- and data-flow, and UDFs. Two additional abstractions on top of the IR enable various optimizations: (*i*) a functional view enables operator fusion and pushdown across type boundaries; (*ii*) A high-level view captures the semantics of both domains and optimized data access and validation of ML algorithms. The content described in this chapter was published in [KAKM16] and [KKS+19].

Fused Operator Pipelines – In Chapter 5, we propose fused operators at the intersection of linear and relational algebra operators. In particular, we describe a distributed join algorithm that produces block-partitioned matrices from normalized data. It uses an index-join to create the required matrix partitioning information without materializing the join result. Based on the partitioning information, our algorithm selects the most efficient block materialization strategy based on the shape of the normalized data. Thus, it avoids unnecessary data shuffling and is resistant to data skew. The content described in this chapter was published in [KKS⁺17].

1.3 Publications

This thesis is based on the following peer-reviewed publications:

- **Bridging the Gap: Towards Optimization across Linear and Relational Algebra**

  Andreas Kunft, Alexander Alexandrov, Asterios Katsifodimos, and Volker Markl. BeyondMR@SIGMOD 2016: 1

- **Blockjoin: Efficient Matrix Partitioning through Joins**

  Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Tilmann Rabl, and Volker Markl. PVLDB 10(13): 2061-2072 (2017)

- **ScootR: Scaling R Dataframes on Dataflow Systems**

  Andreas Kunft, Lukas Stadler, Daniele Bonetta, Cosmin Basca, Jens Meiners, Sebastian Breß, Tilmann Rabl, Juan Fumero, and Volker Markl. SoCC 2018: 288-300

- **An Intermediate Representation for Optimizing Machine Learning Pipelines**

  Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. PVLDB 12(11): 1553-1567 (2019)

Furthermore, the author collaborated on the following peer-reviewed related publications:

- **Implicit Parallelism through Deep Language Embedding**

  Alexander Alexandrov, Andreas Kunft, Asterios Katsifodimos, Felix Schüler, Lauritz Thamsen, Odej Kao, Tobias Herb, and Volker Markl. SIGMOD Conference 2015: 47-61

- **PEEL: A Framework for Benchmarking Distributed Systems and Algorithms**

  Christoph Boden, Alexander Alexandrov, Andreas Kunft, Tilmann Rabl, and Volker Markl. TPCTC 2017: 9-24

1.4 Outline

The remainder of the thesis is structured as follows: in Chapter 2, we give an overview of end-to-end ML pipelines for model training and explain the critical concepts and systems used throughout the rest of the thesis. Chapters 3, 4, and 5 describe our main contributions to enable efficient specification and execution of end-to-end machine learning pipelines. Each chapter also contains the relevant related work, a specific conclusion, and directions for future work. In Chapter 6, we revisit our contributions and discuss their limitations and drawbacks. Finally, we provide broader directions for future work.

2

BACKGROUND

There are multiple options to implement end-to-end machine learning pipelines, from libraries over general dataflow systems to dedicated systems. On the one hand, the amount of data that needs to be processed determines this choice heavily. On the other hand, there is also a strong correlation between the implementation of the DSL to specify pipelines and the opportunities to optimize the system, which ultimately determines the efficiency and amount of data that can be processed. This chapter introduces the necessary preliminaries relevant to all chapters. First, we give an introduction to end-to-end machine learning pipelines and describe the core components, namely *preprocessing*, *model training*, and *model evaluation* in Section 2.1. Next, we provide a timeline and overview of systems that have been proposed to implement such pipelines in Section 2.2. Starting with the motivation, architecture, and design principles of distributed dataflow systems, we introduce specialized systems that build and extend the concepts of dataflow systems. Finally, we showcase different approaches to implement DSLs and their respective connection to the described systems in Section 2.3.

2.1 End-to-End Machine Learning Pipelines for Model Training

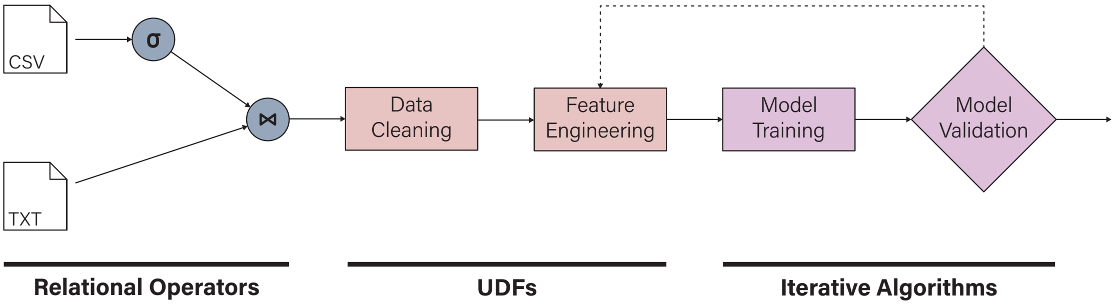


FIGURE 2.1: General schema of an end-to-end ML pipeline for model training.

End-to-end ML pipelines for model training combine data gathering and preparation, as well as model training and validation. To execute such a pipeline, we need to combine relational and linear algebra operators with UDFs and control flow. Figure 2.1 depicts these core components of a training pipeline. It combines multiple data items using relational operators, performs data cleaning and feature engineering with complex UDFs, and validates a trained ML model using iterative algorithms and linear algebra operators. Often, different feature sets and hyperparameters are evaluated to derive the best-suited model for prediction. In the following, we discuss these components in detail.

2.1.1 Preprocessing

The term *data preprocessing* describes the steps necessary to bring data in a suitable form for the model training performed in later stages of the pipeline. RDBMSs are well suited for denormalization and aggregation queries over structured data that already resides in the system, but (*i*) they provide limited capabilities for in-situ processing, (*ii*) they suffer from the restricted expressiveness of SQL [PPR+09] in case of UDFs. To exemplify these limitations, we examine the SQL query depicted in Listing 2.1, which combines two relations `Products` and `Reviews` and returns only those tuples that exceed a specific price.

```
SELECT p.rating, p.category, p.price, r.review
FROM   Products p, Reviews r
WHERE  p.product_id == r.product_id and p.price >= 100
```

Listing 2.1: Example SQL query to extract certain product reviews.

(*i*) In the context of end-to-end pipelines, one deals with diverse data sources and formats, such as semi-structured click-logs (e.g., in comma-separated values (CSV) files), unstructured text (e.g., websites or comments), and dense and sparse matrix formats (e.g., the MatrixMarket formats [BBR96] and libsvm [CL11]).

Ingesting such raw data formats in databases is usually done with ETL processes that perform the translation to the relational format, which is optimized for the expected query pattern. In exploratory data analysis pipelines, the executed data transformations and sometimes even the set of data points (i.e., the features set) changes depending on the ML algorithm. The feature set is a tuning parameter itself and static ETL processes delay model validation by data ingestion, especially for large amounts of data [BYM+14]. *In-situ* processing of raw data files, i.e., without having to import the data into the system, overcomes this problem. Often RDBMS allow queries over raw data but use it mostly for data ingestion. Distinguishing features of RDBMS, such as efficient indexing and statistics, are thereby unavailable [ABB+12].

(*ii*) Besides *in-situ* processing, preprocessing adds another new aspect to end-to-end pipelines. The so-called *feature transformations* bring the data in a suitable form for the ML algorithms. The values that are returned by our example SQL query are not yet in a suitable form. Most ML algorithms require normalized, numerical features.

Thus, we require algorithms to convert categorical features [SMYT14] to numerical features (cf. Listing 2.1, the `category` attribute), normalize [Gru15] numerical features (cf. Listing 2.1, the `price` attribute), and bring text into vectorized form [CT+94] (cf. Listing 2.1, the `review` attribute). Analysts have to implement such complex algorithms as external UDFs in RDBMSs. In contrast, UDF-centric dataflow engines and `dataframe` libraries provide APIs and UDF support in GPLs, as discussed in Section 2.2. A common format to describe feature transformations are pipelines of `fit` and `transform` methods, adopted by scikit-learn [PVG+11] and

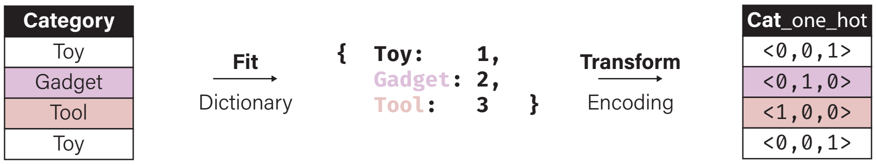


FIGURE 2.2: *Fit* and *transform* step for *one-hot-encoding*.

MLlib [MBY+16].[1] In the initial `fit` phase, the necessary intermediate data is gathered, e.g., aggregates and statistics of the values, before the actual feature transformation is applied based on the gathered data in the `transform` phase. For example, *one-hot encoding* [HH10] creates a sparse vector representation from categorical values, as depicted in Figure 2.2. In the fit phase, the algorithm builds a dictionary that maps each distinct value in a column c to a sequential index. In the transform phase, one-hot-encoding replaces the categorical values with sparse vectors of length $|c|$ that have a 1 at the position corresponding to the index of the category and a 0 elsewhere. It is important to note that the dictionary is not only required to transform the features for training, but also to transform the test data for model validation. Another common transformation is *feature scaling*. It scales numerical features to a specific range, e.g., *normalization* [Gru15] rescales the values to have zero mean and standard deviation of one – as in a standard normal distribution. This is very important for ML algorithms that use gradient-based methods to achieve fast convergence, such as logistic regression [HJLS13] and support-vector machines (SVMs) [SS02].

2.1.2 Model Training

After the data is in a suitable form, a particular ML algorithm is trained based on the required use case. The community distinguishes between *supervised* and *unsupervised* algorithms depending on the data they use for training, as depicted in Figure 2.3.

Supervised Learning. Algorithms that require annotated or *labeled* data for training (i.e., data points for which the correct answer is known) are *guided* in their learning process. For example, to predict the cost of houses based on their features, e.g., size and neighborhood, a supervised learning algorithm is trained with a dataset of houses and their corresponding prices. Within the class of supervised learning, we can distinguish between *regression* and *classification* algorithms. *Regression* is used to predict continuous outputs, such as the previously mentioned cost of a house. A widespread example is linear regression. In practice, Data Scientists often train supervised learning algorithms using iterative optimization algorithms. Gradient-based methods are prevalent, as they are easy to implement and computationally not complex.
They often add a *regularization* term to their loss function. Regularization adjusts the impact of certain features to avoid *overfitting* towards the training data [MRT18]. Two important methods are the L_1 (e.g., in lasso regression) and L_2 norm (e.g., in ridge regression).
Analysts use *classification* to predict the particular instance of a finite set of labels or classes a data point belongs to. The set of possible labels can be two (e.g., to predict whether an email is

---

[1] https://scikit-learn.org/stable/data_transforms.html

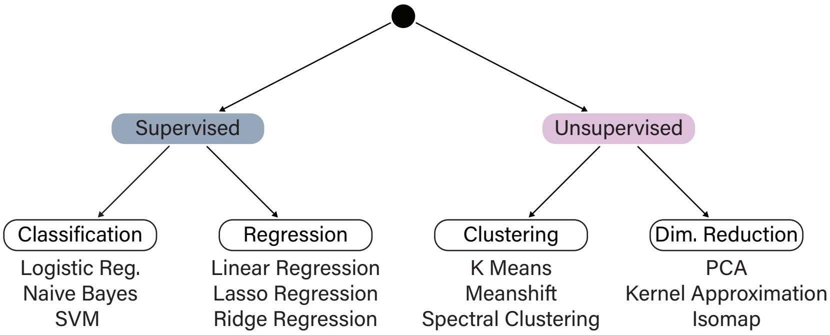


Figure 2.3: Classification of ML algorithms.

or is not spam). In which case, the problem is called binary classification. Logistic regression is a well-known algorithm for binary classification. Even though there are more evolved algorithms, data scientists often use logistic regression in practice due to its simplicity and convex cost function. If we expand the set of possible labels to more than two, e.g., to detect characters, we call the problem *multi-class* classification. Common methods to solve multi-class classification include multinomial logistic regression, decision trees, ensemble learning such as random forest, and deep learning based approaches [MRT18]. It is also possible to apply binary classification algorithms by using the *one-vs-all* method [Aly05].

Unsupervised Learning. Algorithms that detect patterns or structure without guidance by a pre-annotated dataset for training are called unsupervised learning algorithms. The nowadays widely discussed idea of *general artificial intelligence* would be an instance of unsupervised learning in the most general case. Apart from this somewhat hypothetical application, analysts often use unsupervised learning in the two bound problem spaces of *clustering* and *dimensionality reduction*. Cluster algorithms assign data points to clusters without any prior structural knowledge. A typical example is the k-means algorithm [JD88], which assigns points to k centroids based on a distance metric. After selecting random data points as initial centroids, the algorithm iteratively recomputes the centroids and the data points that belong to them. Dimensionality reduction algorithms are used to reduce the number of features in a dataset, e.g., to reduce the feature space and detect dependencies with principal component analysis (PCA) [Jac05].

2.1.3 Model Evaluation

Model evaluation is critical to determine whether an ML model is capable of making reliable predictions on unseen data. There is a multitude of options to tune for, including the ML algorithm itself, its hyperparameters, and the set of features used to train it. For example, let's take a regression model trained by linear regression. We have several options to choose from: firstly, we have to decide on a particular regularization, e.g., lasso or ridge. Secondly, we have to pick hyperparameters for the algorithm, e.g., the regularization parameter λ or the intercept. Finally, we have to decide on a set of features to use, e.g., by feature selection with PCA.

A popular method to validate a particular combination of hyperparameters and features for supervised learning algorithms, such as the described linear regression, is k-fold cross-validation

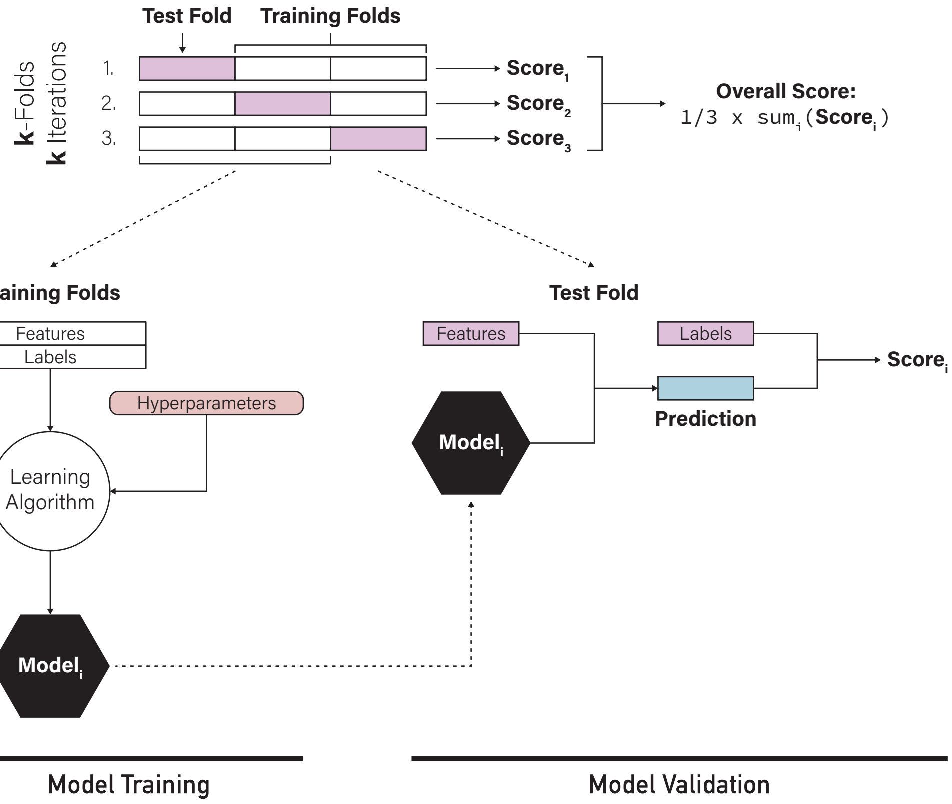


FIGURE 2.4: Schema of 3-fold cross-validation. The figure is based on [Ras18].

(CV) [Koh95]. Figure 2.4 depicts its general idea. The algorithm divides the dataset and the corresponding labels horizontally into k disjoint splits. Then, it executes the algorithm under evaluation k times. In each of the k iterations, CV uses another single split as the test set, and the remaining $k - 1$ splits as the training set. The upper part of Figure 2.4 depicts this process for $k = 3$. In each iteration, CV trains a model based on the current training set and hyperparameters (cf. Figure 2.4, bottom left). After training, CV validates the model based on the test set. These predictions can now be compared with the actual labels from the test split to calculate a *score* for the current iteration (cf. Figure 2.4, bottom right). Standard measures for classification are *precision*, *recall*, and the *F1* score, while the *mean absolute error* and the *mean squared error* are common for regression analysis [Mur12]. CV calculates the overall score as average overall scores of a model for the different training and test set combinations. Based on the overall score, different hyperparameters can be tested to perform a so-called *hyperparameter-tuning*. In the most basic form, data scientists perform a grid search over the space defined by the set of hyperparameters. More sophisticated methods use pruning techniques and heuristics, e.g., Bayesian optimization and multi-armed bandits to reduce the search space [BB12].

2.2 Distributed Dataflow Systems

Even though research on parallel RDBMSs [DGG⁺86, DG92] was already adopted commercially in the 1980s [DS08], they were not considered to be well suited for the demands of web-scale data processing [Bre05]. In addition to the strong *ACID* guarantees, parallel databases provide consistency (C) and high availability (A) in terms of the CAP-Theorem for distributed data stores. In scenarios where mostly read-only queries are executed, these strong consistency guarantees, i.e., commits are atomic across the entire distributed system, have been dropped in favor of partition tolerance.

For example, the novel way to calculate the *page rank* [PBMW99] of web pages introduced by Google uses the link structure of web pages. It required fast sequential read access on unstructured data to build what is essentially an inverted index over the web pages crawled. This approach does not fit well in the architecture of traditional data management systems, which require structured data and therefore in this scenario, costly data ingestion. This mismatch ultimately led to the development of the Google File System (GFS) [GGL03] and MapReduce (MR) [DG04] – distributed storage and processing for shared-nothing clusters of thousands of nodes with commodity hardware. Since then, these concepts have been adopted in open-source implementations and inspired the development of several new systems for large scale data.

2.2.1 Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) [B⁺08] is an open-source implementation of the GFS and part of Apache Hadoop, a collection of open-source software.[2] It is designed to store large scale data on large clusters of commodity hardware. This results in the following assumptions and goals: (*i*) the system is highly fault-tolerant. Due to commodity hardware, it treats hardware failures as the norm, not as an exception; (*ii*) applications are expected to access data in a streaming fashion – the system does not support random-access. Thus, it provides high throughput but no low latency; (*iii*) applications are expected to have a *write once – read-many* access pattern to the data stored in HDFS. Modifications to existing data are not possible.

HDFS implements a driver-worker architecture for shared-nothing clusters. A dedicated driver node, called *NameNode*, is responsible for the file system namespace and manages opening, closing, and renaming of files and directories. The NameNode has a transaction log, called *EditLog*, that persistently records every change to the file system metadata, e.g., file creation or renaming. The file system namespace is persisted in a file called *FSImage*. The worker nodes, called *DataNodes*, are responsible for storing the actual data in fixed-size blocks (64MB per default). HDFS uses these blocks as the basis for replication and distributes them among different DataNodes. With the default replication factor of three, HDFS replicates a block on a different physical node in the same rack and on a node in a different rack to handle failures or network outages of full racks. Blocking and replication enable concurrent read access. While the NameNode validates file access and manages user rights, the DataNodes handle all read and write operations on the data blocks. So, even though HDFS tolerates network partitioning for

---

[2] https://hadoop.apache.org/

the DataNodes, the NameNode must be reachable in order to acquire the block locations upon a read request and thus, is the single point of failure in the system.[3] HDFS prioritizes local access to blocks, but the actual access method (e.g., network or disk) is transparent to the user.

2.2.2 Hadoop MapReduce

Next to HDFS, Apache Hadoop also provides an open-source implementation of *MapReduce*. It is a parallel dataflow engine with a static execution model that processes UDFs based on the semantics provided by the second-order functions `map` and `reduce`. MapReduce has a shared-nothing architecture and facilitates data-parallel execution by reading data blocks in parallel from a distributed file system (e.g., HDFS). Therefore, its worker nodes typically run the same nodes as HDFS to benefit from local data access.

MR represents data as key-value pairs (k, v), where k and v can be arbitrary types. For instance, it reads a text document as $(Integer, String)$ pairs, where the key represents the offset and the value the corresponding line in the document. The semantics of the `map` second-order function ensure that the UDF is executed independently on each key-value pair of the input and emits 0, 1 or n key-value pairs per input: $map : (k_1, v_1) \rightarrow list(k_2, v_2)$.[4] Thus, the number of key-value pairs defines the theoretical upper bound for the degree-of-parallelism (DOP). In practice, MR consumes all key-value pairs of a block in a single task and thus worker node.

After the map phase, MR shuffles the output: it partitions the key-value pairs by their keys (MR uses hash-partitioning by default). The tasks that execute the `reduce` function fetch all key-value pairs for their particular key. This pull-based mechanism ensures fault-tolerance, a new task can process a partition if the original task fails. The user-defined aggregate function (UDAF) of the `reduce` function receives a key-value pair $(k_2, list(v_2))$ with all values that share the same key and returns a list of aggregated values: $reduce : (k_2, list(v_2)) \rightarrow list(v_3)$. The number of distinct keys k_2 emitted by the `map` functions thereby defines the theoretical maximum for the degree of parallelism.

The programming model of MR enables distributed execution without writing parallel code. The user provides sequential implementations for `map` and `reduce`. The framework takes care of the distributed execution. The expressiveness of the high-level general-purpose programming language for the UDFs eases implementation of complex analytical queries, e.g., pattern-matching and document clustering. Thus, Hadoop represents an open-source analysis system for user-defined function-centric tasks on schema-less data. MR has been the subject of many discussions. Critics doubt the novelty and usefulness of the proposed model, and they described ML as a *major step backwards* [DS08]. They use (parallel) RDBMS for comparison and complain about the absence of indexes, the missing declarative programming model, and the schema-less design of MR. Despite all the criticism, MR became immensely popular and initiated the development of a new generation of dataflow systems that address its shortcomings.

---

[3] Since version 3.x, NameNodes can be replicated to eliminate the single point of failure.

[4] Actually, these are the semantics of a `flatmap` function, but MR does not provide a dedicated `map` function, which has exactly one output kv-pair per input kv-pair.

2.2.3 Beyond MapReduce

As the field of applications expanded, the shortcomings of MRs became more prevalent. Its static computation model is limited to a `map` followed by an `reduce` function, which requires multiple, successive MR jobs for complex analytics. The absence of relational operators forces users to hard-code them as UDFs (e.g., joins) and, therefore, prevents optimizations. Iterative algorithms, commonly found in ML and graph algorithms, have to stage intermediate data in the distributed file system.

These limitations led to the development of dataflow systems with support for arbitrary, directed acyclic graph (DAG) structured operator graphs. Apache Flink [CKE+15, ABE+14] extends the available operators in MR by relational operators, e.g., `join` and `filter`. Flink applies established database optimizations in the context of dataflow systems, e.g., physical operator selection for joins. Flink's API is based on the *DataSet* abstract data type that represents a distributed collection. A dedicated operator for iterations avoids re-deployment of iterative programs and efficient caching of intermediate results [ETKM12]. Apache Spark [ZCF+10] provides a fluent API to express transformations on distributed collections, called resilient distributed dataset (RDD) and tracks data provenance to compute iterative algorithms in memory.

In addition to the evolution of the execution backends of dataflow engines, their programming abstractions evolved with regards to the application domains. The API of Spark mimics collection transformations in functional languages and shaped the *look and feel* of today's dataflow engines for general processing. Flinks API offers a special tuple element type (with fixed arity and typing of the fields), together with an extended set of operators. This enables a more declarative specification of the execution pipelines, e.g., explicit join operator vs. user-defined join logic in MR. Spark also added a typed `DataSet` API, which introduces a schema for its elements. The `DataFrame` represents a special DataSet with named columns as elements. Its design closely follows the API of the Pandas library from Python. The demand for declarative, relational-like DSLs with first-class support for UDFs on large-scale data initiated the development of specialized APIs, systems, and file formats. *Pig Latin* [ORS+08], Hive [TSJ+09], Impala [KBB+15], and SparkSQL [AXL+15] extend dataflow systems with SQL-like APIs to query data maintained by system catalogs. Specialized file-formats for semi-structured data, such as RCFile [HLH+11] and Parquet [Voh16], store nested data in columnar formats organized in row groups and enable filter-pushdown and indexing. Systems, such as Pregel [MAB+10] and GraphLab [LGK+12], are used to explore and analyze data structured in graphs, i.e., *follower graphs* in web-applications. Their vertex-centric APIs provide vertices and edges as first-class constructs to specify graph analysis and exploration algorithms, e.g., connected components and node rankings. The research and open-source community proposed several libraries to ease the development of machine learning algorithms on dataflow systems [MBY+16, SVK+17]. These libraries provide ML algorithm implementations on top of dataflow systems and relieve users from the error-prone task of expressing ML algorithms as transformations over collections. SystemML [BDE+16, GKP+11] provides an R-like DSL to specify ML algorithms as linear algebra over matrices and vectors. SystemML optimizes programs logically and physically before it executes them either on a single node, distributed on a dataflow engine or in a hybrid execution mode. The execution backend depends on the size of the input and intermediates, as well as the sparsity of the data. Even though modern, general-purpose dataflow systems, such as Flink and Spark, support iterative algorithms, they follow a bulk synchronous parallel (BSP) execution model to ensure consistent state: a new iteration step is only started when all parallel instances of the previous iteration step are finished. Therefore, the slowest parallel working instance determines the overall execution time of an iteration step (or *epoch*). This synchronization barrier after each epoch leads to degradation of the execution time in case of *stragglers* (i.e., worker instances that execute significantly slower than an average worker, e.g., due to skewed data). GraphLab [LGK$^+$12], a framework for distributed graph processing, alleviates these shortcomings by an asynchronous execution model, but lacks elasticity and fine-grained fault-tolerance. Hogwild! [RRWN11] overcomes this problem by introducing asynchronous processing in the particular case of stochastic gradient descent, and the authors prove that the algorithm still converges. The parameter server architecture [LZY$^+$13] generalizes asynchronous computation in driver-worker architectures for efficient distributed ML algorithms. Worker nodes *own* certain parts of the data, while the driver nodes maintain the global state, e.g., a machine learning model. Worker nodes communicate with the driver nodes asynchronously to receive the latest, globally aggregated state, i.e., the model. SystemML implements the parameter servers model on top of Spark (with BSP and *asynchronous parallel* execution model) for mini-batch ML algorithms. Mxnet [CLL$^+$15] and TensorFlow [ABC$^+$16] are the latest evolution towards efficient large-scale machine learning with a focus on deep neural networks. The previously described parameter server inspired their computation model, which enables updates to shared models and parameters. Their APIs combine state and operator definition in an operator graph, which they use to execute on CPU, GPU, and specialized devices, such as Tensor Processing Units (TPUs) [JYP$^+$17].

2.3 Language Implementation Approaches

The implementation approach of DSLs for libraries and systems heavily influences the range of applicable optimizations. Next to standalone implementations, programmers often embedded DSLs as *guest language* on top of an already existing *host language*. Thereby, the existing infrastructure of the host language, such as its compiler infrastructure, can be leveraged, which reduces the implementation effort and enables users to take advantage of existing developer tools of the host language, e.g., integrated development environments (IDEs). The embedding approach directly determines the expressiveness and optimization capabilities of the DSL.

2.3.1 Shallowly-Embedded Libraries

Many general-purpose programming languages provide dedicated libraries for different domains, such as collection processing and ML. In particular scripting languages such as Python and R, provide popular libraries for data processing. Prominent examples are Python's Pandas [McK12] and scikit-learn [PVG$^+$11]. These libraries are embedded *shallowly* [GW14] in the host language (e.g., Python) and executed *as is*, i.e., without further optimizations except for those hard-coded in the algorithms or performed by the GPL interpreter/compiler.

2.3.2 Type-Based DSLs

Type-based DSLs use operations defined on a type (e.g., the `RDD` in Spark) to construct an operation graph, rather than directly executing the operations. Specific operations trigger the *evaluation* of the built operator graph, which then can be analyzed and optimized before the actual execution. Examples are the APIs of dataflow engines, such as Spark [ZCF+10] and Flink [ABE+14], as well as ML systems, such as MXNet [CLL+15] and Tensorflow [ABC+16]. Type-based DSLs enable several optimizations on the operator graph, which is treated as IR for optimization. For example, logical `join` operators can be replaced by the best-suited physical implementation, e.g., hash or sort-merge joins, and specific operators can be chained together. However, the IR only reflects the operators defined on the type. Guest language UDFs specified in second-order functions, such as `map` and `reduce`, are generally treated as black-boxes.[5] Black-box UDFs prevent operator reordering, such as filter pushdown. The system can not determine whether attributes are modified in a UDF and therefore not ensure correct semantics for operator reorders. To provide more semantics to the optimizer, type-based DSLs often allow developers to specify user-defined logic in a restricted subset of the host language, e.g., via strings as `ds.filter("a < 10")` in Flink. Such a restricted language for UDFs enables optimizations, such as filter-pushdown, but limits the expressiveness to the defined language subset and prevents detection of type errors and typos at compile time.

Furthermore, control flow constructs, such as `loop` constructs and `if` statements, are not visible in the IR. Thus, the optimizer cannot reason about control flow, which prevents certain optimizations, such as caching, compression, and partitioning of loop invariant data, because each iteration de facto builds a new operator graph. Specialized loop constructs address this problem partially [YAB+18, ETKM12], as the DSL can represent them in the operator graph. They still hinder linguistic reuse [RAM+12] of the language's native control flow.

2.3.3 Quotation-Based DSLs

While shallow-embedded and type-based DSL can be implemented on any GPL, quotation-based DSLs [NLSW16] rely on *meta programming* capabilities of the guest language [SBO13, TS97] to access the Abstract Syntax Tree (AST) that represents a program. Examples for quotation-based DSLs are LINQ [Mei11], LMS [RO10], and Squid [PVSK18].

Quotation allows reusing the syntax and type system of the host language and gives access to the program's AST during compilation and runtime. In contrast to the previously presented approaches, quotation gives access to the entire AST of the GPL, including types, UDFs, and control flow. The AST can then be altered and optimized before execution.

As the AST is too detailed and inefficient for domain-specific optimizations, DSLs typically provide an IR with a higher abstraction level that reflects the domain and types of the user-facing API. Such an IR enables optimizations based on domain-knowledge while providing control flow and white-box UDFs. Thus, quotation-based DSLs overcome the limitations of the previously presented approaches, but require a careful design of domain-specific IRs [AKM19].

---

[5] A notable exception is the detection of read-write sets at the byte-code level [HPS+12].

2.3.4 Standalone DSLs

Standalone DSLs, such as *SQL* and SystemML's *DML* [GKP+11] overcome the problems stated above. They provide a full-fledged compiler infrastructure, and the optimizer can reflect all language features. However, a programmer has to develop this infrastructure (together with libraries and development tools) from scratch, and support for complex UDFs written in GPLs is hard to achieve [PPR+09]. Another major problem is user acceptance: it is hard to convince programmers to learn and adapt to a new language. Often, standalone DSLs try to ease the transition by staying close to the syntax of an existing language, e.g., R in case of SystemML.

3

SCALING DYNAMIC LANGUAGES

Extracting value from data is a very important, but complex task. Typically, data analysts rely on complex execution pipelines composed of several stages, (e.g., data cleansing, transformation, and preparation) that need to be executed before the actual analysis or machine learning algorithm can be applied [ZKR16]. Often, these pipelines are repeatedly refined to obtain the best-suited subset of data for prediction and analysis. Therefore, programming languages with rich support for data manipulation and statistics (provided as library functions), such as *R* and *Python*, have become increasingly popular [IEE17]. More recently, such languages also started receiving increased attention in other domains such as enterprise software ecosystems [Smi17]. While these languages are convenient for non-expert programmers, they are typically designed for single-machine and in-memory usage. Thus, they run out of memory if data exceeds the available capacity and cannot scale-out without significant implementation efforts.

In contrast, parallel dataflow engines, such as Apache Flink [ABE⁺14] and Spark [ZCF⁺10], are able to handle large amounts of data. However, data scientists are often unfamiliar with the systems' native language and programming abstraction, which is crucial to achieve good performance [AKK⁺15]. To overcome this barrier, dataflow engines provide additional programming interfaces in *guest languages*, such as R and Python, which build on familiar abstractions, e.g., *dataframes*. Current state-of-the-art solutions integrate guest languages in two fundamental ways. They either use inter-process communication (IPC) or source-to-source translation (STS).

Inter-process communication: In this approach, the guest language runtime runs in a separate process. Input and output data has to be exchanged via IPC between the process running the dataflow engine and the process running the guest language. IPC supports any valid code in the guest language but can incur major performance overhead in the form of *data exchange* between the processes and *serialization* to a format readable by both languages.

Source-to-source translation: In this approach, guest language code is translated to host language code, e.g., to the dataflows' native API. While STS achieves competitive performance, as the translation happens before program execution, it is limited to a restricted set of functions and library calls. Support for a rich set of language features would require a full-fledged compiler.

The impact on the execution time for both methods is demonstrated in Figure 3.1, by comparing SparkR [VYL⁺16], which supports STS and IPC. In this case, the execution of a simple UDF

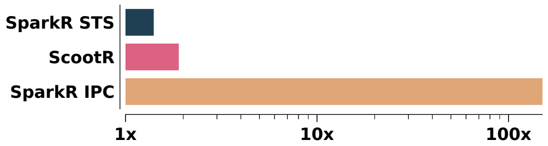


Figure 3.1: R function call overhead compared to the native execution on the dataflow system. STS and IPC are compared to Spark. Our approach, called ScootR, is compared to Flink.

via IPC is more than 100× slower compared to STS.[1] Thus, current approaches either yield sub-optimal performance or restrict the set of usable guest language features.

Research Contribution. In this chapter, we introduce ScootR, a novel language integration approach based on an efficient IR for both the guest and the host language.[2] We focus on the execution of UDF heavy pipelines – the bottleneck in current state-of-the-art solutions – and provide a dataframe-centric R API for transformation, aggregation, and application of UDFs with minimal overhead. Using a common IR, ScootR avoids the data exchange and serialization overheads introduced by IPC. ScootR extends on STS by using the existing compiler infrastructure and back-end of the host language to support a rich set of language features and pre-compiled modules. ScootR is based on a tight integration of the fastR [SWHJ16] language runtime with the Java Virtual Machine (JVM) responsible for executing Flink data processing pipelines. fastR is a GNU-R compatible R language runtime based on the Truffle [WWS⁺12] language implementation framework and the Graal dynamic compiler [WWS⁺12, WWW⁺13] for the JVM. Thus, ScootR efficiently executes a rich set of R UDFs within the same runtime and completely avoids IPC. By harnessing Truffle's efficient language interoperability system, ScootR accesses Flink data types directly inside the R UDFs, avoiding data materialization and unnecessary data copying due to marshalling. Our experiments show that ScootR achieves comparable performance to STS and outperforms IPC based approaches by up to an order of magnitude while supporting a rich set of language features. Analytics pipelines written in ScootR can either be executed on a single local machine, utilizing multi-threaded execution or distributed in a cluster, using both intra-node multi-threading and inter-node parallelism.

In summary, we make the following contributions:

- We present a new integration technique that enables seamless, low-overhead, interoperability between the fastR R language runtime and the Flink dataflow engine. Our approach avoids the overhead of IPC and serialization present in state-of-the-art solutions.
- We describe how we enable efficient exchange and access of data structures between fastR and Flink with minimal overhead and why it is necessary to achieve good performance.
- We compare our implementation in an experimental study against the current state-of-the-art, as well as native execution in R and fastR.

---

[1] The full benchmark is discussed in detail in Section 3.3.2.

[2] This chapter is based on [KSB⁺18].

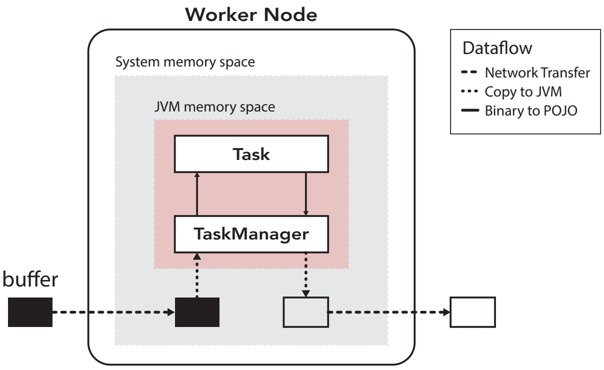


Figure 3.2: Data buffer flow in Apache Flink.

3.1 Background

In this section, we provide the necessary background to the systems used in ScootR. First, we describe the language interoperability features of Truffle that we use to achieve efficient data exchange between R and the Flink execution engine. Second, we explain the available options for language integration in detail.

3.1.1 Graal, Truffle, and FastR

Truffle [WWS+12] is a language implementation framework. It is used to develop high-performance language runtimes by means of self-optimizing AST interpreters. These ASTs collect profiling information at runtime and specialize their structure accordingly. Examples for such specializations include elision of unnecessary type conversions as well as the removal of complex method dispatch logic. Truffle provides interoperability features to efficiently exchange data and access functions between languages build on top of it [GSS+15].
Graal [WWW+13] is a dynamic compiler that has special knowledge of Truffle ASTs and can produce highly-optimized machine code by means of (automatic) partial evaluation: as soon as a Truffle AST self-optimizes itself by reaching a stable state, Graal assumes its structure to be constant and generates machine code for it. De-optimizations and speculation failures are handled automatically by Graal by transferring execution flow back to the AST interpreter.
fastR is a high-performance GNU-R compatible R language runtime implemented using Truffle that relies on the Graal dynamic compiler for runtime optimization. It is open-source, and is available as one of the default languages of the GraalVM multi-language runtime [GSS+15, WWW+13]. GraalVM can execute Java applications and Truffle-based language runtimes such as JavaScript, Ruby, Python, and LLVM on top of the HotSpot JVM [PVC01].

3.1.2 Guest Language Integration

In this section, we discuss different approaches to call R code from within Java. While we focus on integrating R in a dataflow engine, the presented approaches apply to other programming languages as well. In the following examples, we concentrate on the task of evaluating a UDF,

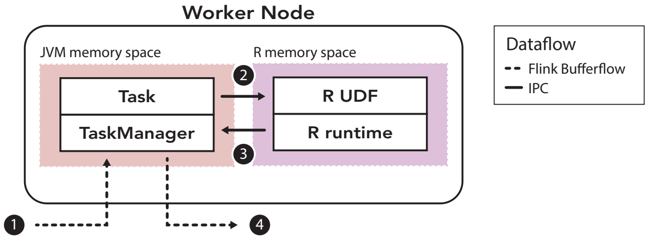


Figure 3.3: IPC between Java and an external R process.

written in R, within a worker node of a dataflow engine, e.g., Apache Flink or Apache Spark. Figure 3.2 depicts the general flow of a data buffer in Flink, which contains the elements to process. UDFs are executed in *Tasks* of the Flink *TaskManager*, which resides on a worker node. Binary data buffers arrive over the network in the operation system memory space. Java's network layer copies the binary data buffer to the JVM memory space (i.e., the managed JVM heap). The binary data has to be de-serialized to *plain old java objects* (POJOs) before it is fed to the Task that executes the UDF. Flink sends the output in the very same sequence over the network, only in reversed order. In the following descriptions, we focus on the data exchange between Java and R, and neglect the process of data de-serialization.

Inter-Process Communication

In the IPC approach, the worker node sends and receives elements from an R process, which evaluates the function within the native R interpreter as depicted in Figure 3.3. The numbers show the data flow of the tuples during execution. The approach introduces three drawbacks: (i) The data in Java has to be serialized to a format suitable for exchange and deserialization in R. (ii) Additional communication overhead is introduced, as data is exchanged either through a (local) socket or a (memory-mapped) file, shared between the two communicating processes. (iii) In resource restricted environments, Java and R have to compete for the available memory, due to their isolated address spaces. Despite the presented drawbacks, IPC is used by several systems [YZP14, GHR$^+$12, DSB$^+$10], as it only requires basic I/O facilities.

Source-to-Source Translation

STS tackles the problem from a completely different direction as the previously presented approach based on IPC. Instead of exchanging data between the processes, the execution of R code is avoided altogether by executing a (semantically-equivalent) translation of the UDF to a programming language natively supported by the dataflow engine.

As an example, Figure 3.4 shows how the R source code of a user-defined function is translated to equivalent Java source code, *before* the actual execution of the dataflow program takes place. Once the translation is done, there is no interaction with R during program execution and STS translation offers native performance. STS is often restricted to a domain-specific subset of the guest language (e.g., library functions that can be directly translated to the target language) to reduce the implementation effort. Extensive support of guest language features essentially requires a full-fledged compiler and yield a huge effort.

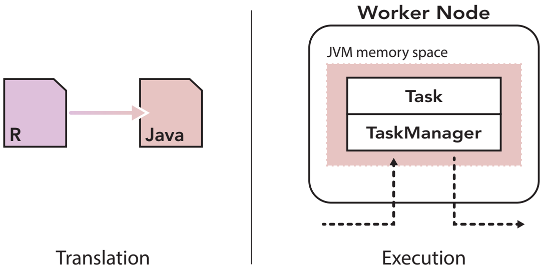


Figure 3.4: STS of R to Java, followed by the execution in the native dataflow API.

Hybrid Approach

The R integration in Apache Spark [ZCF+10], called SparkR [VYL+16], builds on a hybrid approach, combining STS and IPC. R language constructs that can be directly mapped to Spark's native dataframe API are source-to-source translated, as described in the previous Section. These constructs are limited to a subset of filter predicates (e.g., `>`, `<`, `=`, etc.), column manipulations and transformations (e.g., arithmetic operators, string manipulations, etc.), and library function calls. For instance, in the following example, an R `filter` function selects all tuples in the dataframe `df` that have the value `"english"` in their `language` column:

```
df <- filter(df, df$language == "english")
```

The R `filter` function can be translated to the following `filter` operator in Spark's native Scala dataframe API, including the user-defined predicate:

```
val df = df.filter($"language" === "english")
```

To run arbitrary UDFs, the user can specify functions on partitions of the dataframe, analogous to the `apply` function in R, and grouped data for aggregation. Here, STS cannot be used anymore and SparkR falls back to the previously presented approach based on IPC (cf. Section 3.1.2). Thus, SparkR combines both presented approaches. It achieves near native performance for a subset of operators via STS, but falls back to IPC in case of general user-defined functions.

Common Intermediate Representation

To avoid IPC while supporting a rich set of language features, one can define a common IR for both languages. The IR is then interpreted and/or just-in-time (JIT) compiled on a common runtime or compiled to machine code, as depicted in Figure 3.5. Implementing such a compiler is a big effort. Translating high-level languages to an existing compiler infrastructure reduces this implementation effort, increases portability, and facilitates the reuse of compiler back-end components, e.g., to generate efficient machine code through a Virtual Machine (VM). Prominent examples are the Java Virtual Machine, which uses byte code as IR, and LLVM [LA04], which uses, e.g., bitcode.

As described in Section 3.1.1, the GraalVM provides Truffle, a language implementation framework. Languages implemented in Truffle are automatically optimized and JIT compiled by the Graal compiler. GraalVM runs on the HopSpot runtime and therefore, can run and access Java seamlessly. In the next section, we describe how ScootR uses GraalVM to provide efficient execution of R code within the worker nodes of dataflow systems.

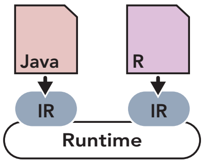


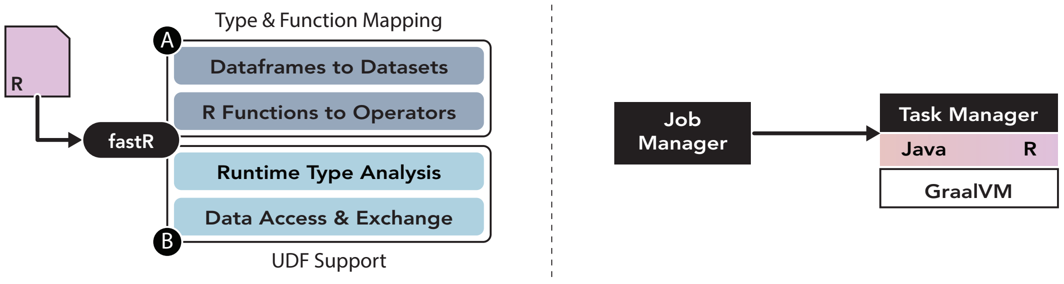
FIGURE 3.5: A common runtime for the intermediate representations of both languages.

3.2 ScootR

In this section, we describe our approach to execute R code in Apache Flink. We first provide an overview of all the components in general, before we discuss each step in detail. The focus of Lara is the efficient execution of UDFs, as they introduce a big performance overhead in currently available solutions (cf. Section 3.1.2).

3.2.1 Overview

We base our approach on fastR, the R implementation on top of the GraalVM. As introduced in Section 3.1.1, GraalVM is a language execution runtime capable of running *multiple* languages – including R and Java – in the same virtual machine instance. GraalVM enables seamless interoperability between all of its supported languages and provides efficient *language interoperability* [GSS^{+}15] capabilities. ScootR builds on such capabilities to expose Flink's internal data structures to the fastR engine. We distinguish between two main phases: the *plan generation phase* and the *plan execution phase*. Figure 3.6 details the components of each phase.



Plan Generation Phase

Plan Execution Phase

FIGURE 3.6: The two main phases in ScootR.

In the plan generation phase, described in Section 3.2.2, ScootR builds a Flink operator plan from R source code, which is later executed by the dataflow engine. Similar to Flink's native APIs, the dataframe API of ScootR is evaluated lazily. Calls to the API trigger no execution, but build a Flink operator graph until a *materialization* point – a sink in the graph – is reached. Section 3.2.2 Ⓐ explains the necessary steps for the *Type and Function Mapping*. ScootR defines the correct mapping of R dataframes to Flink's DataSet abstraction. Based on this mapping, R functions are translated to their corresponding Flink operators as defined by ScootR's function implementations in fastR. We detail the necessary steps to enable efficient execution of UDFs

in Section 3.2.2 🅑 (*UDF Support*). First, we show how ScootR determines the result types of UDFs via runtime type analysis. Second, we describe how ScootR achieves efficient data exchange between Java and R and why it is necessary to provide access to Flink's underlying data structures.
After the operator plan is created, it is deployed on the Flink cluster and executed during the plan execution phase described in Section 3.2.3. R UDFs are executed in parallel by each worker node. ScootR's integration with the GraalVM ensures that each UDF is automatically optimized by the Graal JIT compiler.

Running Example. Listing 3.1 gives an example of an R application, which makes use of the ScootR dataframe API. We use it as running example throughout the rest of this Section. In Line 1 – 2, we specify the Flink cluster we execute on and its DOP. In Line 4 – 8, we read an input file and convert it to a dataframe. Next, we project the `flight_id` and `miles` columns (cf. Line 10) and create a new column `km` by applying the UDF in Line 11. Finally, we retrieve the first five entries of the dataframe in Line 12.

```
1   flink.init(host, port)
2   flink.parallelism(dop)
3
4   df    <- flink.readdf(
5              "hdfs://some/input/file",
6              list("flight_id", "distance", ...),
7              list("integer", "integer", ...)
8            )
9
10  df    <- flink.select(df, flight_id, miles)
11  df$km <- df$miles * 1.6
12  df$head(5)
```

Listing 3.1: Code snippet for the running example in ScootR.

3.2.2 Plan Generation Phase

🅐 **: Mapping R Data Frames to Flink DataSets.** Dataframes are a popular abstraction to represent tabular data in languages such as Python and R and used in many libraries. As ScootR's API is built around dataframes, it is crucial to provide a valid and efficient mapping from an R dataframe to a data type suitable for processing in Flink. While Flink can work with arbitrary Java data types, it provides special facilities for instances of its `Tuple`N type, where N specifies the tuple's fixed arity. The fields of a tuple are typed and can be compared to a row in a database table or an entry in a dataframe. Thus, we can define a mapping from an R dataframe df with N columns and types $t_1, t_2, \ldots, t_N$ to a Flink dataset ds with element type `Tuple`N<$t_1, t_2, \ldots, t_N$>. As individual dataframe columns can be accessed either by index or name, we maintain a mapping of the dataframe column names to their respective tuple indexes in our dataframe wrapper class for the dataset.

Ⓐ : Defining R Functions for Flink Operators. During lazy evaluation, an R program using ScootR's API is translated to a Flink operator plan. To generate such a plan from the R source code, ScootR introduces new Truffle AST nodes (called *RBuiltinNode*) that correspond to new built-in functions available to the R user. Some important functions in ScootR are depicted in Table 3.1. Listing 3.2 shows a snippet for the `flink.select` built-in function used in Line 10 of our running example in Listing 3.1. The specification of Truffle nodes relies on annotations, while the actual code for the AST nodes is generated during compilation by the Truffle framework. The `flink.select` built-in expects a dataframe *df* and a variable length argument (indicated by three dots) representing the columns to project (cf. Line 2). The behavior of the node is defined by methods annotated with *@Specialization*. For example, the behavior of the `flink.select` node in our snippet is defined in the `select` method in Line 5. It extracts the projected columns and adds the according Flink `project` operator to the execution graph.

```
1  @RBuiltin(name           = "flink.select",
2            parameterNames = {"df", "..."})
3  abstract class FlinkSelect extends RBuiltinNode.Arg2 {
4    @Specialization
5    DataFrame select(DataFrame df, RArgsValuesAndNames fields) {
6      // determine projected columns
7      // add Flink `ProjectOperator` to Execution Plan
8    }
9  }
```

Listing 3.2: Simplified snippet of the `flink.select` RBuiltin.

Ⓐ : Functions without User-Defined Code. R functions that do not involve user-defined code are directly mapped to their counterpart operators defined by the Flink dataset API. For instance, the `flink.select` function from the running example (cf. Listing 3.1, Line 10) is directly mapped to the `project` operator from Flink's DataSet API, as described in the previous paragraph. ScootR performs the entire mapping of R functions without user-defined code during the plan generation phase and, therefore, they introduce no runtime overhead.

Ⓑ : Runtime Type Analysis. R does not require explicit type ascription for UDFs. In contrast, Flink requires the input and output types of operators to build the operator plan. While the *container* type is fixed to `Tuple`N, the arity N and the types of the fields may change when the UDF is applied. Thus, ScootR needs to execute the R UDF to determine its result type before the corresponding Flink operator, which calls the function at runtime, is created. Since the data might reside in a distributed file system, such as HDFS [SKRC10], we avoid taking a sample of the actual data to determine the data types due to performance considerations. Therefore, the current implementation requires to specify the data types in the R API when reading files (cf. Listing 3.1, Line 7). The result types of all other operators in the pipeline are determined automatically by ScootR. The result type of non-UDF operators is defined by their semantics. In the case of UDF operator, the R function is executed during the plan generation phase, while the operator graph is created. We instantiate temporary tuples with field values based on the

Table 3.1: Examples from the ScootR API.

Function	Example	Description
flink.select	`flink.select(df, x = COL1, COL2)`	Project (and rename) the specified set of columns
←	`df$new ← (COL1 / COL2) * 0.1`	Create (Override) column by applying the UDF on each row

flink.apply	`flink.apply(df, func)`
	`flink.apply(df, key = COL1, func)`
		Apply `func` on each row.
		Group by `COL1` column and apply `func` on each group

flink.groupBy	`max ← flink.groupBy(df, ’COL1’)`	Group by `COL1` for further aggregation, e.g., `max`
flink.collect	`fastr_df ← flink.collect(df)`	Collect a distributed dataframe `df` on the driver

runtime type inference of the previous operator, call the function with them, and thereby determine the result type of the UDF. Thus, ScootR keeps track of the current tuple type until the operator graph is built. In case the UDF does not return results for the temporary tuple used (e.g., it requires a specific number range), ScootR throws an exception during compilation and requests an explicit type annotation.

🅑 : Data Access and Exchange. An important aspect in ScootR is the efficient access to Java data types in R and vice versa. As we operate in the context of dataflow engines, the R UDFs are on the hot path and get called for each processed data item in the worst case, e.g., for the `map` operator. Thus, efficient data exchange *and* access between Java and R is crucial. Figure 3.7 depicts the data flow during the execution of a Flink operator. The unoptimized data flow is shown on the left side of Figure 3.7. Even though ScootR avoids data exchange due to the shared runtime, it still has to apply type conversion and introduces materialization points. The right side depicts the optimized version. Due to the direct access of Java types in R (and vice versa), as well as access to Flink's abstract data types, we avoid type conversion and materialization. In the next paragraphs, we show how ScootR achieves these optimizations.

❶ : **Java to R**. In the context of dataframes, efficient access to the processed elements means fast access to Flink Tuples (representing *rows*) and their fields (representing *columns*). ScootR distinguishes operators by their expected input – single or multiple tuples per function call:

Single Tuple: The first case are tuple-at-a-time operators, e.g., `map` or `flatmap`. A naïve solution is to determine the columns that are accessed in the UDF and to expose them as explicit function arguments. This is achieved, by *wrapping* the UDF in an R function which expects the column values required by the UDF as arguments. For example, the ← `apply` function from Listing 3.1, Line 11, expecting one argument for the values of the *miles* column, is wrapped into following function: `function(miles) miles * 1.6`
In general, multiple columns are accessed in the UDF and their values have to be extracted in a loop before being passed to the R function in the naïve solution. To avoid this scenario and be able to call the function directly with the tuple instance, ScootR makes use of the Truffle language interoperability features, a message-based approach to gain access to foreign objects internals, called *dynamic access* [GSS+15]. It enables a guest language (e.g., R) to efficiently access objects from another language (e.g., Java) running in the same GraalVM instance.

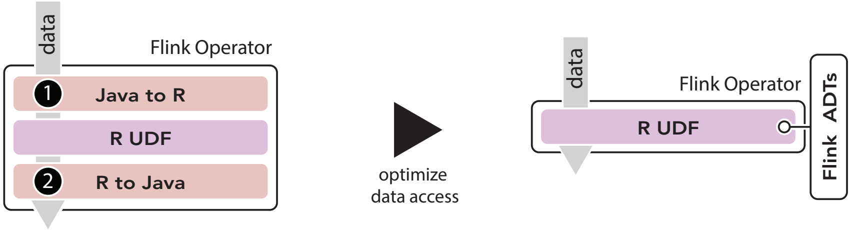


Figure 3.7: Dataflow of an Flink operator without (left) and with (right) optimizations.

Access to foreign objects is based on language-independent messages, e.g., *read* or *write*, which an object needs to implemented with corresponding methods. For instance, the $Read(t_{rec}, t_{id})$ message provides access to fields or elements t_{id} of an object or array t_{rec}.[3] Truffle replaces foreign object access (e.g., to a Java object) in the AST of the guest language (e.g., R) with a language-independent message node, which is ultimately replaced by the message implementation provided by the foreign object. ScootR integrates Flink's tuple type in the Truffle framework and thus, directly passes tuples as arguments to R functions, which can access their fields as they would be dataframe columns.

Multiple Tuples: The second case are operators that expect multiple tuples per function call, e.g., a `mapPartitions` operator. Flink provides access to all tuples expected by the function (e.g., all tuples contained in a partition) by an iterator. Using the aforementioned interoperability features, we provide direct access to the Flink iterator in the R UDF. As the iterator itself returns tuples, ScootR can access them directly as described before. Without this optimization, ScootR would need to materialize all tuples contained in the partition before passing them to the R UDF, e.g., as an `RList`. Therefore, it would introduce a pipeline barrier in the normally streaming execution, as all tuples have to be materialized before the R function can be called.

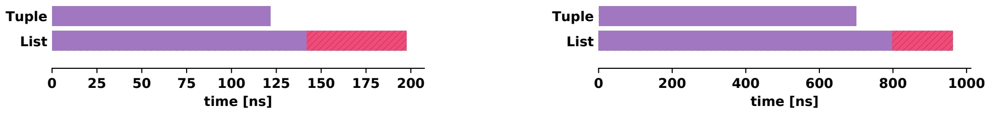
❷ : **R to Java**. Likewise, an R UDF returns results that are passed back to the calling Flink operator for further processing in the operator pipeline. Therefore, ScootR also needs an efficient mechanism to access results of an R UDF in Flink. The R return type has to be handled differently depending on the higher-order operator that calls the R function:

Single Value: The simplest type is a `map` operator that returns exactly one value per input tuple. ScootR guarantees this case by the semantics of the ← `apply` function (cf. Table 3.1). In this case, a new tuple is created after the R function execution, either appending a new column or replacing an existing one with the new values.

Single Vector: In the general `apply` function (cf Table 3.1), the UDF returns a vector of length N, which represents the output tuple (aka. row in a dataframe). Since fastR provides wrappers for all primitive type lists in R, the result vector can be accessed with similar methods as provided

---

[3] Truffle keeps arrays from managed languages in the JVM heap to trace their references and restricts pointer arithmetics in unmanaged languages (e.g., C).



(a) Tuple2 **(b)** Tuple19

Figure 3.8: Creating a Flink tuple in the R UDF vs. creating the tuple from an `RList` in Java. Purple depicts the time spent in the function call, pink the time for type conversion.

by Java's `List` type[4]. While these wrappers grant access to the values in Java, we still have to convert the R vector to a Tuple$N$ for further processing in Flink. To avoid this type conversion, ScootR provides built-in functions (cf. Section 3.2.2 Ⓐ) to create Flink tuples directly in the R function. Thus, instead of returning an R vector, the function is rewritten to create and return instances of the corresponding tuple type directly using the built-in functions. Figure 3.8 shows the execution time of a general `apply` function that does nothing except returning a small (cf. Figure 3.8a) and a large tuple (cf. Figure 3.8b). We can observe that the function execution (purple bars) itself is about 15 percent faster when we create the Flink tuples directly in the R function. In addition, when an R list is returned, it still has to be converted into the equivalent tuple class, indicated by the pink bars in Figure 3.8. Overall, ScootR achieves 1.75× better performance by instantiating the tuples directly in the R UDF in this micro-benchmark.

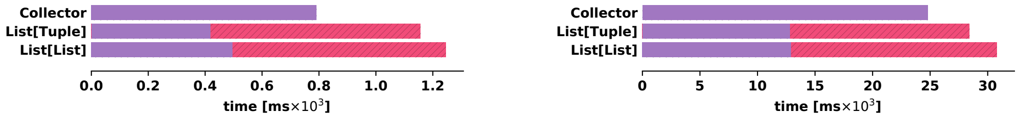
Multiple Values: Finally, ScootR needs to handle the case where higher-order functions return multiple values per input tuple, e.g., the `flatmap` operator. To this end, Flink provides an additional `Collector` class as argument, which *collects* the results of the function. Again, we provide direct access to the `Collector` from R. This avoids returning a list containing the results of the UDF, which would require an additional pass over the results to insert the values into the `Collector` in Java. Figure 3.9 shows the time to execute a `flatmap` operator returning a list of lists (the inner lists representing the tuples), a list of tuples, and finally directly using the `Collector` class in the R function. The function just returns 3 tuples with arity 2 (cf. Figure 3.9a) and 20 tuples with arity 19 (cf. Figure 3.9b) for each input tuple. We can observe that ScootR achieves 1.3× speedup when using the `Collector` directly. Interestingly, the function call takes almost twice as long using the `Collector`. This is due to increased complexity, as the collector stores the output using Flink's internal buffer management in the function call. Returning a list, the tuples have to be inserted after the function execution as depicted by the pink bars.

3.2.3 Plan Execution Phase

After ScootR successfully generated the operator graph for the pipeline, it forwards it to the JobManager, which schedules its execution. During this process, the JobManager also sends the *serialized* R code to the responsible TaskManagers. The ScootR operator implementations evaluate the R UDFs upon their first use. Since a TaskManager can execute the same function simultaneously in its available task slots, ScootR caches the code and shares it between exe-

---

[4] R lists are backed by a Java array and provide constant-time random access.



(a) Tuple2, 1:3 **(b)** Tuple19, 1:20

FIGURE 3.9: `flatmap` using Flink's `Collector` directly, returning an `RList` with `RList` as elements, and returning a `RList` of Tuples. Purple depicts the time spent in the function call, pink the time for type conversion.

cutions. Initially, the UDF is interpreted, however, as it gets *hot*, the JIT compiler will produce an efficient compiled version of it, which is executed for every subsequent call. The result of the job can either be directed to an output file or the user can collect it on the driver node via the `flink.collect` operator. If it is collected, ScootR passes the result as a dataframe to fastR, which can then be used for further local processing.

3.2.4 Illustrative Rewrite for the Running Example

Figure 3.10 shows the succession of R functions used and their corresponding Flink operators to represent the running example from Listing 3.1. Only the `apply` function includes user-defined code which has to be called and executed at runtime. All other functions can be replaced with the corresponding Flink operators during the plan generation phase. In the following, we describe the necessary modifications to the ← `apply` function before job execution.
Since the UDF is executed on every row in the example dataframe, a Flink `map` operator is generated. To determine the result type of the function, we execute it during the plan generation phase with a $Tuple2_{in} : (long, long)$ instantiated with random instances $(1.1, 0.3)$, based on the field types defined by the previous operator. The operator then calls the R function during the execution of each input tuple and has the following signature:

$$Tuple2_{in} : (long, long) \mapsto Tuple3_{out} : (long, long, long)$$

The additional field in the return value results from the extension of the dataframe with the *km* column (cf. Line 11 in Listing 3.1). Furthermore, given the mapping from column names to tuple indexes, the access to the `miles` column is replaced with a tuple index access:[5]

```
function(tuple) tuple[[2]] * 1.6
```

3.2.5 Implementation

We implemented ScootR in Flink without any changes to its existing codebase. Thus, it benefits from new features introduced by new versions of Flink. All functions of ScootR's API are represented via `RBuiltin` nodes. ScootR does not alter R UDFs, and relies on Truffle to perform optimizations such as dead code elimination and common subexpression elimination. In addition,

---

[5] The tuple fields indexes are 1 based in R.

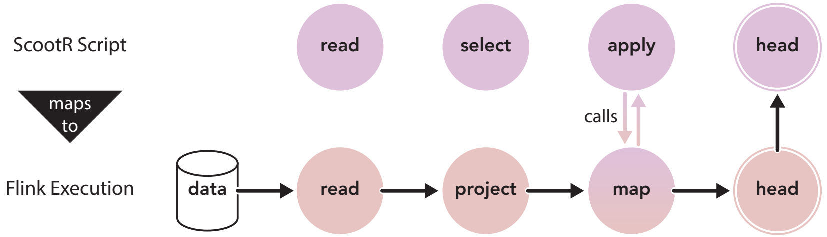


Figure 3.10: Mapping an R script (cf. Listing 3.1) to the corresponding Flink execution plan.

all internal data structures that are accessible inside R UDFs are provided as `TruffleObjects`. This enables, e.g., direct access to the Java tuples and their fields in R, without data exchange, as described in Section 3.2.2 Ⓑ § *Data Access and Exchange*.

Library Support. R packages are one of the reasons for R's popularity and used very frequently, e.g., for statistics and ML. Many packages use the native language extensions of R to call C implementations internally to achieve good performance. fastR implements the C API of GNU-R using Graal's native function interface [GRS$^+$13] and therefore can execute packages and external libraries that use the natural language extensions. While this works for most of the popular packages, some rely on GNU-R internals (e.g., data structures of the interpreter), which complicates the integration in fastR. fastR is continuously improved and more packages are added, which are then directly available in ScootR too. Project Sulong [RGM16] creates a Truffle AST from LLVM [LA04] bitcode, i.e., it provides LLVM as yet another language on top of the Truffle framework. Libraries using the native language extensions can be compiled to LLVM bitcode. Thus, the fastR AST and the AST of the native language extensions bitcode can exploit the dynamic access capabilities of Truffle (cf. Section 3.2.2), which alleviates crossing the language barrier from R to C.

3.3 Evaluation

In this section, we compare ScootR against the previously presented approaches by evaluating both micro-benchmarks and operator pipelines using real-world datasets.

3.3.1 Experimental Setup

Cluster Setup. We conducted our experiments on a four-node cluster. Each node features an Intel E5530 processor (2.4GHz, 8 cores), and 24GB main memory. The nodes are connected via a 1GBit Ethernet connection. We used Spark v2.2.0, Flink v1.3.1, and Hadoop v2.7.1 for our distributed experiments. Furthermore, we use GNU-R v3.2.3 [T$^+$15], the latest versions of fastR[6] and Graal[7] available while conducting the experiments, and JVMCI v0.33, based on the JDK v1.8.0_141. We execute each experiment 7 times and report the median time with error bars.

---

[6] https://github.com/graalvm/fastr, commit: 72b868a

[7] https://github.com/graalvm/graal, commit: 7da41b3

Datasets. We used two real-world datasets for our evaluation. The first dataset is the *Airline On-Time Performance Dataset*[8], which is also used to evaluate SparkR [VYL+16] and dplyr [W+17]. It contains JavaScript Object Notation (JSON)-formatted arrival data for flights in the USA with detailed information such as departure time, origin and destination, etc. We cleaned the data and reduced it to 19 columns per record (many of the original dataset columns contain no entries for 99.9% of the rows). As parsing JSON infers a high overhead in dataflow systems [LKC+17], we converted the dataset to the CSV format. The resulting file size, containing data from the years 2005 – 2016, is 9.5GB. The second dataset is the *Reddit Comments*[9] dataset, which contains JSON entries of user comments on the news aggregator website *www.reddit.com*. In addition to the actual text of the comment, it contains further meta-information, such as the author name, up and downvotes, category, etc. Similarly to the first dataset, we cleaned and converted the data to CSV in a preprocessing step. The raw data is provided as separate files for each month and we use the first 4 consecutive months starting from 2016. Each month amounts to roughly 33GB of raw data, resulting in about 14GB per month for the CSV used as input.

Benchmark Overview. We evaluated our approach for single and multi-node execution, comparing against GNU-R, fastR, and SparkR. We also compare SparkR and ScootR against the standard dataflow APIs in the system language, which we call *native APIs*.
First, we conducted a set of micro-benchmarks for *(i)* operators without user-defined code (e.g., `select`), and *(ii)* operators with user-defined code (e.g., `map` and `flatmap`). Here, we also compare the execution of SparkR with STS against the IPC approach. The goal of this set of micro-benchmarks is to highlight the benefits from the efficient execution of UDFs in ScootR. Second, in order to show the relevant performance impact of efficient UDFs in the context of real-world applications, we evaluated benchmarks consisting of operator pipelines. To this end, we chose to evaluate an ETL (i.e., preprocessing) pipeline on the airline dataset proposed by Oscar D. Lara et al. [YZP14],[10] a MapReduce-style aggregation pipeline, and a mixed pipeline with a multi-threaded ETL phase and successive, single-threaded model training in R.

3.3.2 Micro-Benchmarks

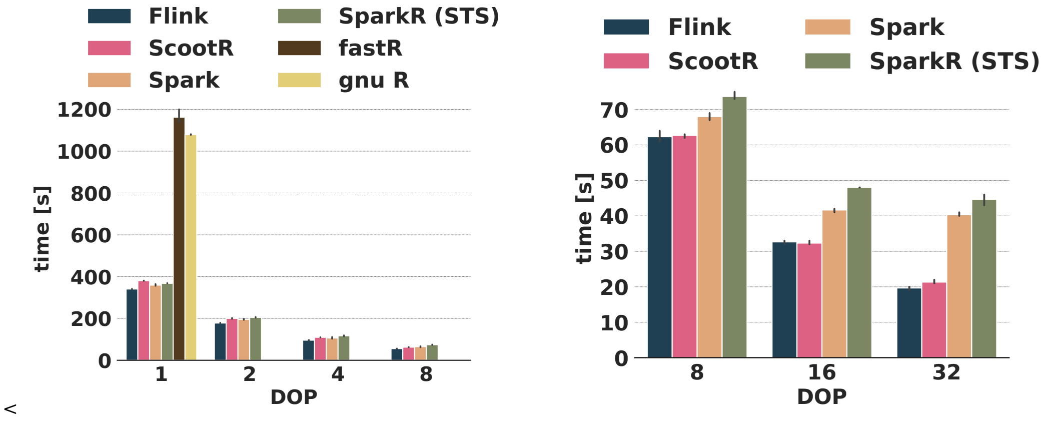
In this section, we present micro-benchmarks for several operators in isolation. For non-UDF operators, both ScootR and SparkR achieve almost the same performance compared to the native dataflow API. This is expected, as the operators can be translated before execution. Compared to standalone GNU-R and fastR, SparkR and ScootR are up to 20× faster on a single node (using 8 cores) and up to 46× for distributed execution (4 nodes × 8 cores).
The micro-benchmarks for operators with user-defined code show that ScootR and SparkR with STS achieve almost equal performance compared to their respective native API. It is important to note that ScootR achieves this even though the R UDF is executed. Benchmarks using IPC in SparkR, and thereby executing the R UDF, reveal its performance drawbacks as it fails to execute on the airline dataset within the set experiment timeout of 4 hours. Experiments on 10% of the

---

[8] https://www.transtats.bts.gov/Tables.asp?DB_ID=120

[9] http://files.pushshift.io/reddit/comments/

[10] The pipeline reports the maximum arrival delay per destination for flights from NY.



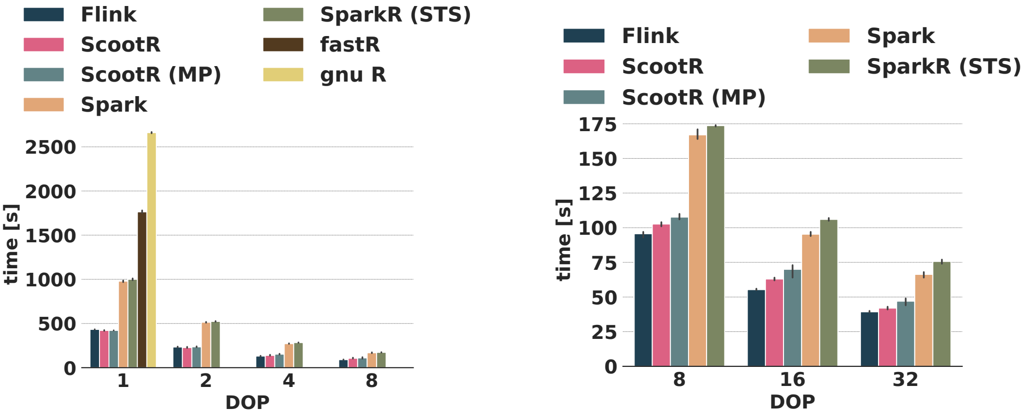
(a) Single node **(b)** Cluster

Figure 3.11: Micro-benchmark for a single `select` function.

original data show up to a magnitude slower execution times for SparkR with IPC compared to ScootR. The benchmarks also show the importance of direct access to internal data-structures to avoid additional cost due to materialization barriers. The benchmarks for 1:N output operators, e.g., `flatmap`, verify our assumptions that direct access to the Flink `Collector` class in R yields comparable performance to the native API.

Non-UDF Operator. In this first micro-benchmark, we project three columns from the airline dataset and write the result to a file. Figure 3.11a depicts the results on a single node and Figure 3.11b on the four nodes cluster. *SparkR (STS)* reflects the result for SparkR with source-to-source translation.
As expected, SparkR (STS) and ScootR achieve almost the same performance as the R `select` function is mapped to the `project` operator of the native APIs of the benchmarked systems. SparkR (STS) is about 1.15× slower than native Spark and ScootR about 1.13× slower than Flink. In contrast to GNU-R and fastR, which materialize the input data in memory before applying the `select` function, Flink and Spark stream the input data directly to the `project` operator. This results in a speedup of about 3× for both SparkR and ScootR compared to GNU-R for single-threaded execution. With increasing DOP, the speedup increases further to about 20× on a single node with DOP 8 (cf. Figure 3.11a) and up to 46× for the fully distributed execution on 32 cores (cf. Figure 3.11b). This result is expected, as the project operator is embarrassingly parallel. Interestingly, fastR is by a factor of 1.06× slower than GNU-R. We attribute this behavior to a more efficient implementation of the `read.csv` function in GNU-R.

UDF Operator with 1:1 Output. In this micro-benchmark, we compare the execution of an ← *apply* function similar to the one in the running example (cf. Line 11 in Listing 3.1). It multiplies the *distance* column by a constant factor and appends the result as a new column to the dataframe. The function is executed in ScootR via a `map` operator, as detailed in Section 3.2.4. SparkR (STS) uses source-to-source translation.
Figure 3.12a depicts the result of the benchmark on a single node. Both SparkR (STS) and ScootR



(a) Single node **(b)** Cluster

Figure 3.12: Micro-benchmark for the apply function from Listing 3.1, Line 11.

achieve almost the performance of their respective implementations in the native APIs. ScootR is at most 1.15× slower than Flink, while SparkR (STS) is about 1.07× slower respectively. These results are expected for SparkR (STS), as the UDF is translated to the native API. For ScootR, these results validate our expectations that we can achieve comparable performance to the native API even though the R function is executed in a `map` operator. GNU-R is outperformed by fastR (1.5×), and by both SparkR (STS) (up to 15×) and ScootR (up to 25×). Again, this is mainly due to the *streaming* facilities in Spark and Flink. In contrast to the previous benchmark, fastR is able to outperform GNU-R due to the more efficient execution of the `apply` function. Figure 3.12b depicts the same experiment, but now we distribute the computation on up to 4 nodes. Again, ScootR (1.1× for a DOP of 32) and SparkR (STS) (around 1.08× for a DOP of 32) introduce only a small overhead compared to their respective native APIs.

To determine the cost of IPC, we implemented the UDF using the `dapply` function of SparkR, which internally executes the UDF in a `mapPartitions` operator. For a fair comparison, we implemented the UDF using the general `apply` in ScootR, shown as *ScootR (MP)*, which internally also uses a `mapPartitions` operator. SparkR (IPC) failed to execute the function within the set experiment timeout of 4 hours for DOPs up to 32. In comparison, ScootR (MP) is competitive (around 1.1× overhead) to the ← `apply` function, due to direct access to Flink's data structures. To obtain results for SparkR (IPC), we sampled the airline dataset from 9.5GB down to roughly 100MB. Figure 3.13 shows the results for single-node execution with increasing DOP using the down-sampled airline dataset. For single-thread execution, SparkR (IPC) takes ∼50 minutes to complete the task compared to 30 seconds for Spark (STS). Using the 8 available cores, SparkR (IPC) executes in ∼7 minutes. Both versions of ScootR are about 1.8× slower than native Flink, while SparkR (IPC) is about 170× slower than native Spark. This performance overhead is due to the drawbacks of IPC discussed in Section 3.1.2, namely *serialization* and *data transfer*. In addition, the `dapply` function in SparkR (IPC) uses Spark's `mapPartitions` operator to execute the UDF. The operator provides all tuples contained in the partition via an iterator to the UDF. As SparkR cannot provide access to the iterator in the R UDF, all tuples

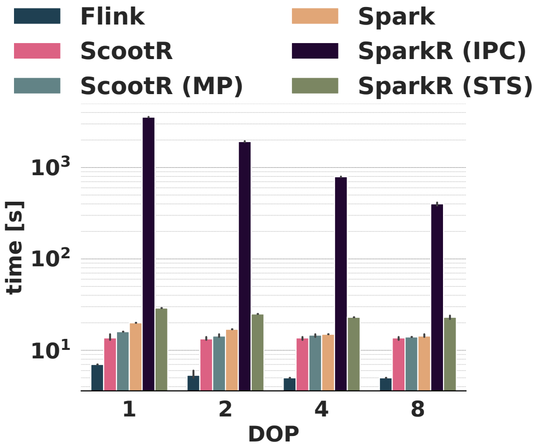


FIGURE 3.13: Single node micro-benchmark for the apply method from Listing 3.1, Line 11 with a 10% sample of the original data. The y-axis is in log scale.

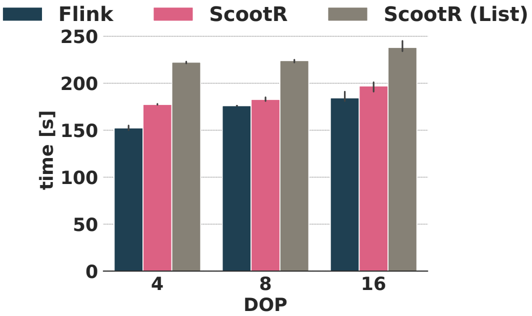
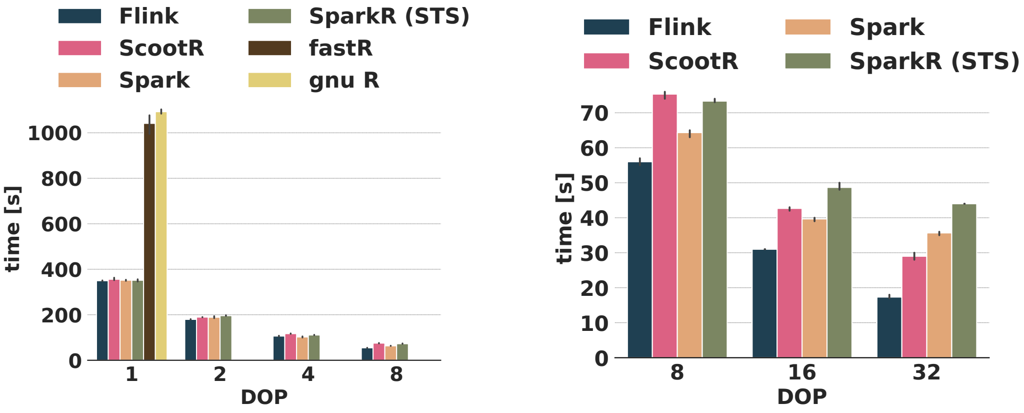


FIGURE 3.14: Cluster micro-benchmark for calculating the N-grams in the *body* column of the Reddit comments dataset. The data is scaled according to the number of used nodes.

in the iterator have to be materialized in the `mapPartitions` function and are provided as dataframe to the R UDF. This introduces a materialization barrier in the streaming execution and causes additional performance overhead. ScootR (MP) also uses the `mapPartitions` operator of Flink, but has access to the iterator via the language interoperability features described in Section 3.2.2 Ⓑ § *Data Access and Exchange*. Thus, ScootR does not have to materialize and can directly access the tuples in a streaming fashion via the iterator in the R UDF.

UDF Operator with 1:N Output. The next micro-benchmark executes a more complex UDF, where we generate all *2-grams* within the *body* column of the Reddit comments dataset. Compared to the previous benchmarks, the UDF is compute-heavy and second, the function is called within a `flatmap` operator. As the body has N 2-grams per comment, the function may emit $0, 1, \ldots, N$ elements per input tuple. The ScootR function used in the experiment is detailed in Listing 4 on Page 104. As described in Section 3.2.2 Ⓑ § *Data Access and Exchange*, ScootR has direct access to the Flink `Collector` class, which collects the output directly in the R UDF. Figure 3.14 depicts the result of the benchmark. The data size is increased alongside with



(a) Single node **(b)** Cluster

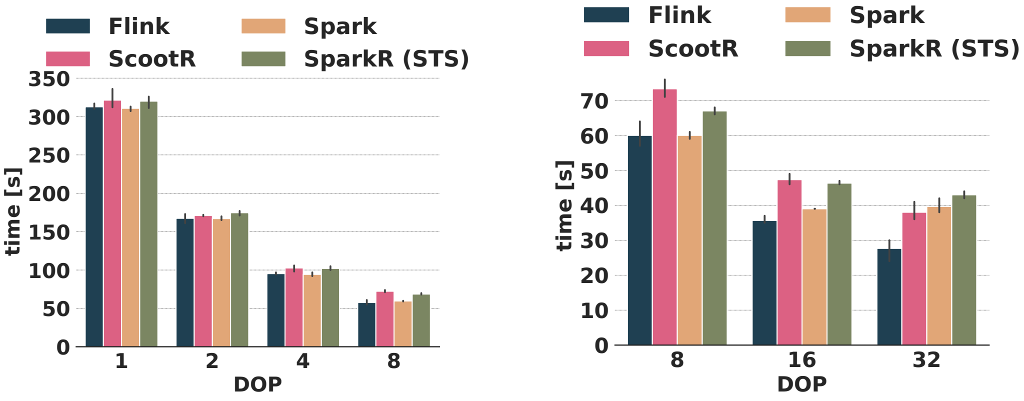
Figure 3.15: Benchmark for the ETL pipeline shown in Listing 1 on Page 103.

the number of nodes and we use 1, 2, and 4 months of data. We observe that ScootR is only about a factor of 1.15× slower than Flink. As we can access the `Collector` class and create the Flink tuples directly inside the R function, we avoid the materialization and type conversion of the returned result. We report the execution times without access to the `Collector` as ScootR (List) to show the benefit of direct access to Flink's data structures. As discussed in Section 3.2.2 Ⓑ § *Data Access and Exchange*, the necessary additional pass over the List and the type conversion results in 1.2× slower execution compared to ScootR with direct access. SparkR with IPC failed to execute within the set experiment timeout of 4 hours. The UDF cannot be expressed in SparkR with STS.

3.3.3 Results for Operator Pipelines

In this section, we provide benchmarks for operator pipelines. The first benchmark shows an ETL pipeline composed of several operators. It validates the results from the previous micro-benchmarks and shows competitive performance for ScootR and SparkR compared to their respective system's native APIs. Both outperform GNU-R and fastR by up to 2.5× for single-threaded and up to 20× for distributed execution. Again, while SparkR uses STS, ScootR achieves this while executing the UDFs. The second benchmark shows a classical MapReduce pipeline. The runtime of ScootR and SparkR is on par with the native API. The third benchmark shows a mixed pipeline combining preprocessing and model training. It shows the benefits of the seamless integration of ScootR, as we collect the distributed data for further local processing in the same R script (cf. Listing 3 on Page 104). Thereby, we can achieve up to 12× performance improvement compared to executing the complete pipeline in fastR as the majority of the time is spent for preprocessing. In the following paragraphs, we describe each benchmark in detail.

ETL Pipeline. In this experiment, we execute the pipeline described by Oscar D. Lara et al. [YZP14]. The ScootR code for the pipeline is depicted in Listing 1 on Page 103. The goal of this pipeline is categorizing the delay of flights by two carriers in the airline dataset.



(a) Single node **(b)** Cluster

Figure 3.16: Benchmark for the MapReduce pipeline shown in the Listing 2 on Page 103.

Figure 3.15a depicts the results for the execution on a single node and Figure 3.15b the four nodes cluster. SparkR (STS) is around 1.2× slower than Spark and ScootR is up to 1.4× slower than Flink in the worst case. Both outperform GNU-R and fastR by 2.5× for single-threaded and up to 20× for distributed execution. GNU-R is only 1.05× slower than fastR. This is mostly due to the high selectivity of the `filter` operator at the beginning of the pipeline, which significantly reduces the amount of data. Thus, the data that has to be processed by the two successive UDFs is reduced significantly.

MapReduce Pipeline. So far, the benchmarks did not involve aggregations. Therefore, we designed a MapReduce-style aggregation pipeline to determine the largest arrival delay per destination for flights that started from New York. The ScootR code for the pipeline is depicted in Listing 2 on Page 103.
Figure 3.16a depict the results for the benchmark on a single node and Figure 3.16b the four nodes cluster. ScootR is up to 1.3× slower than Flink and SparkR (STS) up to 1.15× than Spark. While both translate the aggregation function to a native API call, ScootR directly executes the R predicate for the `filter` function. Even though the data size is reduced significantly by the `filter` operation, the aggregation, due to the necessary shuffle step, together with the initial reading of the data is still responsible for the majority of the execution time.

Mixed Pipeline. In this experiment, we evaluate a mixed pipeline. We use Flink to perform the initial data preprocessing and gather its result on the driver node. The result is then used for further analysis as in-memory dataframe. Specifically, we train a generalized linear model [NW72] with the `glm` function provided in R and show the model description with the `summarize` function. The ScootR code for the pipeline is depicted in Listing 3 on Page 104.
Figure 3.17 depicts the results for the described pipeline. We can observe that most of the execution time is spent in the ETL phase, which reduces the initial Airline dataset from 9.5GB to approximately 90MB. While all of the systems spend the majority of the time in the ETL phase, we can decrease its duration significantly by using ScootR, even in the single-threaded

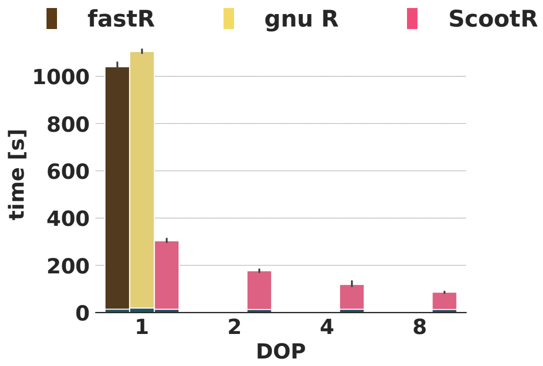


Figure 3.17: Benchmark for the mixed pipeline shown in Listing 3 on Page 104. The fraction of time spent for the `glm` function is indicated in dark blue.

execution case. Compared to GNU-R, ScootR is about 3.6× faster for single-threaded execution, and 12.3× faster when using 8 cores. The performance drawbacks of fastR and GNU-R result from the initial dataset reading and materialization costs.

3.3.4 Discussion

The main goal of our evaluation was to highlight our two main contributions: (*i*) The integration of an R language API based on a common runtime to avoid data exchange, while supporting a rich set of language features. (*ii*) The necessity of our applied optimizations to share data structures between the guest- and the host language to provide fast data access and avoid type conversion. To this end, we conducted benchmarks comparing ScootR, SparkR, fastR, and GNU-R for both single operators and operator pipelines for single node and cluster configurations. The non-UDF micro-benchmark functions clearly show that ScootR and SparkR provide reliable mapping of R functions to their respective native dataflow engine API calls, with below 1.2× overhead. For functions calling R UDFs, ScootR can compete with SparkR's STS approach, even though ScootR executes the R function in the fastR language runtime. In contrast, when SparkR has to fall back to IPC, its performance degrades by an order of magnitude compared to ScootR. The benchmarks for 1:N operators show that direct access to data structures is necessary to avoid data materialization and therefore achieve comparable performance to the native API of the dataflow engine. The benchmarks for operator pipelines, validate the assumptions behind the micro-benchmark experiments, and show very small performance overheads of up to 1.2× for SparkR (STS) and 1.4× for ScootR. Both SparkR and ScootR outperform GNU-R and fastR.

3.4 Related Work

In this section, we discuss related work on DSL language compilers, parallelization based on existing dataflow systems, and parallelization packages for the R programming language itself.

Compiler-based Approaches. Weld [PTS+17] offers a functional IR based on nested parallel loops and *builders* that specify what should be computed. Libraries and functions represent their operations using this IR to avoid materializing/copying of intermediate results, which is normally required when data is passed from one library to another. Weld applies optimizations such as loop tiling and vectorization and generates code for diverse processors including CPUs and GPUs. Tupleware [CGD+15] is a distributed analytics system that focuses on the efficient execution of UDF-centric workflows. It provides an IR based on the LLVM compiler framework [LA04], which can be targeted by any language that emits LLVM code. Tupleware applies high-level optimizations, such as predicate pushdown or join reordering, as well as low-level optimizations, such as reordering of the program structure and vectorization. Pydron [MAAC14] parallelizes sequential Python code to execute on multi-core, cloud, and cluster infrastructures. It is based on two Python decorators, one to mark functions to parallelize and another one to mark side-effect free functions. Functions annotated for parallelization are translated to an intermediate representation. Pydron applies several optimizations based on this IR, including control flow and scheduling decisions for its parallelization.

All of the systems mentioned above provide a familiar interface to the programmer, while they achieve efficient execution by carefully applied optimizations or parallelized execution. ScootR shares this goal, but incorporates R into an existing dataflow system *without* changing it. It achieves this by relying on the GraalVM, a JVM-compatible language runtime that enables multi-language execution. The approach is not restricted by an IR specially designed for the systems optimization goals. Thus, ScootR profits directly from the ongoing efforts to advance the performance of Graal, and can be easily extended with support for new languages and diverse processors, e.g., GPUs [FSSD17].

Parallelism based on Dataflow engines. Hadoop's *Streaming* utility is used as a common basis for IPC in several frameworks. It allows specifying executables and scripts that are called in the map and reduce functions. Here, scripts receive data via *stdin* while results are emitted via *stdout*. RHadoop is a collection of tools to work with the Hadoop ecosystem within R. Likewise, R Revolution, now called Microsoft R Open and Server (commercial version), provides the option to run R on top of Hadoop. All the presented systems inherit the drawbacks of IPC, namely communication overhead, serialization, and data-processing pipeline disruption, as discussed in Section 3.1.2.

RHIPE [Guh10] is also based on Hadoop and uses IPC while exchanging data via Google's ProtocolBuffers, a language- and platform-neutral mechanism for serializing structured data.[11] RHive allows for easy use of HSql, the query language of Hive [TSJ+09], in R. Also, UDFs and UDAF can be specified in R, which are executed via IPC with an external R process. RHIPE has

---

[11] https://developers.google.com/protocol-buffers/

a more efficient data exchange format compared to Hadoop Streaming, but it still inherits the drawbacks of IPC, as the executables run in separated processes.
Ricardo [DSB+10] was developed by IBM and executes Jaql [BEG+11] queries on top of Hadoop. Beside Jaql functions that do not involve user-defined code, users can specify R UDFs, which are executed in an external R process. Thus, it provides a hybrid approach as discussed in Figure 3.1.2. Ricardo inherits the drawbacks from IPC when executing UDFs, but it provides so-called *trading* that allows for mixed R and Hadoop execution. Preprocessing can be executed in Hadoop before the results are fetched in R and can be used as input to the manifold libraries in R. ScootR is influenced by the trading concept, but does not have to fall back to IPC in case of UDFs.
Big R [YZP14] is based on IBM BigInsights and uses a restricted, overloaded set of R operations and transformations specified in Jaql that can be executed on top of Hadoop. The results are returned as a dataframe, which is used for further processing in R. In contrast to Big R, ScootR is not restricted to a limited set of operators and executes arbitrary R functions.
SparkR [VYL+16] provides a dataframe-centric programming abstraction in R. As described in Section 3.1.2, SparkR avoids IPC by applying source-to-source translation for a subset of operations and library calls. In case the source-to-source compiler cannot translate the R program, SparkR falls back to use an external R process by IPC. This fall back causes large performance penalties. ScootR builds upon the ideas of SparkR for non-UDF operators, while improving execution time for arbitrary UDFs. Spark also provides a programming abstraction for Python, called PySpark. While the underlying concepts are the same as in SparkR, there is an ongoing effort to integrate Apache Arrow [Apa18]. Apache Arrow's goal is to provide a common in-memory data representation that provides efficient access and APIs in Python, C, and Java. Therefore, it improves data exchange between Spark and Python, while also providing more efficient access for the popular Python *pandas* dataframes. While Arrow looks promising, data needs to be serialized to and de-serialized from the binary format of Arrow.
SystemML [BDE+16] is a system for the efficient execution of linear algebra programs on Apache Spark written in a DSL based on R's matrix abstraction. While its focus is clearly on linear algebra, it provides basic facilities to transform input data with a restricted set of operations and predefined functions. SciDB [Bro10] is an array database that focuses on the efficient execution of array manipulation and linear algebra. SciDB provides an R abstraction in addition to its native API. Both systems do not focus on UDF and are therefore orthogonal to ScootR.

R Parallelization Packages. *Explicit* parallelization packages, such as Snow [TRLS16] and Snowfall [Kna15], provide parallel versions of the `apply*` methods. In addition, there are packages based on parallelized versions of the `foreach` construct [MW17, Wes17] for different back-ends, such as Sockets, Message Passing Interface (MPI), and Parallel Virtual Machine (PVM). These packages focus on parallelizing computation heavy, splittable tasks, but not on large amounts of data. They offer no facilities to read distributed files but reflect the scatter/gather model from MPI. ScootR focuses on parallelizing computations on large amounts of data.

3.5 Conclusion

In this chapter, we presented ScootR, a novel approach to execute R user-defined functions in dataflow systems with minimal overhead. Existing state-of-the-art systems, such as SparkR, use source-to-source translation to the systems' native API to achieve competitive performance for a restricted subset of UDFs. When running arbitrary UDFs, their performance may degrade by a factor of up to 170×, as they have to fall back to inter-process communication. This overhead is due to the necessary data serialization and data transfer imposed by IPC.

ScootR avoids such overheads by tightly integrating the dataflow engine with the R language runtime, using the Truffle framework and the Graal compiler. By making Flink abstract data types accessible to the R user-defined functions, ScootR avoids type conversion as well as intermediate result duplication and copies. Our experimental study shows that ScootR achieves comparable performance to systems based on STS, even though ScootR executes the UDF in an R language runtime. When SparkR has to fall back to inter-process communication, ScootR has up to an order of magnitude higher performance.

Future Work. The techniques and approaches presented in this chapter are general and can be applied to other dataflow systems as well. With possibly few exceptions, most of the existing systems provide a relational-style API based on typed, fixed-length tuples. For instance, one could provide a similar abstraction implemented on top of the Spark `Dataset` and/or `Dataframe` abstractions, following the approach outlined in this chapter. Another interesting extension would be the integration of other Truffle-based (dynamic) languages such as JavaScript or Python. To this end, a small language agnostic and data-processing centric Truffle API could be defined and used as a common abstraction by different language runtimes.

One goal of ScootR was to realize a guest language integration without changes to the existing code base of the dataflow engine. Deeper integration of Truffle and the runtime would yield further potential for optimizations: ScootR does not exploit the self-optimizing capabilities of Truffle, as Flink (and all other dataflow engines implemented in typed languages) requires typed UDFs. A dataflow engine that is implemented as Truffle AST nodes (e.g., its operators, internal data structures, etc.) could exploit the self-specializing capabilities of Truffle to optimize itself to the data at runtime.

This provides several opportunities: First, the internal data structures can use specialized variants for certain types, e.g., to provide specialized hash tables for joins and sorting. Second, the profiling facilities of Truffle can be used to gather statistics during runtime. Third, type sizes can be changed gradually during runtime, e.g., a numeric can be represented as 4-byte integer and converted to an 8-byte long once a number-overflow is detected. Such behavior would be especially interesting in streaming settings with longstanding queries.

4

OPTIMIZING ML TRAINING PIPELINES

Modern data analysis pipelines often include preprocessing steps, such as data cleaning and feature transformation, as well as feature engineering and selection [SHG+15, STH+15, KMNP15, BFG+17, BBC+17]. 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, 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* [GW14] in the host language, i.e., they are executed *as-is*, without any inter-library optimizations and support for large data [PTN+18]. General-purpose dataflow systems [ZCF+10, ABE+14] provide second-order functions (e.g., *map* and *reduce*) to transform collections via UDFs. They defer program execution by providing a type-based 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 [HPS+12]. In contrast, dedicated systems for ML, such as SystemML [BDE+16] and Tensorflow [ABC+16] 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 [SBJ+18], 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.

Research Contributions. To address these issues, we propose Lara, a DSL that combines collection processing and machine learning.[1] Lara is based on *Emma* [AKK+15, ASK+16], a quotation-based DSL [NLSW16] 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 AST of the whole program, allowing us to inspect and rewrite UDFs and native control flow. Emma's IR is based on monad comprehensions [GS99] 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 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 [GM93]: 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:

- We propose Lara, a quotation-based DSL for end-to-end model training pipelines with dedicated types for collections and matrices (Section 4.1).
- 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 4.1 and 4.2).
- 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 4.2.4).
- 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 4.3).

4.1 Language and IR

In this section, we provide an overview of important design decisions based on an introductory example. Next, we describe the IR of Lara and introduce two views on top of the IR, which are used to perform the diverse optimizations showcased in Section 4.2.

---

[1] This chapter is based on [KAKM16] and [KKS+19].

4.1.1 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 [AKK+15, ASK+16], 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.numRows, M.numCols)((i,j) => means(j))
19    // Compute the covariance matrix
20    val C = 1 / (U.numRows - 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 two datasets `Users` and `Reviews`. ML pipelines are expressed as high-level linear algebra operators. 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.

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 (cf. Line 14) or provided as library functions (cf. Line 1 & 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 [Bur13] (cf. Line 4).

❸ – Type conversion methods (cf. 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 set are disjoint. The iterative calculation of the mean in Line 16 can be optimized. Instead of executing the mean function for each column separately, it is executed for all columns at once.

❻ – High-level linear algebra operators in the API (cf. Line 20) and an IR that captures the domain-specific semantics of operators enable the selection of specialized operator implementations, e.g., BLAS [LHKK79] instructions for linear algebra.

4.1.2 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, a normalized representation of the Scala AST in let-normal form (LNF) [App98]. Then, we present two higher-level *views* on top of the LIR. The *monadic view* represents monad comprehensions [GS99] over the `DataBag` and `Matrix` types. The *combinator view* represents high-level operators (e.g., matrix multiplication) as single entities or *combinators* [Gru99] in an operator tree.

Low-Level Intermediate Representation. Emma uses the meta-programming features of Scala [Bur13] to access the AST of a quoted program. In an initial step, Emma transforms the original AST into LNF, a functional representation of static single assignment form (SSA) [App98], 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* [AK01] 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 the basis for the *views*, which we introduce in the next paragraphs. The views combine expressions of the LIR that represent an operation in their higher abstraction, 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 [GS99] 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 (cf. Listing 4.1.1, Line 10). Matrices and vectors can also be represented as a *Set* monad $\{(\mathtt{i}, a)\}$ of index-value tuples, where $\mathtt{i}$ is a singular *index* in case of a vector and a tuple (*row-index*,*column-index*) in case of a matrix [FM00]. 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. In Section 4.2.1 and 4.2.2, we showcase optimizations on monads and their interplay with control flow analysis. 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:[2]

```
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 [TER18]. Comprehension combinators [Gru99] 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 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 [LHKK79]. 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.

---

[2] Fegaras et al. provide a list of linear algebra operators and their corresponding monad comprehensions [FM00].

The combinator view for the computation of the gram matrix $\mathbf{X}^\top\mathbf{X}$ is illustrated in Figure 4.1a. Figure 4.1b 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 4.1c 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 [GM93] establishes the data layout desired by the DGEMM instruction for its second operand. We provide a detailed discussion in Section 4.2.3 and 4.2.4.

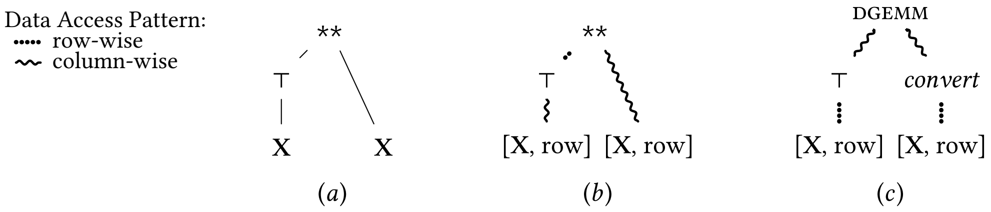


Figure 4.1: Combinator view for gram matrix $\mathbf{X}^\top\mathbf{X}$.

4.2 Optimizing End-to-End Pipelines

Listing 4.1 depicts our running example of an end-to-end ML pipeline for model training, which leverages historical data about clicks on advertisements to predict the number of future clicks on other advertisements using a regression model. First, we preprocess input data to obtain a dataset of labels and numerical features (cf. Lines 3 to 9). Next, we learn and validate regression models with different hyperparameters on the resulting dataset (cf. Line 11 to 27).

The pipeline showcases common user-defined feature transformations. We omit the implementation of the UDFs for the sake of space. The categorical features in columns 11 to 15 are *dummy-encoded* [HH10] as sparse vectors in Line 4. The numerical features in columns 1 to 10 are *normalized* [Gru15] 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 [Koh95] on the normalized feature matrix X. Lara provides cross-validation as utility method, similar to popular ML libraries such as scikit-learn.[3] 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 X and target vector y. This example trains a ridge regression model in Line 17 – 20, and calculates its test error in Line 22 – 23.

---

[3] http://scikit-learn.org/stable/modules/cross_validation.html

```
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 // diag(lambda)
18     val XtX = X_train.t ** X_train + reg
19     val Xty = X_train.t ** y_train
20     val w   = XtX \ Xty                            // solve(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 4.1: An end-to-end training pipeline in Lara.

Overview of Lara's Optimizations. In the next sections, we showcase how our IR enables different optimizations and present the LIR's interplay with the monadic and combinator view. To this end, Section 4.2.1 describes how the monadic view enables the pushdown of `Matrix` UDFs to the `DataBag`. Section 4.2.2 discusses how the monadic representation enables operator fusion across loop boundaries. Subsequently, we describe how to choose data layouts based on interesting properties, by using the combinator view and its domain-specific semantics in Section 4.2.3. Finally, Section 4.2.4 highlights the integration of cross-validation into the IR and introduces optimizations for the applied learning algorithm based on its semantics in the combinator view.

4.2.1 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 4.1, 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 [LRU14] 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` – 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 (NNZ). However, the numerical features are separate entries in the array elements (which provides constant time access) of the `DataBag` before they are combined with the categorical features in Line 5 & 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 desired pushdown of UDFs.

$$\begin{tikzcd}
\mathsf{T}A \arrow[r, "\mathsf{map}f"] \arrow[d, "\eta_A"] & \mathsf{T}B \arrow[d, "\eta_B"] \\
\mathsf{U}A \arrow[r, "\mathsf{map}f"] & \mathsf{U}B
\end{tikzcd}$$

(a) $\mathsf{T}\,A \to \mathsf{U}\,B$

$$\begin{tikzcd}
\mathsf{Bag}\,A \arrow[r, "\mathsf{idx}"] & \mathsf{Bag}\,\mathsf{i},B \arrow[d, "\eta_B"] \\
 & \mathsf{Set}\,\mathsf{i},B
\end{tikzcd}$$

(b) $\mathsf{Bag}\,A \to \mathsf{Set}\,\mathsf{i},B$

Figure 4.2: Natural Conversion (*a*) and modified conversion method in Lara (*b*).

Background: Conversion Methods. Generic type conversions correspond to a categorical concept called *natural transformations* [ML13]. Natural transformations are polymorphic functions $\eta_A : \mathsf{T}\,A \to \mathsf{U}\,A$ which change the container type from T to U (e.g., from from Bag to Set) irrespective of the element type A. Their characteristic property states that application of η_A commutes with application of map f for all f, as depicted in Figure 4.2a. Unfortunately, this is not the case when converting a DataBag to a Matrix. In this case, the container type, from `Bag` to `Set`, and the type changes, from `A` to (i, A), introducing the index i of the matrix/vector. To overcome this problem, the conversion methods (cf. Listing 4.1, Line 8) accept only `DataBag`s with instances of Lara's vector type as elements or expect an index function of form $idx : A \to (\mathsf{i}, B)$, as depicted in Figure 4.2b. Lara's vector type provides instances for `Product`, `Array`, and `Vector` in the moment. Therefore, Lara can track data provenance, i.e., how the access to a matrix cell $((i, j), value)$ commutes with access to a $(i, vector)$ element in a `DataBag`, which enables operator pushdown. Conversion methods are represented as monad comprehensions in the monadic view, as depicted in Figure 4.3 (top arrows). In a first conversion step, the row index

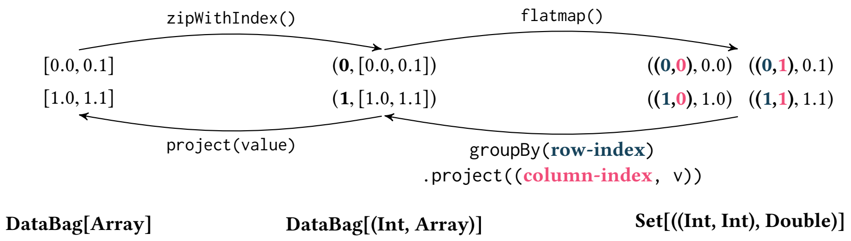


Figure 4.3: Conversion between `DataBag` and Matrix.

i is created on the unordered `DataBag` with a `zipWithIndex` method. Second, the indexed elements are split up in a `flatmap`, which emits $((i, j), value)$ tuples that represent the cells of the matrix. With this explicit representation of the conversion methods and types as monad comprehensions, we can define a mapping for the second-order functions that apply UDFs from the `Matrix` type to the `DataBag`. A natural 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. For instance, a function that is applied to a particular column can not be pushed before the feature hashing [WDL$^+$09] function that created the numerical representation of this column.

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 `groupBy` and the conversion to a vector (`project`) revert the conversion method depicted in Figure 4.3 (bottom). Thus, if we pushdown the `forRows` UDF through the conversion method, only the application of the UDF is left, which can be simply executed in a `map` on the `DataBag`. Table 4.1 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)`

Table 4.1: 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
	`.map` : (rowIdx, value) : values as vector
	+ `.map(m)` or `.map(a)` or `.withFilter(f)`
		`forCols(m:Vector => Vector)`
		`forCols(a:Vector => Double)`
		`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)`

on a `DataBag` requires the following comprehensions: `flatmap.groupBy.map.withFilter(a)` as listed in Table 4.1. The `flatmap` splits the row-wise partitioned `DataBag` into separate cells. Then, the cells that share the same column index are grouped and combined to a vector representation, before the filter UDF a can be applied.

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 4.1, 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 } yield {
5      (elem.index.colIndex, elem.value)
6    }
7  }
```

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 4.1.

4.2.2 Operator Fusion

As discussed in Section 4.2.1, 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 necessary 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 [Wad88, CLS07]. It fuses consecutive applications of UDFs in `map` second-order functions into a single, composed function call:

`T.map(`f_1`).map(...).map(`f_N`) = 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* [BdM96] law states that pairs of folds that is applied on the same data type can be fused into a single fold, resulting in following code:

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

The *cata-fusion* [BdM96] law enables us to fuse `map` and `filter` operations into a consecutive `fold` application. Lara leverages the fusion capabilities of Emma [AKM19], 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 (cf. Listing 4.1, Line 4), which is implemented as follows. For simplicity, we hide the implementation details and only show the input parameters.

```
def dummyEncode(bag: DataBag[Array[Any]], columns: Seq[Int]) = {
  var encoded = bag
  for (columnIndex <- columns) {
    // Fit: build a dictionary of column values
    val dictionary = encoded.fold(zero)(vec => init(columnIndex), plus)
    // Transform: create encoding in sparse vector
    encoded = encoded.map(vec => createVectors(vec, columnIndex, dictionary))
  }
  encoded
}
```

Line 5 and 7 depict the dummy encoding for a single column, which is applied iteratively for all columns defined by the `columns`. 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` appears to be a fusion barrier, as the built dictionary is required to perform a lookup in the successive `map` operator. A closer look reveals that the code accesses only a distinct feature column within each iteration. Thus, when we would *unroll* the loop, the consecutive `fold` applications could be fused (cf. Background: Fold-Fusion), *if* disjoint columns are accessed. Analogously, we could combine all `map` applications.

```
val columnIterator = columns.toIterator
@whileLoop def WHILE(encoded: DataBag[Array[Any]]) = {
  val hasNext = columnIterator.hasNext
  if (hasNext) LOOP_BODY() else LOOP_SUFFIX()

  @loopBody def LOOP_BODY() = {
    val columnIndex     = columnIterator.next()
    val dictionary      = encoded.fold(zero)(init, plus)
    val updatedEncoded  = encoded.map(createVectors)
    WHILE(updatedEncoded)
  }
  @suffix def LOOP_SUFFIX() = { encoded }
}
WHILE(data)
```

● *def*
○ *use*

Listing 4.2: Simplified LIR of the loop in the dummyEncode method.

Lara analyses the loop based on the *direct style* control flow representation of the LIR. It validates that no *loop-carried dependencies* [AK01] exist and that the read and write accesses to the array elements inside the UDFs are conducted with the `columnIndex` loop variable. The code snippet in Listing 4.2 depicts the loop of the dummyEncode method in the LIR. We hide the implementations of the UDFs in the `fold` and `map` in order to increase readability. The snippet highlights the *direct style* control flow representation, i.e., loops are replaced by recursive calls to the `WHILE`, `LOOP_BODY`, and `LOOP_SUFFIX` methods. All native loop primitives supported in Lara are translated to this canonical form and annotated (e.g., `@whileLoop`) to avoid ambiguity during the analysis. The highlighted variables show the 1:N def-use chains [AK01] of the loop variable `columnIndex` from the sequence iterator to the access in the UDF of the `fold` and the `map` functions. The use-def chain of the `dictionary` created by the fold shows the dependency

```
val createVectors = (array: Array[Any]) => {
  val columnValue = array.apply( columnIndex )
  val index = dictionary .get(columnValue)
  val dimensions = dictionary .size
  // Create sparse, dummy encoded vector
  val vector  =  SparseVector.apply(index, 1.0, dimensions)
  val _ =  array.update( columnIndex , vector)
  array
}
```

Listing 4.3: Simplified LIR of the `createVectors` UDF of the `dummyEncode` method.

to the `map` function. Direct edges between definition and usage of the `columnIndex` ensure that a new value is taken from the iterator. To validate that the column index is not modified, the UDFs have to be inspected.

The code snippet in Listing 4.3 depicts the UDF `createVectors` of the `map` in the `dummyEncode` method, which is called for each `Array[Any]` element of the `DataBag`. Direct access to the `columnIndex` ensures that the read and write accesses to the array elements of the `DataBag` use the exact values from the iterator. Read-only access to the dictionary validates that its values are not modified. 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 (as in our running example which uses the constant sequence `11 to 15`) or the function semantics guarantee disjoint access (e.g., all Bulk-Operations in Table 4.1). Lara analyzes the UDFs executed in the `fold` analogously.

Fusion. After the dependencies have been evaluated successfully, Lara *unrolls* loops to enable operator fusion. First, we leverage the banana-split [BdM96] rule to combine 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 (cf. 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 [Neu11], i.e., (aggregated) data, which is required by a successive operator and thus, has to be materialized. While Lara can not overcome this inherent limitation, it can fuse multiple `fold`s 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* optimizer for MR [LHB12]. Horizontal packing combines map (or reduce) functions from multiple jobs that use the same data set, which corresponds to the described function composition for `map` and banana-split for `fold` functions. 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`

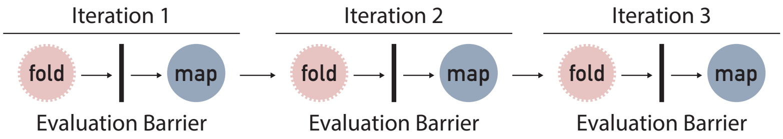


Figure 4.4: Evaluation barriers due to lazy evaluation in type-based DSLs.

and `map` higher-order functions in the example. Control flow and UDFs are not visible, which prevents the required dependency analysis. Figure 4.4 depicts the evaluation barriers introduced by the `fold` for the `dummyEncode` method. The `fold` triggers the evaluation and execution of the the operator graph.
SystemML avoids dependency analysis and provides a *transform* function, which can apply multiple pre-defined transformations on a dataset. Thus, it can automatically apply transformations over multiple columns as their semantics are known, but does not support user-defined transformation functions.

4.2.3 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 [TK18]. We leverage the combinator view to choose appropriate physical implementations of operators based on the layout of the data. Figure 4.5 depicts three plan variants for the ridge regression algorithm (Listing 4.1, 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 [GM93], Lara applies the following optimizations: (*i*) expressions (i.e., sub-graphs) 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 [GM93] 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.[4] 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 4.5b). 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

---

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

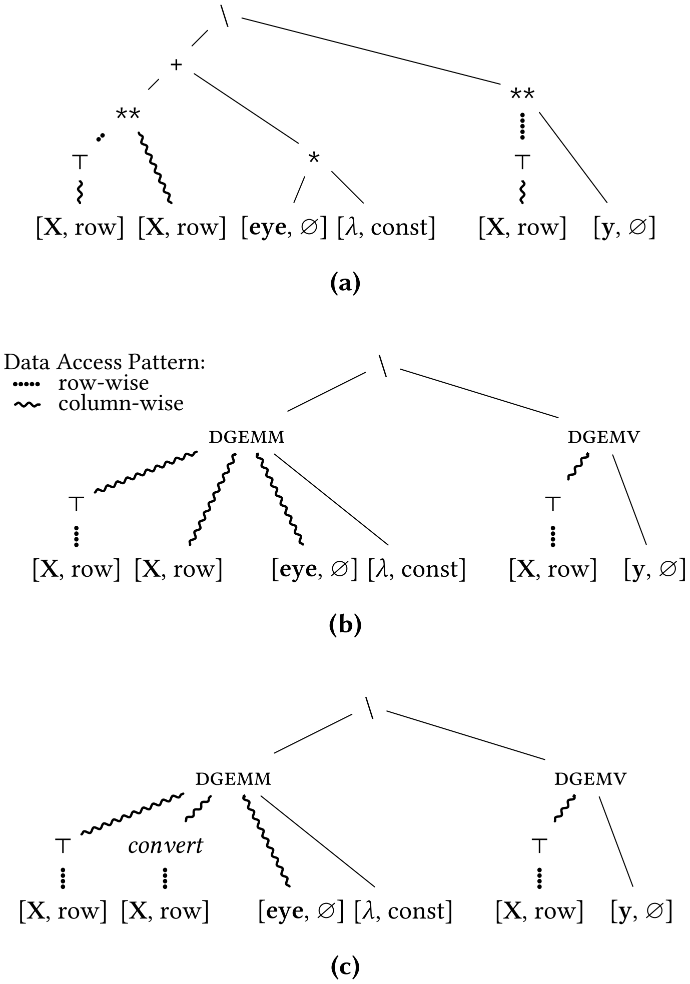


Figure 4.5: (*a*) the combinator view for ridge regression (cf. Listing 4.1, Line 17 – 20) with default operators, (*b*) replaced sub-trees with equivalent BLAS instructions, and (*c*) an enforcer *convert* for the interesting properties of plan variant (*b*).

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 [GM93]. For instance, the default Scala implementation of a matrix-matrix multiplication (`**`) in Figure 4.5a 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 4.5b 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 format depending on the properties of the parent node. Consider the plan variant in Figure 4.5b 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 4.5c 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. A cost model would require access to statistics, e.g., about the shape of input data and cardinality estimates, at compile-time to apply optimizations accordingly. Thus, we would rely on some kind of meta-data, which stores such statistics over multiple program runs. Another interesting strategy would be to implicitly ingest profiling functions (e.g., as `map` functions that are fused with other UDFs) that gather statistics during the execution of the pipeline, e.g. statistics of intermediate results and number of non-zeros. These runtime statistics can then be used to re-optimize at run-time after natural execution barriers, e.g., after conversion to a matrix or after data aggregations.

Compile-Time vs. Run-Time Code Analysis. Run-time statistics could also be used to overcome the limitations of static code analysis. At the moment, Lara has to decide upfront which matrix representation to choose – dense or sparse – as the optimization is done during compile-time. While the best representation can often be determined for the initial data sources (e.g.,

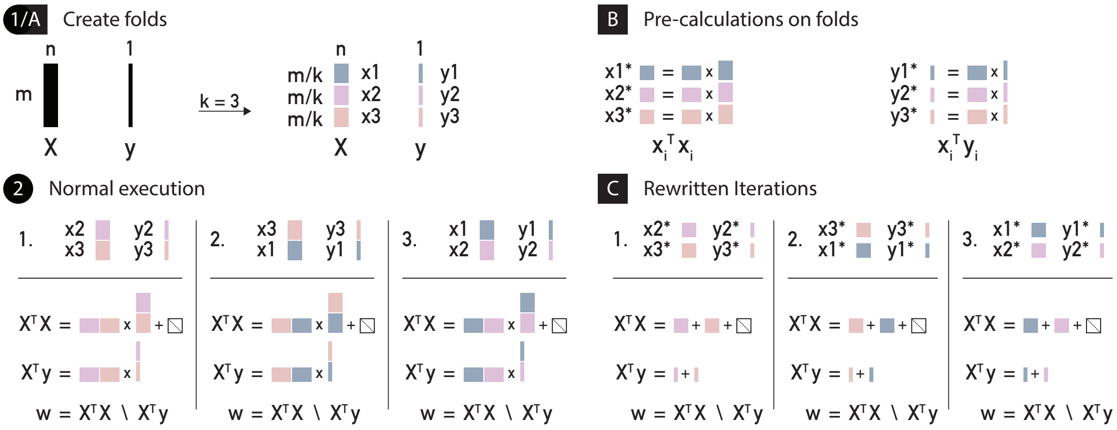


FIGURE 4.6: Cross-validation for ridge regression. Steps 1 and 2 depict the naïve execution with redundant computations. Steps A, B and C depict the optimized execution, which computes partial results outside the cross-validation loop.

due to user-defined formats in feature transformation methods or input files), the best format for intermediate results has to be chosen at runtime. Another drawback of static code analysis is that it prevents the evaluation of dynamic control flow predicates and thus, selection of the best access pattern in such cases. The access patterns in the following listing are dependent on dynamic control flow:

```
var X = Matrix(someDataSet)
val Y = Matrix.rand(...)
if (sum(X)) // sum of all elements
  X = X.t
println(X ** Y)
```

Lara cannot evaluate the predicate `sum(X)` during compile-time. Thus, Lara ignores variables during the data layout optimizations that are modified in dynamic `if-else` branches. Optimization during run-time (e.g., after evaluation of the predicate) could overcome this problem.

4.2.4 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 [Koh95], 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. K-fold cross-validation provides robust estimates of the generalization error of a model for a given hyperparameter and ensures that every data point is in the test set once. We implement cross-validation as utility function (Listing 4.1, Line 14). In our running example, we use cross-validation to select a regularization constant `lambda` for the ridge

Table 4.2: Linear algebra operations and their respective cost in the cross-validation method. $\oplus$ stands for element-wise operations.

Operation			**Time Complexity**			
Baseline	*Pre-Computation*	*Per-Iteration*	*Baseline*	*Pre-Computation*	*Per-Iteration*	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	$(k-1)mn$	$k:(k-1)$

regression model.[5] Its execution is illustrated on the left side of Figure 4.6. Step ❶ is independent of the validation algorithm and partitions the specified feature matrix $\mathbf{X}$ and target vector $\mathbf{y}$ into k splits. In Step ❷, 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 4.6, 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 $\mathbf{X}_1 \ldots \mathbf{X}_k$ with size $\frac{m}{k} \times n$. Block-partitioned matrices can be multiplied when they have *conformable partitions* [Eve80], 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 $\mathbf{X}_1 \ldots \mathbf{X}_k$) 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 4.6, Step 🅱). In each particular iteration, the pre-computed results of the test-set splits have to be added to calculate the final result (Figure 4.6, Step 🅲). The time complexity for $\mathbf{X}^\top\mathbf{X}$ for a matrix with m rows and n columns is $O(n * m * n)$. 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 $O(k[n * (m - \frac{m}{k}) * n]) \equiv O((k-1)mn^2)$. In the optimized execution, the multiplication is performed once for all *individual splits* in Step 🅱. Each split has $\frac{m}{k}$ rows and n columns,

---

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

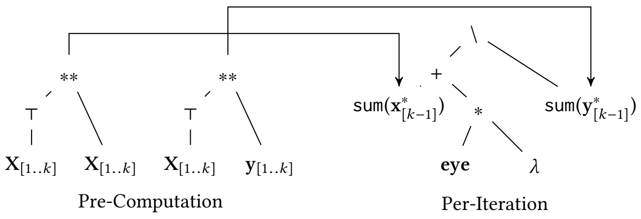


Figure 4.7: Optimized combinator tree for ridge regression under cross-validation.

resulting in the complexity of $O(k[n\frac{m}{k}n]) \equiv O(mn^2)$ for all k splits. In each iteration k we only have to add $(k-1)$ pre-calculated partitions of size $n \times n$, resulting in the overall complexity of $O(k[(k-1)nn])$ 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 4.2 (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. *Pre-Computation* is the one-time cost for the part of the operator that can be pre-calculated statically on each split. *Per-Iteration* 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 $(k-1) \times$ 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 4.7 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. The sub-trees extracted from the original tree are replaced with the Per-Iteration operation of the sub-tree root node operation. 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) [AK01]. 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-Computation is 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.

4.3 Evaluation

We implemented Lara and the outlined optimizations in Scala, based on Emma v0.2.3.[6] 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 an 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[7] dataset contains click feedback of display ads. (ii) The *Reddit*[8] dataset contains comments of the news aggregator website *reddit.com*.

Overview. We evaluate the optimizations for the preprocessing pipelines based on the introductory example (cf. Section 4.1.1) and our running example (cf. Listing 4.1) and validate the importance of operator pushdown in Section 4.3.1 and 4.3.2. We measure the impact of data layout optimizations on ML algorithms for dense and sparse data in Section 4.3.3. We benchmark the optimizations for cross-validation and hyperparameter tuning in Section 4.3.4 and 4.3.5.

4.3.1 Preprocessing

In this experiment, we evaluate the presented optimizations on the preprocessing pipeline of our running example depicted in Listing 4.1, Line 3 – 9. We conducted the benchmark on differently sized samples of the Criteo dataset. We evaluate the impact of operator fusion and pushdown on each of the three preprocessing steps in Figure 4.8. Figure 4.8a 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 4.8b. *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 (cf. Listing 4.1, Line 4). *Normalize* transforms the numerical features to have zero mean and unit variance (cf. Listing 4.1, Line 9). *Concat* combines the numerical and the dummy encoded features in a single sparse vector (cf. Listing 4.1, Line 5 – 6). In Figure 4.9, we compare Lara to scikit-learn, Spark, and Tensorflow Transform on different data sizes.

---

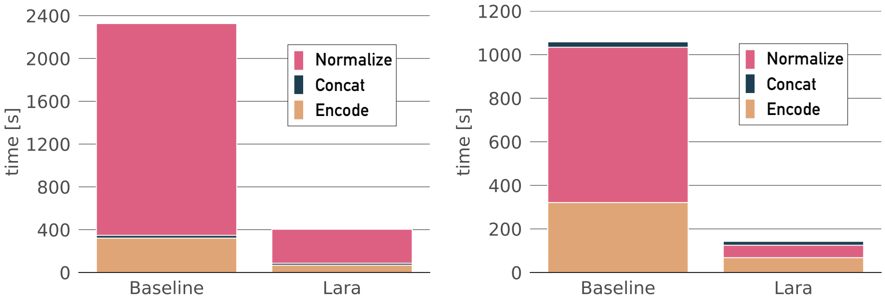
[6] http://emma-language.org/

[7] http://labs.criteo.com/2013/12/download-terabyte-click-logs-2/

[8] http://files.pushshift.io/reddit/comments/

Discussion. *Baseline* shows the runtime for the pipeline executed without any optimizations. *Lara* depicts the results with operator fusion and function composition (cf. Section 4.2.2). Figure 4.8a 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 4.8b 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 4.9 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 and out of memory errors.

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 4.9 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 Transform only supports the unoptimized DirectRunner and Google Dataflow as execution backends of Apache Beam in the moment, Tensorflow runs on Apache Beam, but only supports Google Dataflow and the unoptimized



(a) w/o pushdown **(b)** w/ pushdown

Figure 4.8: Preprocessing steps in detail on a 5GB sample.

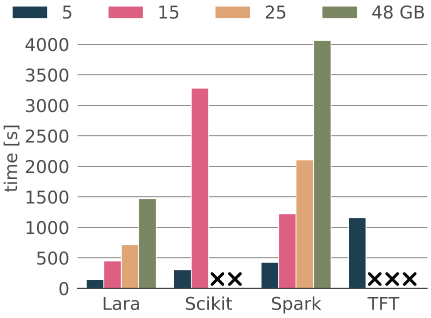


Figure 4.9: Preprocessing on different data sizes and systems.

DirectRunner as back-ends 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.

4.3.2 Operator Pushdown

In this experiment series, we evaluate the effects of operator pushdown based on Line 6 – 14 in the introductory example in Section 4.1.1, 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 [WDL$^+$09] of the n-grams obtained from the `user-name` ($n = 2$) and `comment-text` ($n = 10$) to a sparse vector space of 10000 and 50000.

Discussion. As described in Section 4.3.1, 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

Table 4.3: 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
	BLAS	0.02 ± 0.001 s	0.02 ± 0.001 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
	BLAS	1.42 ± 1.514 s	1.69 ± 0.693 s

require a conversion beforehand. It is important to note that the pushdown is only possible because the filter is applied to 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 applied on the `user-name` and `comment-text` has no inverse and would prevent the filter pushdown.

Results. Lara takes 120.60 ± 7.30 seconds to create the matrix representation of the filtered join result, while it takes 881.41 ± 27.95 seconds to run the pipeline without filter pushdown. Thus, filter pushdown achieves an performance improvement of 7.3×.

4.3.3 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 4.1, Line 17 – 20. It calculates the solution directly using a 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 4.4. We evaluate both algorithms on synthetic datasets with 10000 rows and 1000 columns. We use sparse data with 10 percent non-zero values for the logistic regression experiments.

Discussion. All results are depicted in Table 4.3. 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 4.5a for row-wise and column-wise features. *BLAS* executes the variant shown in Figure 4.5b for both layouts. For row-wise features, *BLAS+Convert* executes the plan shown in Figure 4.5c. 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.

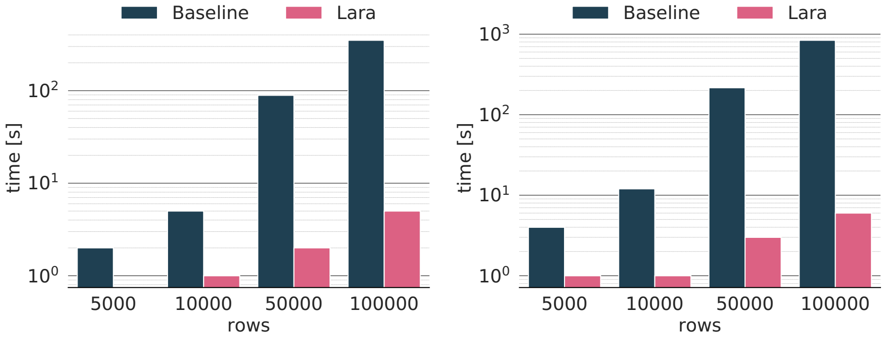
```
1 var weights = Vector(...) // initialize weight vector
2 for (_ <- 0 until Iterations) {
3     val hyp  = X ** w
4     val exp  = hyp.map(value => 1 / (1 + math.exp(-1 * value)))
5     val loss = exp - y
6     weights  = weights - alpha * ((X.t ** loss) / X.numRows)
7 }
```

Listing 4.4: Logistic regression with BGD in Lara.

The results for logistic regression are conducted on sparse data (10 percent non-zero values) and we ran 1 and 100 iterations. Analogous to the ridge regression example, we conducted the experiments on row- and column-wise features **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 4.4): for row-wise features, *Breeze+Convert* converts the feature matrix **X** used in the multiplication with the loss vector `X.t ** loss` (cf. Listing 4.4, Line 6); for column-wise features, *Breeze+Convert* introduces an enforcer for **X** read in the multiplication with the weight vector `X ** w` (cf. Listing 4.4, Line 3). *BLAS* represents the results for dense matrix representation.

Results. The benchmarks for ridge regression on dense data show the importance of specialized physical operators. Using the BLAS subroutines is 51.8× faster for column-wise and 122× faster for row-wise features compared to the native Scala implementation. For the Scala implementation, the column-wise layout matches the properties of the operators (cf. Figure 4.5a) and is 2.4× faster than the row-wise 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 instructions can not overcome the overhead that is introduced by the layout 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 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. An 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. The execution on a dense format using BLAS instructions is faster for 1 iteration, but its performance degrades for 100 iterations and is 1.54× slower for column-wise and 1.9× slower for row-wise features compared to the sparse implementation. This is due to the sparsity of the data, which results in faster execution for sparse formats in case the right access patterns are established.



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

Figure 4.10: Cross-validation with ridge regression.

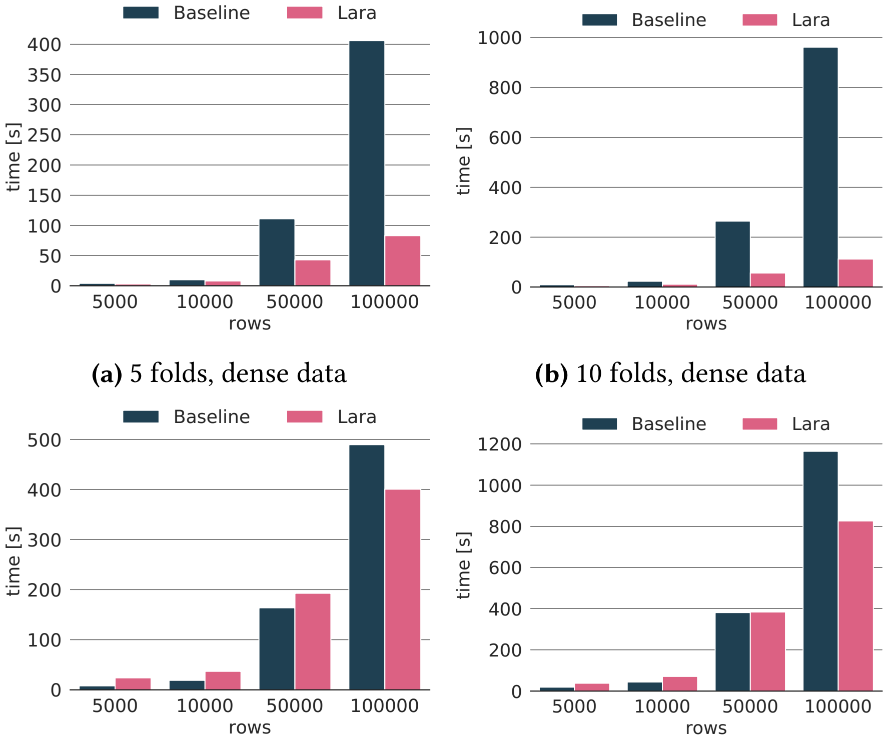
4.3.4 Cross-Validation

In this experiment series, we evaluate the impact of the proposed rewrites for cross-validation. We first evaluate ridge-regression as shown in Listing 4.1, Line 17 – 20. Next, we evaluate logistic regression with batch gradient descent (BGD) that calculates the model iteratively over a fixed number of iterations (cf. Listing 4.4). Both algorithms are evaluated on synthetic datasets. Each figure depicts the results for 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 4.10 depicts the results for dense data. The *Baseline* implementation executes the algorithm without the proposed optimizations for cross-validation described in Section 4.2.4. *Lara* executes the matrix-matrix and matrix-vector multiplications in a Pre-Computation step on each split before the cross-validation iterations are executed. Table 4.2 shows the time complexity for operations in the baseline and Lara.

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 4.2. 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 $\mathbf{X}$. Thus, Lara enables users to explore up to a magnitude more models in the same time, which potentially results in better models.

Logistic Regression Discussion. Figure 4.11a and 4.11b depict the results on dense data, while Figure 4.11c and 4.11d 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 $\mathbf{loss}$ to calculate the gradient. Lara is able to extract these operations to lower the computational complexity, as described in Section 4.2.4. The Pre-Computation of the hypothesis *hyp* can now be calculated in a single matrix multiplication $\mathbf{X}_k\mathbf{W}$ by stacking all weight vectors w_k into a matrix $\mathbf{W}$.



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

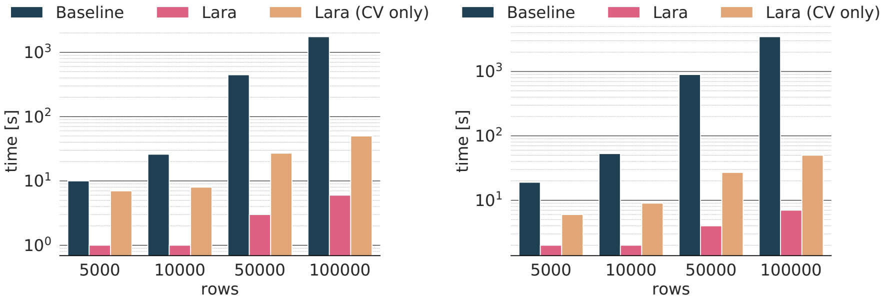
(c) 5 folds, sparse data **(d)** 10 folds, sparse data

Figure 4.11: Cross-validation with logistic regression.

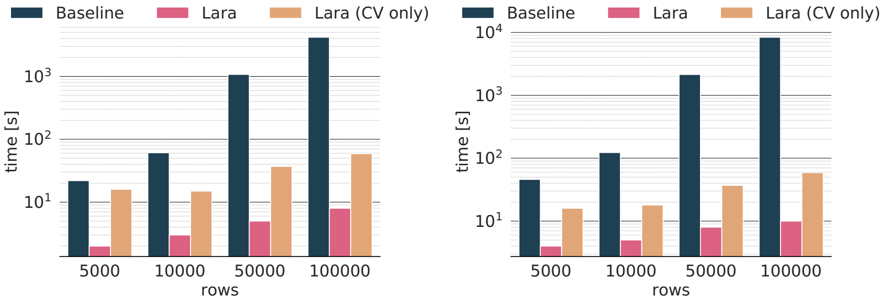
Logistic Regression Results. On dense data, Lara is up to 4.8× faster for 5 folds and 8.6× faster for 10 folds than the baseline (Figure 4.11a and 4.11b). The impact of the redundant computations in the baseline grows with the number of rows in the training set. Additionally, Lara benefits from the more 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 4.11c and 4.11d). 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. To summarize, Lara also achieves considerable speedups in case of iterative algorithms due to the CV optimizations.

4.3.5 Hyperparameter Tuning

In this experiment series, we evaluate the performance impact of our proposed rewrites for the cross-validation utility function with hyperparameter tuning. First, we evaluate hyperparameter tuning for ridge regression. We evaluate different λ values for the regularization matrix **reg** as hyperparameters, as shown in the running example. Next, we evaluate logistic regression with BGD in a second experiment series. Here, we evaluate different initializations of the weight



(a) 5 Hyperparameters, 5-fold CV **(b)** 10 Hyperparameters, 5-fold CV



(c) 5 Hyperparameters, 10-fold CV **(d)** 10 Hyperparameters, 10-fold CV

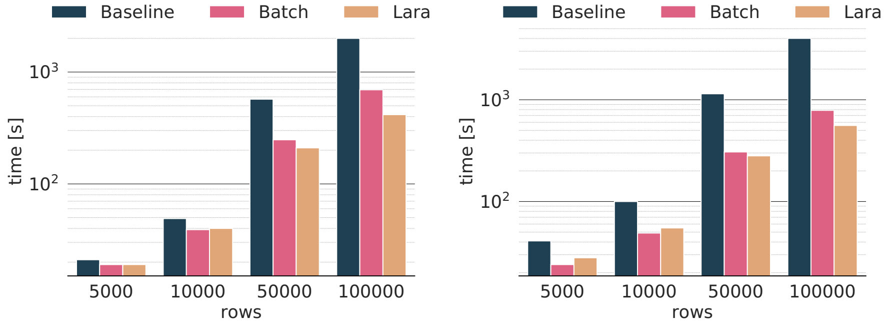
Figure 4.12: Hyperparameter tuning for ridge regression.

vector w. The feature matrix has 1000 columns in both experiment series, while we scale the number of rows. We tune for 5 and 10 different hyperparameters and validate them with 5 and 10 fold CV. The logistic regression with BGD runs for a fixed amount of 100 iterations.

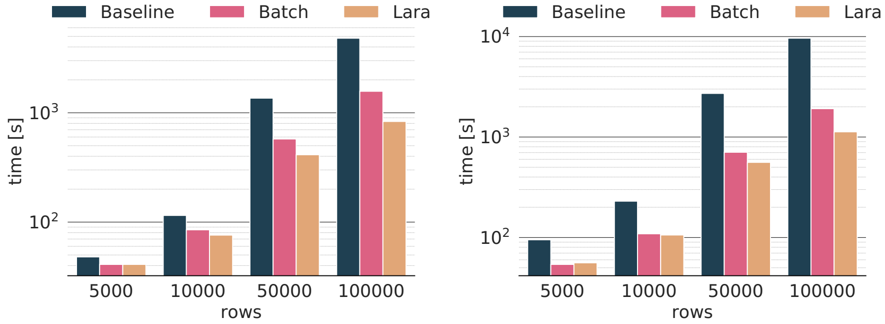
Ridge Regression Discussion. Figure 4.12 depicts the results of the benchmark. The *Baseline* executes the cross-validation and hyperparameter loop without rewrites but uses BLAS instructions. We provide experiments for two optimization variants: *Lara (CV only)* depicts the results of the cross-validation optimizations without removing loop invariant code. *Lara* depicts the results after the loop invariant code is pulled out of the hyperparameter loop (cf. Section 4.2.4).

Ridge Regression Results. The baseline implementation takes 5× and 10× longer than the single cross-validation (cf. Section 4.3.4), because it is executed for 5 and 10 hyperparameters. 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).

Logistic Regression Discussion. We evaluate the hyperparameter tuning for logistic regression on dense and sparse matrix representation with 10 percent non-zero values. Figure 4.13 depicts the results for dense matrix representation and Figure 4.14 depicts the result for sparse matrix representation of the benchmark. The baseline executes the cross-validation and hyperparameter loop without rewrites but uses BLAS instructions. We provide results for an



(a) 5 Hyperparameters, 5-fold CV **(b)** 10 Hyperparameters, 5-fold CV

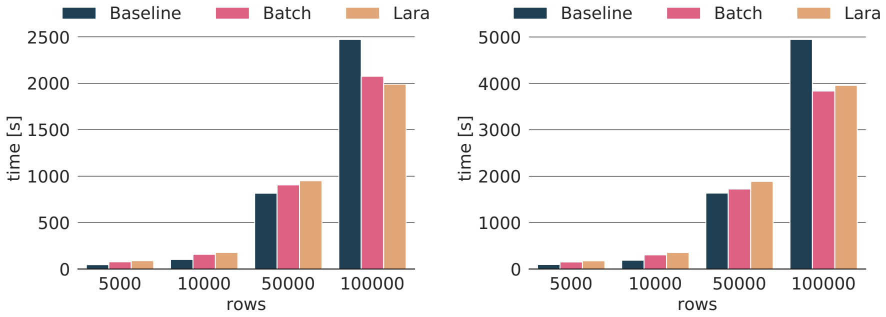


(c) 5 Hyperparameters, 10-fold CV **(d)** 10 Hyperparameters, 10-fold CV

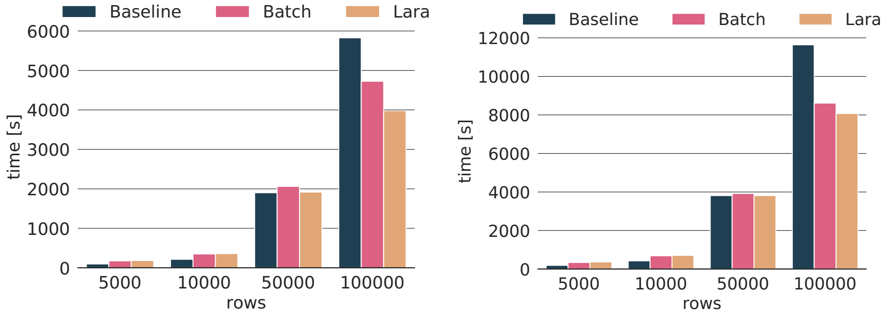
FIGURE 4.13: Hyperparameter tuning for logistic regression in dense matrix representation.

implementation that uses *batching* as additional baseline (similar to the approach presented in TuPaQ [STH+15]) 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 4.2.4. The hyperparameter loop adds an additional nesting layer: we do not use a single weight vector w per fold k, but a matrix $\mathbf{W}$, which contains all the weight candidates (hyperparameters) as columns, thus $hyp = [\mathbf{W}_0, ..., \mathbf{W}_k]$. Therefore, the resulting Pre-Computation for the individual folds involves k^2 iterations, as depicted in Listing 4.5.

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×. Batching performs better with increasing numbers of hyperparameters, whereas Lara gains a performance advantage for an increasing number of folds for the CV.
For the sparse matrix representation, we can observe that the baseline outperforms batching and Lara until a scaling factor of 50000 rows. For a small number of rows, the baseline is up to 1.5× faster. Until 10000 rows, Lara and batching provide comparable performance. Batching performs best with a large number of hyperparameters and a small number of folds in comparison to



(a) 5 Hyperparameters, 5-fold CV **(b)** 10 Hyperparameters, 5-fold CV



(c) 5 Hyperparameters, 10-fold CV **(d)** 10 Hyperparameters, 10-fold CV

Figure 4.14: Hyperparameter tuning for logistic regression in sparse matrix representation.

the other approaches: it slightly outperforms Lara by 1.1× and the baseline by 1.3× for 10 hyperparameters and 5-fold CV. In contrast, Lara achieves its best performance with a small number of hyperparameters and a large number of folds: it outperforms batching by 1.2× and the baseline by 1.5× for 5 hyperparameters and 10-fold CV. 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. Even though batching does not require additional data structures, it can not exploit a more efficient instruction for the multiplication of the feature matrix with the weight matrix. In future work, we plan to integrate batching as a rewrite rule for tuning large a large number of hyperparameters.

```
for (i <- 0 until k) {
  for (j <- 0 until k) { // W_j 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
  }
}
```

Listing 4.5: Pre-Computation for logistic regression with hyperparameter tuning.

4.4 Related Work

ML Libraries & Languages. SystemML [BBE+14] and Mahout Samsara [SPQ+16] 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.
Delite [CSB+11] is a compiler framework for DSLs providing a staged IR based on higher-order functions. OptiML [SLB+11] is a DSL for machine learning based on the Delite [CSB+11] 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 [SVK+17] 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 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. [KNP15] propose learning of linear models on data in relational databases, which was later extended to linear algebra operators [CKNP17]. In this work, linear algebra operations can be pushed down to relations in databases, similar to [CS94]. control flow, UDFs and general preprocessing pipelines are not considered.
SystemML provides a `ParFOR` [BTR+14] primitive which, depending on the access patterns to the data, executes the task defined in the body of the loop in parallel. It applies several optimizations for efficient execution in both single-node and distributed environments. Similar to this work, Lara IR allows to detect task-parallelism. The loop fusion presented in Section 4.2.2 detects independent tasks (e.g., the encoding of distinct columns), but fuses them instead of executing them in parallel. Executing those in parallel is left for future work. Furthermore, SystemML introduced operator fusion [ELB+17, BRH+18], which 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. [YAB+18] 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 [STH+15] 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 [KTD+13] 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 [PTS+17] provides a functional IR and a lazily evaluated runtime API, which is used to collect operator code from different libraries, e.g., Python Pandas and scikit-learn. Its optimizations focus 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 [LGG+18] 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 [FCW+18] 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.
TensorDB [KC14] extends the array database SciDB [Bro10] with tensor-algebraic operations for tensor decomposition and users can interleave relational and tensor operators. TensorDB implements tensor decompositions in the database and thus, can decompose tensors that exceed the available main memory. It focuses on mixed relational and tensor operators and to the best of our knowledge does not support complex UDFs, e.g., for feature transformations.

4.5 Conclusion

In this chapter, we present Lara, a DSL for end-to-end machine learning pipelines for model training. 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 on the example of an end-to-end ML pipeline that yield speedups of up to an order of magnitude.

Limitations & Future Work. Our prototype is not integrated into a dedicated runtime nor uses code-generation at 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 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 the 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 [KKC+17, BRH+18, KKS+17]. Furthermore, one could extend the combinator view by integrating more data representations (e.g., block-wise or compressed [EBH+16]). To apply the layout optimizations also to intermediate results, it is essential to extend Lara to support runtime code analysis and a cost-based optimizer. Furthermore, we restricted the scope of this work to single-node execution and focus on the performance impact of our holistic optimizations. We think that hybrid parallelization and distribution strategies inspired by the hybrid approach of SystemML are other interesting directions for future work.

5

FUSED OPERATOR PIPELINES

Requirements for data analytics applications based on machine learning techniques have changed over the last years. End-to-end ML pipelines nowadays go beyond pure linear algebra and often also include data preparation and transformation steps (ETL) that are best defined using relational algebra operators. Data scientists construct feature-vector representations for training ML models by filtering, joining, and transforming datasets from diverse data sources [ZKR16] on a daily basis. This process is often repeated many times in an *ad-hoc* fashion, as a variety of features are explored and selected for optimal predictive performance. Such pipelines are most conveniently expressed in languages with rich support for both ETL and ML tasks, such as Python or R, but these implementations do not scale. In enterprise setups, the source data usually resides in a data warehouse. One possible strategy in such situations is to run the ETL part of the pipeline in situ, and the ML part in a specialized engine such as SciDB [Bro10] or RasDaMan [BDF+98]. This approach has two drawbacks. First, moving data between engines is an expensive operation that is frequently repeated as the pipeline is refined. Second, it does not allow to easily join warehouse and external data sources without support by the system [TÖZ+16].

Parallel dataflow engines such as Spark [ZCF+10] or Hadoop [Apa] offer a more flexible execution infrastructure that does not suffer from the problems outlined above. Initially developed for ETL-like workloads, these systems have been increasingly used by practitioners to implement ML algorithms [MBY+16, BDE+16, SPQ+16]. To support scalable execution of ML workloads, the functionality of established libraries for scalable linear algebra, such as ScaLAPACK [CDPW92], is being implemented on top of parallel dataflow systems by projects like SystemML [GKP+11], MLlib [MBY+16], Apache Mahout Samsara [SPQ+16], and Pegasus [KTF09]. A common runtime engine avoids data transfer, but the mismatch in data representation still manifests itself when executing mixed analytics pipelines. While dataflow engines typically row-partition large datasets, scalable linear algebra operators are implemented on top of block-partitioned, or *blocked* matrices. The difference in the partitioning assumptions results in a re-partitioning barrier whenever a linear algebra operator follows a relational one. The dataflow engine has to re-partition the entire row-partitioned dataset into a block-partitioned matrix. One possible solution would be to execute linear algebra operators on row-partitioned matrices. Although this performs well for operations that access one row at a time (e.g., to calculate the sum for each row), superlinear operations, such as matrix multiplication that consume multiple rows and/or columns become very inefficient [GKP+11]. For computational and storage efficiency, the majority of scalable linear algebra frameworks perform matrix multiplications on blocked matrices [GKP+11, HBY13, KTF09].

Research Contribution. In this chapter, we demonstrate the optimization potential of fusing relational and linear algebra operators.[1] As a first step, we focus on a common pattern – a relational join, followed by a per-element transformation for feature extraction and vectorization, and a subsequent matrix conversion. To reduce the total shuffling costs of this operator chain, we propose *BlockJoin*, a specialized distributed join algorithm that consumes row-partitioned relational data and directly produces a block-partitioned matrix. We focus on the major drawback posed by an independent operator chain: The intermediate result of the join, row-wise partitioned by the join key, is discarded immediately to form a block-partitioned matrix. This materialization implies the risk of heavy load data on a few nodes for skewed data, which results in performance degradation. Even more important, it results in an unnecessary shuffle operation for the join in general. BlockJoin avoids the materialization of the intermediate join result by applying the vectorization function and the successive block partitioning independently to both relations. Analogous to joins that have been proposed for columnar databases [LR99, BMK99, THS+09, AMH08], BlockJoin builds on two main concepts: *index joins* and *late materialization*. More specifically, we first identify the matching tuple pairs and their corresponding row indexes in the matrix by performing a join on the keys and tuple-ids of the two relations (analogous to TID-Joins [MR94]). Based on the gathered metadata, we apply the vectorization function separately to the matching tuples of both relations, and repeat this for the block partitioning, without having to materialize the intermediate join result. Therefore, we can apply different materialization strategies for the matrix blocks based on the shape of the input relations, namely *Early* and *Late* materialization. Our experiments show that BlockJoin performs up to 6× faster than the state-of-the-art approach of conducting a row-wise join followed by a block-partitioning step.

In summary, we make the following contributions:

- We demonstrate the need for implementing relational operators producing block-partitioned datasets (cf. Section 5.1.2).
- We propose BlockJoin, a distributed join algorithm which produces block-partitioned results for workloads mixing linear and relational algebra operations. To the best of our knowledge, this is the first work proposing a relational operator for block-partitioned results (cf. Section 5.2).
- We provide a reference implementation of BlockJoin based on Apache Spark with two different block materialization strategies (cf. Section 5.3).

---

[1] This chapter is based on [KKS+17].

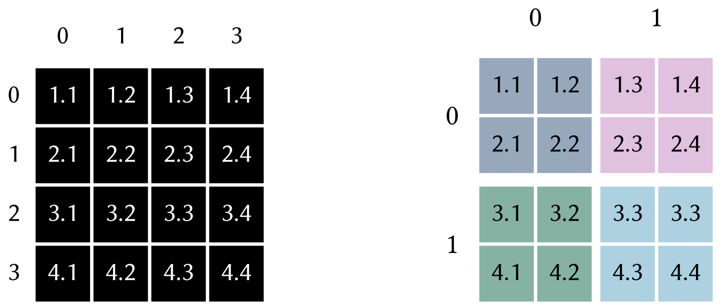
- We provide a cost model to select the best-suited materialization strategy based on the shape of the input tables (cf. Section 5.2.4).
- We experimentally show that BlockJoin outperforms the baseline approach in all scenarios and, depending on the size and shape of the input relations, is up to 6× faster. Moreover, we show that BlockJoin is skew resistant and scales gracefully in situations when the state-of-the-art approach fails (cf. Section 5.4).

5.1 Background

In this section, we introduce the blocked matrix representation. We also discuss a running example we will use throughout the chapter and discuss the state-of-the-art implementation for dataflow systems.

5.1.1 Block-Partitioned Matrix Representation

Distributed dataflow systems use an element-at-a-time processing model in which an element typically represents a line in a text file or a tuple of a relation. Systems that implement matrices in this model can choose among a variety of partitioning schemes (e.g., cell-, row-, or column-wise) for the matrix. For common operations such as matrix multiplications, all of these representations incur huge performance overheads [GKP$^+$11]. Block-partitioning the matrix provides significant performance benefits. This includes a reduction in the number of tuples required to represent and process a matrix, block-level compression, and the optimization of operations like multiplication on a block-level basis. These benefits have led to the widespread adoption of block-partitioned matrices in parallel data processing platforms [GKP$^+$11, HBY13, KTF09]. A blocked representation splits the matrix into sub-matrices of fixed size, called *blocks*, as depicted in Figure 5.1. These blocks become the processing elements in the dataflow system. Fixed-sized blocks greatly simplify the necessary join operations in distributed systems for element-wise and matrix-multiplication. Fixed-sized squared blocks simplify the combination of blocks using different dimensions for the join predicate, e.g., to join two matrices **A** and **B** on **A**`.block-row-index` = **B**`.block-column-index` for matrix-multiplication.



(a) 4 × 4 matrix **(b)** 2 × 2 *blocked* matrix

Figure 5.1: Block-wise matrix partitioning.

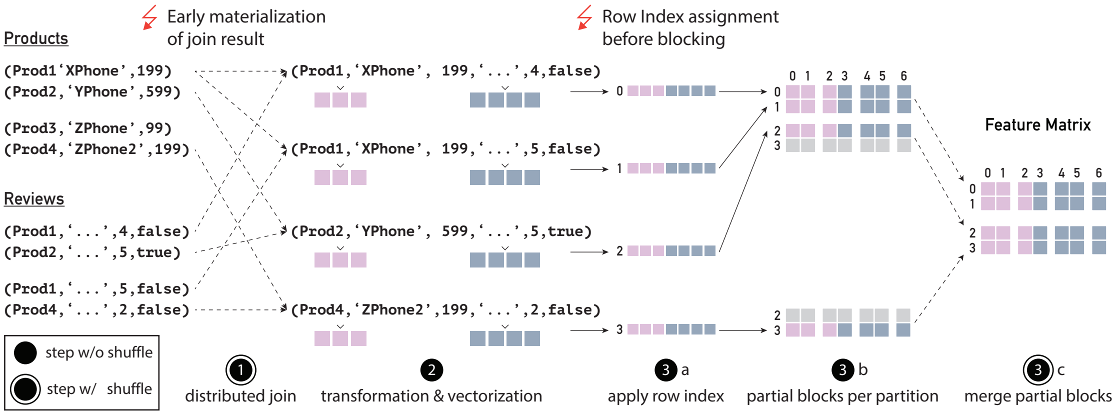


Figure 5.2: The baseline implementation for our running example. We prepare the data in order to learn a Spam classifier in an e-commerce use case: ❶ we perform a distributed join of Products and Reviews, ❷ call user code to transform the join result into feature vectors, and ❸ **a** assign consecutive rows indexes to the resulting feature vectors, ❸ **b** create partial blocks per partition, and ❸ **c** merge them into a blocked matrix.

5.1.2 Motivating Example

Our running example is learning a Spam detection model, a common use case in e-commerce applications. Assume that customers write reviews for products, some of which are Spam, and we want to train a classifier to automatically detect the Spam reviews. The data for products and reviews are stored in different files in a distributed file system. We need the attributes from both relations to build the features for the model in our ML algorithm. Therefore, we first need to join the records from these tables to obtain reviews with their corresponding products. Next, we need to transform these product-review pairs into a suitable representation for an ML algorithm. To this end, we apply a UDF that transforms the attributes into a vector representation. Finally, we aggregate these vectors into a distributed, blocked feature matrix to feed them into an ML system (e.g., SystemML). Figure 5.2 illustrates how to execute such a workload. Listing 5.1 shows how it can be implemented in a distributed dataflow system like Spark, expressing a mixed linear- and relational-algebra pipeline. We will refer to this as *baseline* implementation in the rest of this chapter. The input data resides in the tables `Products(product_no, name, price, category)` and `Reviews(product_no, text, num_stars, is_spam)`. The following steps are depicted in Figure 5.2 and Listing 5.1 accordingly. Step ❶ performs a foreign-key join on the `product_no` attribute. Step ❷ applies user-defined vectorization functions to each row of the join result, to transform it into vector-based features, using techniques such as feature hashing and *one-hot-encoding* [Gru15]. We assume that the vector resulting from a row is a concatenation of the vectorization of the input tuples of the participating relations. Step ❸ is split into three sub-steps that are necessary to form a block-partitioned matrix: ❸ **a** : creates a sequential index for the join result that is used as a row index for the matrix. This is necessary, as dataflow engines, in contrast to database systems, do not provide a unique tuple identifier. ❸ **b** : builds the initial matrix blocks by splitting the rows at block boundaries. ❸ **c** : in a final aggregation step, where partially filled blocks (which span multiple data partitions) are merged.

```
val Products: Dataset[Product] = // read csv...
val Reviews:  Dataset[Review]  = // read csv...
val JoinResult = Products.joinWith(                                        ❶
    Reviews,
    Products("product_no") === Reviews("product_no")
)
// Vectorize each tuple in the join result
val Vectorized = JoinResult.map { case (p, r) =>                           ❷
    val pv = vectorizeProduct(p)
    val rv = vectorizeReview(r)
    pv ++ rv
}
// Convert `Vectorized` into blocked matrix `M`
val M = toMatrix(Vectorized)                                               ❸
```

Listing 5.1: Code snippet for the running example.

5.2 Blocking Through Joins

In this section, we present BlockJoin, our chained, context-aware operator, leveraging the example of Figure 5.2. We first introduce a baseline implementation of independent operators for that example, which cannot leverage join metadata for the blocking phase. We then detail BlockJoin in Section 5.2.1 and Section 5.2.2, and discuss its improvements.

Drawbacks of an independent operator chain. The baseline implementation, which uses independent operators, is illustrated in Figure 5.2 and proceeds as follows: We first partition `Products p` by its primary key `p.product_no` and `Reviews r` by its foreign-key `r.product_no` to execute the distributed join. After vectorizing the join result `Vectorized v`, we introduce a consecutive index (e.g., by a *zipWithIndex* method in Spark), called *row-idx*, to uniquely identify each tuple. Then, we split each `v` of `Vectorized` into its components, based on the *col-idx*, and re-partition by the block index of the resulting matrix. The block index is obtained by:

$$block\text{-}idx(v, col\text{-}idx) = \{\frac{v.row\text{-}idx}{\textbf{block\_size}}, \frac{col\text{-}idx}{\textbf{block\_size}}\}$$

The `block_size` represents the number of rows and columns in a block. Although matrix blocks can have arbitrary row- or column-sizes, we use square blocks, for the sake of simplicity. One can easily derive the function for non-square blocks by substituting `block_size` with the number of rows and column per block. In general, squared blocks are more efficient as they do not require costly alignment when different dimensions are joined as described in Section 5.1.1. We observe that an independent operator chain (cf. Figure 5.2) has to re-partition the data twice and materializes the join result, even though this result is split according to block boundaries immediately after applying the index assignment in Step ❸ **a**. Thus, the costly join is only executed to create a sequential index for the rows of the matching tuples in the matrix. Another danger during materialization of the join result is that the two input tables can be very wide, and we, therefore, risk running out of memory when executing the join.

In the following, we introduce BlockJoin and explain how it avoids materializing the intermediate join result by introducing information exchange between the operators. We start by discussing a simplified case in Section 5.2.1, and extend our solution to the general case in Section 5.2.2.

5.2.1 BlockJoin under Simplifying Assumptions

For the sake of clear presentation, we introduce two assumptions solely to the blocking, we drop these assumptions in the next section and describe how to apply BlockJoin for general equi-join cases: (i) the join keys on both relations are *consecutive* integers and the relations are ordered by their keys; (ii) there is a strict $1:1$ relation between the tables, that is: they have the *same cardinality* and the *same values* in their primary key. Joining two relations, which fulfill these conditions, is equivalent to *concatenating* the relations. Moreover, the cardinality of the join result will be the same as the cardinality of the two joined relations. Now, suppose that we want to block-partition the join result of the two relations. The question we are going to answer throughout the rest of this section is:

Can we achieve joined and block-partitioned results, without first materializing the join result in a row-partitioned representation?

Blocking without materializing the join result. Given our simplifying assumptions, we can safely treat the key `product_no` as the unique, sequential identifier of each tuple. Hence, we can not only use it as join key, but but can also define $v.\boldsymbol{row\text{-}idx} = v.\boldsymbol{product\_no}$, to uniquely identify the rows in the resulting matrix. Now, as we do not need to materialize the join result to obtain the *row-idx*, we discuss how we apply the blocking function on both relations independently after the vectorization. The first component of the *block-idx* function $(\frac{v.\boldsymbol{row\text{-}idx}}{\textbf{block\_size}})$ assigns the row index of the block *blk-row-idx*, which the cells in a row belong to. Due to our assumptions, matching tuples already share the same *row-idx*. The second component of the *block-idx* function $(\frac{v.\boldsymbol{col\text{-}idx}}{\textbf{block\_size}})$ defines the column index of the block *blk-col-idx*, which the cells of a rows are split across. We can use this part of the equation on the individual tables without joining after we apply some small changes: the function has to account for the fact that the *blk-col-idx* of the second relation have to be offset by the number of columns in the first relation (because the result concatenates the two relations). Thus, we add the offset `cols(pv)` (i.e., the number of columns of the vectorized version of the first relation `p`) to the column index of the second relation.[2] Equation 5.1 shows the modified *block-idx* function that is applied on the vectorized tuples of the individual input relations.

$$\begin{aligned} block\text{-}idx_{\mathbf{P}}(pv, col\text{-}idx) &= \{\frac{pv.\boldsymbol{row\text{-}idx}}{\textbf{block\_size}}, \frac{\boldsymbol{col\text{-}idx}}{\textbf{block\_size}}\} \\ block\text{-}idx_{\mathbf{R}}(rv, col\text{-}idx) &= \{\frac{rv.\boldsymbol{row\text{-}idx}}{\textbf{block\_size}}, \frac{\textbf{cols(pv)} + \boldsymbol{col\text{-}idx}}{\textbf{block\_size}}\} \end{aligned} \tag{5.1}$$

---

[2] Section 5.3 details how we determine this value at runtime.

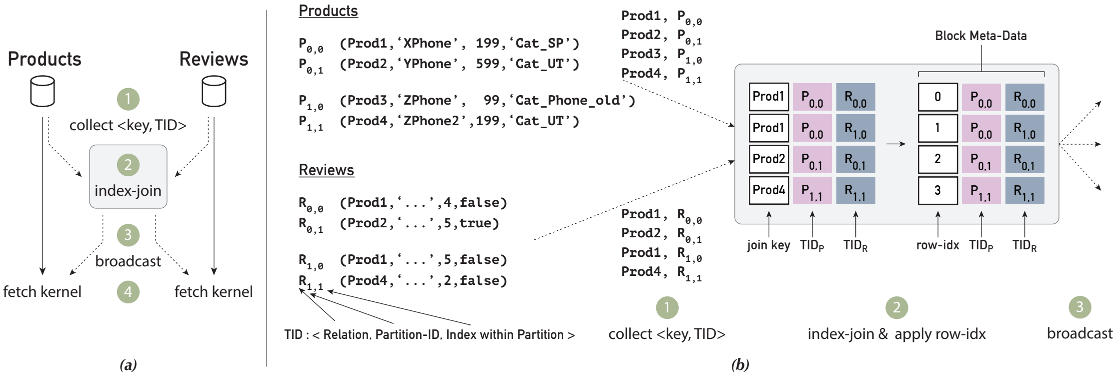
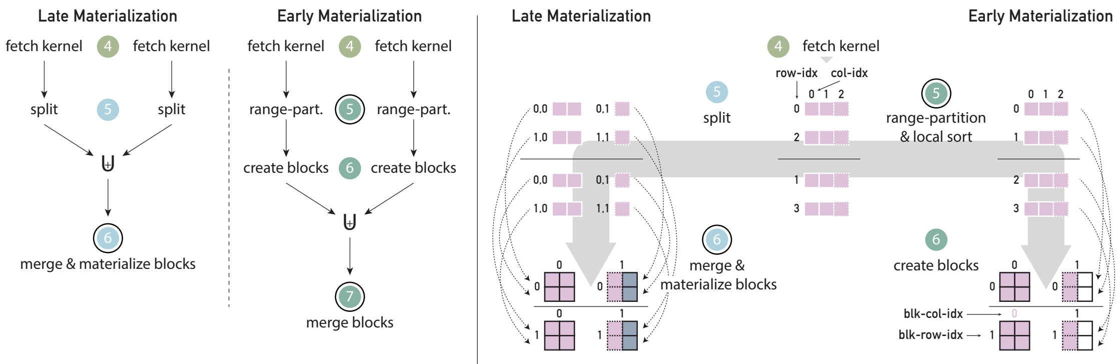


Figure 5.3: Local index-join & sequential row index assignment for the running example: ① we collect the `<key, TID>` pairs on the join coordinator, ② we perform an index-join on the collected tuples and introduce the sequential row index `row-idx` on the result. Afterwards, ③ we broadcast the result back to the nodes. The fetch-kernel ④, is shown in Figure 5.4.

5.2.2 BlockJoin for the General Case

The simplifying assumption of an ordered, consecutive index on both relations from the previous section obviously does not hold in reality. In real-world scenarios, we observe *primary-key (PK) – foreign-key (FK)* or `1:N` relationships, such as users and items, items and reviews, or even `M:N` relations, as well as normalized database schemata [KNP15]. Therefore, we cannot use the keys of the individual relations to determine the corresponding blocks of the tuples. Moreover, the size of the input relations may vary compared to the join result. For instance, a `Product` can match arbitrarily many `Reviews`. In the subsequent paragraphs, we showcase how BlockJoin works under general conditions.

Assigning indexes to tuple pairs in the join result. BlockJoin first obtains a unique surrogate key *TID* from each tuple of both relations independently. The TID consists of a `<relation-id, partition-id, index-within-partition>` triple as depicted in the bottom left part of Figure 5.3 *(b)*. The triple uniquely identifies each row of the relations. In the next step, we generate the unique identifier *row-idx* for the rows in the *resulting* Matrix `M`. In order to assign the identifier to the matching tuples of both relations, we design a variant of the index-join [DKO$^+$84, MR94]. The main idea of the index-join is to project the key and TID columns of the two relations to determine matching tuples without materializing the *payload* columns. As depicted in Figure 5.3, Step ① projects and collects the `<key,TID>` pairs from both relations on the driver. Therefore, we have all keys of the two relations and execute an *index-join* ②. Based on the result, we assign the *row-idx* to the matching tuples. We call this phase *join-kernel*, following the nomenclature of [THS$^+$09]. In Step ③, we make the *metadata*, which contains the matched `<key,TID>` pairs and *row-idx*'s, available on all nodes for the subsequent *fetch-kernel* phase. Based on the information in the metadata, we prune all non-matching tuples and apply the vectorization function to the remaining tuples ④ on each relation separately. While we can use the very same *block-idx* function (cf. Section 5.2.1, Equation 5.1), we elaborate on two different blocking strategies, enabled by applying the *row-idx* separately, in the next section.



(a) *(b)*

FIGURE 5.4: Block-materialization strategies. We illustrate the materialization strategies for the `Products` relation on the right side of the Figure *(b)*. *Late materialization* breaks the tuples into multiple row-splits locally ⑤ and merges the splits of both relations after union them ⑥. *Early materialization* first range partitions the complete rows in order to group tuples belonging to the same blocks and then performs a local sort on the row index to enable faster block creation ⑤. After materializing the blocks per relations ⑥, potentially partial blocks are merged ⑦.

5.2.3 Block Materialization Strategies

Figure 5.4 *(a)* sketches the two materialization strategies for BlockJoin. Both approaches share the initial Steps ① to ③ from Figure 5.3, explained in the previous section. The main difference stems from the *block materialization* strategy that we use for the values emitted in Step ④. Our goal now is to shuffle the row-splits of each row to the nodes responsible for the splits' destination blocks.[3] A very important consideration is that one row-split may need to fill multiple rows in the same block and might be part of multiple blocks. For instance, consider a row-split of a product which matches multiple reviews. If there are 10 matches and the `block_size` is 5, that product's row-split will have to be duplicated 10 times and, therefore, contribute to at least 2 different blocks. Duplicates can have a huge impact on the runtime of the block materialization phase. For this reason, we devise two materialization strategies which are detailed below.

Late Materialization. The left side of Figure 5.4 *(b)* depicts the execution flow of late materialization. The key idea behind late materialization is to reduce the number of row-splits emitted, by sending each split only *once per destination block*, even if the row-split occurs multiple times in the respective block. The duplicates of each split are materialized on the receiver side for each block. We can apply receiver-side materialization, as we are not forced to materialize the join result (like in the baseline), to obtain the sequential *row-idx*. More specifically, each row emitted from the fetch kernel ④ is split in multiple `<blk-idx, row-offset, duplicates, row-split>` tuples ⑤. Since there might be multiple matches for a key, we store the number of `duplicates` per block, instead of materializing them early. The `row-offset` defines the first row-index of the row-split in the destination block. In the destination node, we merge the

---

[3] Given a row r, a row-split is a tuple which contains a strict subset of the columns or r. The purpose of a row-split is to fit in a given block. For instance, given a block size of 2, a row with 6 columns will be split into 3 row-splits.

Table 5.1: Cost model notation.

Symbol	Meaning
$\|T\|$	Number of rows in relation T
$\|\|T\|\|$	Number of partitions in relation T
$cols(T)$	Number of columns in relation T
$bytes(T)$	Size (bytes) of a tuple in relation T
b	Number of rows/columns per square block
P, R	Input tables of the join
J	Join result

row-splits of the same `blk-idx` and create the complete blocks by materializing their duplicates ⑥. Note that we create complete blocks even in the case they contain data from both relations in one pass (as can be seen for the green cells from the Reviews table).

Early Materialization. The right side of Figure 5.4 *(b)* depicts the execution flow of early materialization. Instead of separating the rows from the fetch kernel ④ into row splits immediately, we emit *a single* `<row-idx, duplicates, row>` tuple per row. Rows matching multiple times are not yet materialized, and we emit one tuple for all duplicates within a block again. In the next step, we *range-partition* the tuples by their `row-idx` and sort within each partition ⑤. A custom partitioner ensures that tuples belonging to the same block end up in the same partition. Next, we create the blocks and materialize the duplicates for each relation separately ⑥. Note that we do not have to shuffle, but potentially create partial blocks (as can be seen for the blocks with column index 1). In the last step, we `union` the relations and merge the partial blocks ⑦. In summary, early materialization replaces the shuffle phase for the join in the baseline with a custom range partitioning, which establishes partitions that match the block size. After the rows are partitioned, early materialization materializes the duplicates for each block and merges partial blocks.

Applicability to the baseline. While we can apply the presented materialization strategies also in the baseline, we do not gain any advantage. The main benefit of late materialization is the receiver-side materialization of duplicates (e.g., PK matching multiple FKs). In the baseline though, we materialize all duplicates during the distributed join phase. As a result, we shuffle the same amount of data as in the baseline, but with a much larger amount of tuples, as we split the rows in late materialization. The advantage of early materialization yields from the custom partitioner, which ensures partitions that do not span over block boundaries. In BlockJoin, we introduce the shuffle needed for this partitioner, as we do not shuffle for the distributed join that is required in the baseline. In the baseline, we would have to introduce the custom partitioner after the row-index is generated for the already materialized join result. Thus, this would introduce yet another shuffle step, making it worse than the actual baseline.

5.2.4 Choosing a Materialization Strategy

To make these trade-offs between late and early materialization more concrete, we compare the two materialization strategies against the baseline implementation described in Section 5.1.2. We base our comparison on the cost model shown below, using the symbols from Table 5.1. For brevity and simplicity, we focus only on the amount of data exchange and the number of tuples during the shuffling phases, and make the simplifying assumption that all the tuples of the two input relations survive the join, which also reflects the worst case for our materialization strategies. On the one hand, late materialization emits multiple row-splits per row, thus increases the number of tuples to be shuffled. On the other hand, early materialization emits full (and materialized) blocks at the expense of an extra range-partitioning on complete rows and local sorting step. Since the blocks in the early materialization schema are complete, apart from blocks containing columns from both relations (which is equal to the number of row-wise blocks), only those have to be considered during the merging process.

Size of Shuffled Data

$$\begin{aligned}
&baseline && \rightarrow |P| \cdot bytes(P) + |R| \cdot bytes(R) && \textit{join}\\
& && + |J| \cdot bytes(J) && \textit{merge blocks}\\[1em]
&early && \rightarrow |P| \cdot bytes(P) + |R| \cdot bytes(R) && \textit{range-partition}\\
& && + |J| \cdot bytes(J) && \textit{merge blocks}\\[1em]
&late && \rightarrow |P| \cdot bytes(P) + |R| \cdot bytes(R) && \textit{merge blocks}
\end{aligned}$$

Deriving the size of shuffled data for the baseline implementation is straightforward: we execute a shuffle in order to perform the join ($|P| \cdot bytes(P) + |R| \cdot bytes(R)$) and another shuffle of the join results for block-partitioning them ($|J| \cdot bytes(J)$). The early materialization strategy has to shuffle the input data in order to range-partition it ($|P| \cdot bytes(P) + |R| \cdot bytes(R)$) and shuffle the join result in order to merge the blocks ($|J| \cdot bytes(J)$), as we might have partially filled blocks. Finally, the late materialization strategy only needs to shuffle once to merge all row-splits in their corresponding block ($|P| \cdot bytes(P) + |R| \cdot bytes(R)$). The late materialization strategy is expected to have the least amount of data shuffling. However, the number of tuples exchanged differs among the three implementations.

Number of Shuffled Tuples

$$\begin{aligned}
&\mathit{baseline} && \rightarrow |P| + |R| && \mathit{join}\\
& && + \tfrac{|J|}{\mathbf{b}} \cdot \tfrac{cols(J)}{\mathbf{b}} && \mathit{merge\ blocks}\\[1em]
&\mathit{early} && \rightarrow |P| + |R| && \mathit{range\text{-}partition}\\
& && + (\tfrac{|J|}{\mathbf{b}} \cdot \tfrac{cols(J)}{\mathbf{b}}) + \tfrac{|J|}{\mathbf{b}} && \mathit{merge\ blocks}\\[1em]
&\mathit{late} && \rightarrow |J| \cdot \tfrac{cols(J)}{\mathbf{b}} && \mathit{merge\ blocks}
\end{aligned}$$

The number of tuples exchanged for the baseline implementation includes the relations themselves ($|P|+|R|$), plus the total number of blocks that form the final matrix. The number of blocks is defined by the rows in the join result divided by the block size ($\frac{|J|}{\mathbf{b}}$) and the number columns, divided by the block-size ($\frac{cols(J)}{\mathbf{b}}$). The early materialization strategy will require an extra $\frac{|J|}{\mathbf{b}}$ for the partial blocks that span both relations (detailed in the Block Materialization paragraph of Section 5.3). In the late materialization strategy, we emit each matching row of both relations ($|J|$) multiplied by the number of splits per row ($\frac{cols(J)}{\mathbf{b}}$). Intuitively, late materialization always emits more tuples than early materialization and the baseline, because each row of the result is split while the early materialization creates (partial) blocks before shuffling.

Estimating Cost. Intuitively, one could only use the size of the shuffled data to determine the cost of the different approaches. In this case, the cost model suggests that Late Materialization is always the fastest approach. Our experiments revealed that the choice between Early Materialization and Late Materialization materialization depends heavily on the shape of the input data and the number of duplicates in the join result. Thus, our cost model also takes the number of shuffled tuples into account. We provide a more detailed discussion in our experiment Section 5.4.1. A straightforward implementation of a cost estimation function would simply calculate a linear combination of size and number of tuples and yield an estimated cost. To this end, one can train two regression models based on our previously presented formulas for computing data size and number of shuffled tuples for both materialization strategies. For early materialization, the regression $r_e = \left[d_e(\theta)\ t_e(\theta)\ 1\right]^\top \mathbf{w}_e$ predicts the runtime r_e. Here θ denotes a vector that contains the data statistics from Table 5.1 for a particular join input, $d_e(\theta)$ and $t_e(\theta)$ refer to the previously presented functions for computing data size and number of shuffled tuples for early materialization and $\mathbf{w}_e$ denotes the learned regression coefficients. Analogously, a regression model $r_l = \left[d_l(\theta)\ t_l(\theta)\ 1\right]^\top \mathbf{w}_l$ can be trained for predicting the runtime r_l for late materialization. The obtained regression coefficients depend on the actual cluster settings. Therefore, a couple of experiments must be executed to obtain a sample of different runtimes for different data characteristics, before the model can be fitted. Afterwards, the prediction model can be used to select the best suited materialization strategy for subsequent runs. We present such an instance of a trained model in our experiments and showcase its accuracy.

Using this model requires statistics on the input tables and the join result. BlockJoin gathers this information during the join kernel phase, where it determines the shape of both input relations and the size of the join result. One could train the described models based on this information and the actual runtime. Once enough data is gathered, the model can predict the strategy during runtime after the join kernel to select the appropriate fetch kernel implementation. We can also integrate the model into an optimizer, which creates an optimized plan statically before job execution (e.g., Catalyst in Spark), but have to integrate table statistics and estimations for the base tables and join result to select the best strategy.

5.2.5 Extensibility

So far we have only considered equality-joins. However, BlockJoin and the general idea of assigning unique identifiers without materializing the intermediate join result is independent of the actual join algorithm that runs locally. Thus, extending BlockJoin for theta and n-ary joins boils down to implementing a variation of the index-join used to define the matching tuples. Theta joins can be implemented by a projection of the columns required for predicate evaluation and a modified version of the shared metadata, to identify matching tuples and conduct row index assignment in the fetch-kernel. Extending BlockJoin to n-ary joins is also possible, once we identify the join results. However, this extension requires further research regarding the choice between multiple binary joins or a solution based on multi-way join algorithms, which we leave to future work.

5.3 Implementation Aspects

In this section, we present important technical aspects to consider when implementing BlockJoin in distributed dataflow systems.

Row Index Assignment. In order to block partition the join result, we need to assign consecutive row indexes to the join result. In the baseline implementation, we conduct this assignment on the distributed join result. For that, we leverage Spark's `zipWithIndex` operation, which counts the number of elements of each partition in the distributed dataset, and uses the result to assign consecutive indexes in a second pass over the data. In BlockJoin, we create the unique row indexes during the join-kernel based on the matching tuples and make them available as part of the metadata. Therefore, the assignment of row indexes to emitted tuples in the fetch-kernel phase can be done on each relation individually, without prior materialization of the join result.

Block Materialization. In the baseline implementation, we create the blocks after assigning the row index. To reduce the number of emitted partial blocks, the baseline uses a `mapPartitions` function to create the matrix blocks. This function provides an iterator over the whole partition inside the UDF. Due to the sequential row index, all rows that belong to a certain block come one after the other, which allows us to create full blocks before emitting. Thus, we only have to combine blocks that are split row-wise between two partitions in the succeeding merge step.

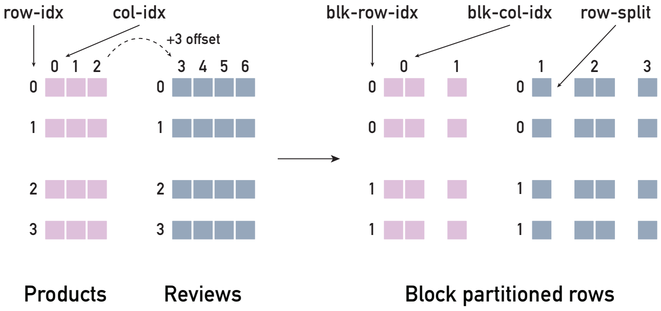


Figure 5.5: Tuples resulting from range-partitioning the vectorized tuples of `Products` and `Reviews` with block size 2 × 2.

As discussed in Section 5.2, we create the correct `block-idx` separately on both tables in the BlockJoin. Figure 5.5 shows the assignment of the block index in detail. We create partial blocks for the *blk-col-idx* 1 in both relations, as the block is split across both relations. In the late materialization approach, we have to merge all individual tuples on the receiver-side, which reduces the data that needs to be shuffled but increases the number of tuples in certain scenarios (as discussed in Section 5.2.3). In the early materialization approach, we also use a `mapPartitions` function to create full blocks on the sender-side. As we can not guarantee sorted row indexes for at least one of the relations, we would risk emitting partially filled blocks, as consecutive tuples might belong to different blocks. Therefore, we provide a custom partitioner, which creates partitions that do not cross block-boundaries. Next, we sort by the row index within each partition to create consecutive blocks. Thus, we only have to merge blocks that contain columns from both relations, e.g., for blocks with column *blk-col-idx* 1 in Figure 5.5.

Determining Matrix Dimensions. In order to assign the vectorized data to matrix blocks, it is necessary to know the dimensionality of the vectors returned by the user-defined vectorization functions upfront. One can either require the user to specify this in the vectorization functions, or alternatively fetch a single random tuple from each relation once, apply the vectorization function, and record the dimensionality of the resulting vector. In addition to the number of columns that is defined by the vectorization function, BlockJoin gathers basic statistics during the execution of the join-kernel. In particular, it calculates the number of rows of both input relations and the join result. Together, these statistics are necessary to calculate the offsets in the fetch-kernel, the custom partitioner for the Early Materialization, and the cost model.

Sparse vs. Dense Matrix Blocks Allocation. BlockJoin allocates dense or sparse blocks depending on the number of non-zero (NNZ) values, which is defined as a threshold upfront. BlockJoin determines the NNZ values during the block materialization. In the Early Materialization, it counts while the vectorized values are copied into the blocks. If the NNZ values is below the set threshold, dense blocks are converted to a sparse representation. In the Late Materialization, BlockJoin follows the same procedure during the block materialization on the receiver node.

Join Kernel. We execute the join kernel as shown in Figure 5.3. We collect the `<key, TID>` pairs, locally identify the matching tuples, assign row indexes and finally broadcast the resulting metadata. This implementation requires us to send data according to a cost model shown in the following (i.e., it reflects the block metadata table shown in Figure 5.3):

$$
\begin{aligned}
|J| \quad & \times \Big[bytes(row\text{-}index) && \qquad \textit{row index} \\
& + bytes(key) && \qquad \textit{join key} \\
& + bytes(Partition\text{-}ID) + bytes(Index\text{-}in\text{-}Partition) && \qquad \textit{TID in P} \\
& + bytes(Partition\text{-}ID) + bytes(Index\text{-}in\text{-}Partition) \Big] && \qquad \textit{TID in R}
\end{aligned}
$$

To make our implementation more memory efficient and to guarantee fast access in the fetch kernel, we implement another version of the join kernel. It solely relies on the partition-ids of a key. Instead of materializing the whole index join result, we create a hash table that contains all distinct matching keys and the overall occurrences in both relations (including duplicates). Additionally, we save the row-index of the key for each partition it occurs in for both relations. This information is enough to determine the correct row-index in the fetch kernel. This format gives us two advantages compared to the naïve version: (i) As we now use a hash table for the meta-data, we also achieve constant access times to the index information in the fetch-kernel. (ii) In this format, the memory consumption is not proportional to the size of the join result anymore, as shown in the cost model below. The model reflects the worst case in which the key occurs in each partition $||T||$ and $||R||$ of both relations.:

$$
\begin{aligned}
distinct\ |keys| \quad & \times \Big[bytes(key) && \qquad \textit{join key} \\
& + bytes(count_P) && \qquad \textit{\# key in P} \\
& + bytes(count_R) && \qquad \textit{\# key in R} \\
& + ||P|| \times bytes(offset_P) && \qquad \textit{key offset per partition of P} \\
& + ||R|| \times bytes(offset_R) \Big] && \qquad \textit{key offset per partition of R}
\end{aligned}
$$

5.4 Evaluation

In this section, we comprehensively evaluate experiments comparing BlockJoin with late and early materialization against a baseline approach on dense and sparse data. As discussed before, the baseline represents the current state-of-the-art: we use Spark to execute the join of the tables, and then SystemML to create a blocked matrix representation from the join result without staging the intermediate results on HDFS.

Sparsity mainly affects the data size and runtime, but not the overall performance trend for the algorithms. For this reason, we show the results for sparse and dense data for each experiment in the same plot. Throughout the experiments, sparse data is indicated with *patched* bars in the front, whereas dense data is indicated with *solid* bars.

Table 5.2: Size of dense data in gigabytes.

Rows Cols	10K	100K	200K	500K	1M
1K	0.2	1.7	3.4	8.5	16.9
5K	0.9	8.5	16.9	42.3	84.7
10K	1.7	16.9	33.9	84.7	169.3
25K	4.2	42.3	84.6	211.6	423.2
50K	8.5	84.7	169.3	423.3	846.7
100K	16.9	169.3	338.7	846.7	1700.0

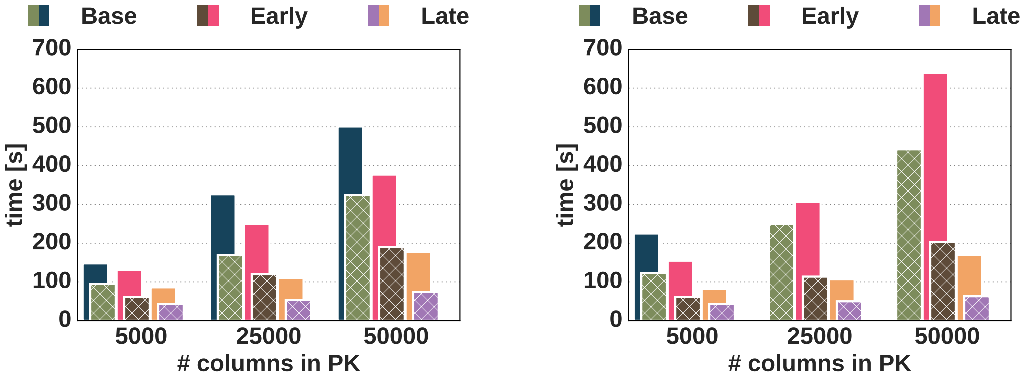
Setup. We used a local cluster with up to 20 worker nodes connected with 1 GBit Ethernet connections. Each machine is equipped with a quad-core Intel Xeon X3450 2.67 GHz, and 16GB of RAM. We implemented BlockJoin on Spark 1.6.2 (each Spark worker has 4 task slots, called *cores*) and store the initial data in HDFS 2.4.1. Every experiment is executed seven times and we report the median execution time. For the experiments on dense data, we use 20 worker nodes, resulting in a DOP of 80, while we use 10 worker nodes (DOP = 40) for sparse data.

Dataset. In order to have full control of the shape, size and content of the input tables we evaluate BlockJoin on synthetic datasets. The simulated tables, called PK and FK, have following schema: PK ($\texttt{key}, \texttt{r}_1, \dots, \texttt{r}_n$) and FK ($\texttt{fKey}, \texttt{s}_1, \dots, \texttt{s}_m$). We use a vectorization function that converts $\mathrm{r}_1, \dots, \mathrm{r}_n$ to an n-dimensional double-precision vector, and analogously $\mathrm{s}_1, \dots, \mathrm{s}_m$ to an m-dimensional double-precision vector. We conducted the experiments for dense and sparse (10% non zero values) vectors and vary the number of rows and columns. If not stated otherwise in the experiments, the tables have a 1:N primary key - foreign key relation. We use squared blocks of 1000×1000 as it was shown to make a good trade-off between computational efficiency and network traffic [GKP+11]. The corresponding sizes of the tables are given in Table 5.2.
In addition, we provide experiments on the publicly available *Reddit Comments*[4] dataset. It consists of line separated JSON entries that represent comments on the news aggregator website *www.reddit.com*. Each JSON entry contains a single comment with additional information such as the author, votes, category, etc. We split the raw data into a comment and author CSV file, by introducing a primary - foreign key relation *author_id* and use these as input to our experiments. The final join input is ~30 million comments (5.1 GB) and ~1.5 million authors (29.9 MB).

Data Distribution. Many real-world datasets exhibit extreme skew in the distribution of data points per object observed (e.g., reviews per product), and it has been shown that this skew increases over time in many datasets [LKF05]. When joining with such datasets, a small number of tuples from the skewed relation will produce a very large amount of tuples in the join result. For this reason, we conduct experiments with uniform as well as power-law distributed foreign keys (with $\alpha = 0.01$).

---

[4] http://files.pushshift.io/reddit/comments/



(a) Uniform

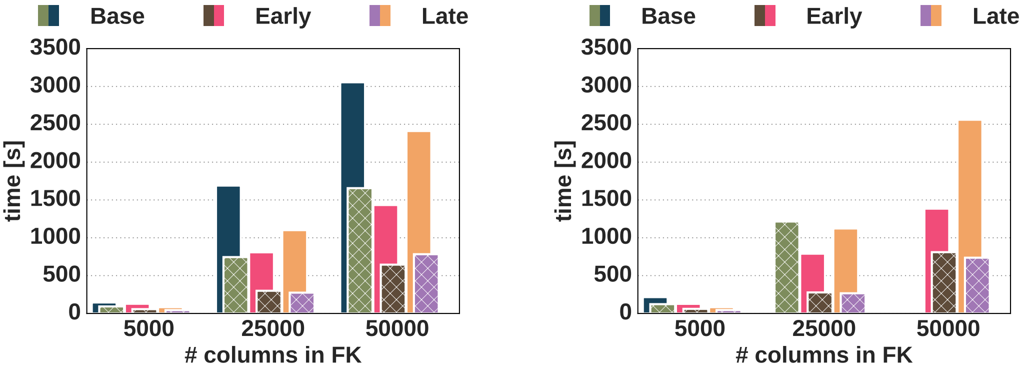
(b) Power-law

Figure 5.6: Scaling the number of columns in the PK table. The FK columns are fixed to 5K.

5.4.1 Effect of Table Shape and Size

In this experiment, we evaluate the scalability of BlockJoin for different numbers of columns. We fix the rows to 100K in the PK and 1M in the FK table. All rows in the FK table match at least one key in the PK table. Therefore, we concentrate on the effects of the block materialization strategies, as BlockJoin can not gain performance by pruning non-matching tuples (an expected effect of the *fetch-kernel* phase). Throughout the experiments, sparse data is indicated with *patched* bars in the front, whereas dense data is indicated with *solid* bars.

Scaling PK Columns. In this experiment, we fix the number of columns in the FK table to 5K, while we scale the PK table until it reaches the same data size as the FK table.
Figure 5.6a depicts the results for uniform distributed foreign keys. A first observation is that Late Materialization scales much better and is up 2.5× faster than the baseline for sparse and dense data. Late Materialization materializes duplicates (primary keys matching multiple foreign keys) at the receiver side. Thus, it only needs to shuffle data equal to the size of the input tables. In contrast, both Early Materialization and the baseline approach, materialize the duplicates (the baseline approach in the join and Early Materialization before merging partial matrix blocks). Therefore, they shuffle up to 847GB + 84,7GB (for 50K dense columns); roughly 10× more data compared to Late Materialization. Even though the baseline and Early Materialization shuffle the same amount of data, Early Materialization appears to outperform the baseline by 10%. The faster execution of Early Materialization is due to *(i)* the independent blocking of the two relations without materializing the join result, and *(ii)* our custom partitioner (cf. Section 5.3), which never splits rows sharing the same `blk-row-idx` across different partitions.
Figure 5.6b shows the same experiment for power-law distributed foreign keys. Note that the baseline approach fails to perform the join for more than 5K columns of dense data. We experienced an internal Spark error, while it tried to read partitions to execute the join on the receiver side. This is due to the heavily skewed data, which results in almost all of the work ending up in one worker node, which is unable to gather and sort the received partitions. For Late Materialization, we can observe that the algorithm is not affected by data skew and outperforms



(a) Uniform **(b)** Power-law

Figure 5.7: Scaling the number of columns in the FK table. The PK columns are fixed to 5K.

the baseline by up to 4× for sparse data. The effect of skewed keys on Early Materialization is not as severe as for the baseline, but the heavily increased amount of duplicates still decreases its performance as the PK table holds the majority of the data.

Scaling FK Columns. Figure 5.7a depicts the inverse experiment with 5K in the PK table and scaling the number of columns in the FK table. This time, Early Materialization outperforms the Late Materialization for dense data and performs up to 2× better than the baseline. Note that in this experiment, (*i*) the FK table grows very large, up to 846.7GB for dense data, in comparison to the previous experiment, while (*ii*) the resulting matrix sizes are exactly the same. Thus, as the PK table accounts for the duplicates, Late Materialization does not save much by late duplicate materialization. However, the number of columns in the FK table increases, which heavily increases the number of tuples shuffled by Late Materialization. Late Materialization emits up to 50M (1M rows split into 50K columns divided by 1K block size) row-splits, while only 1M rows are exchanged by Early Materialization and the baseline. This basically *serializes* the creation of all blocks that share the same block-row-index for Late Materialization: the `groupReduce` function that merges the row-splits of a particular block is called 50K times – for each individual block. In contrast, Early Materialization uses a `mapPartitions` function to simultaneously create blocks for all columns in a row (cf. Section 5.3) and is called only once per partition.

Figure 5.7b shows the experiment with power-law distributed foreign keys. For the two versions of BlockJoin, we can observe almost the same runtime as for the uniform distributed keys, as the data size is dominated by the FK table. Therefore, the impact of the skewed keys on Early Materialization is minor and Late Materialization does not save much data exchange. This time, the baseline approach fails to finish the experiment in case of more than 25K sparse columns due to the increased size of the FK table.

Experiment Conclusion. When the PK table size dominates the data exchange, Late Materialization performs up to 4× better than the baseline and outperforms Early Materialization. However, when the FK table dominates data exchange and the duplication of row-splits is no

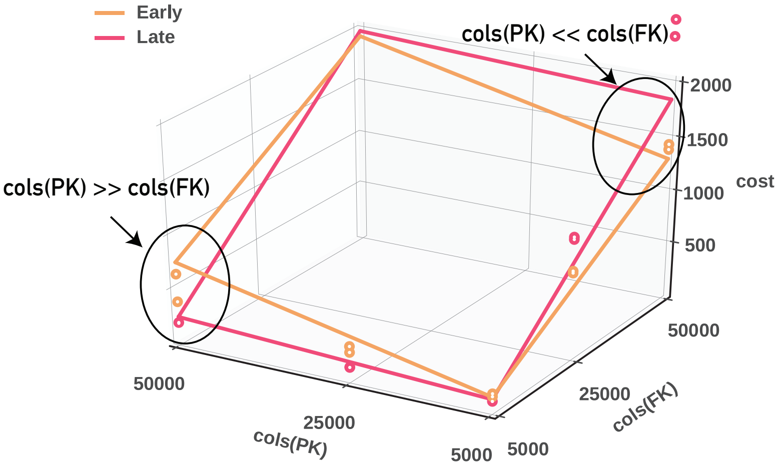
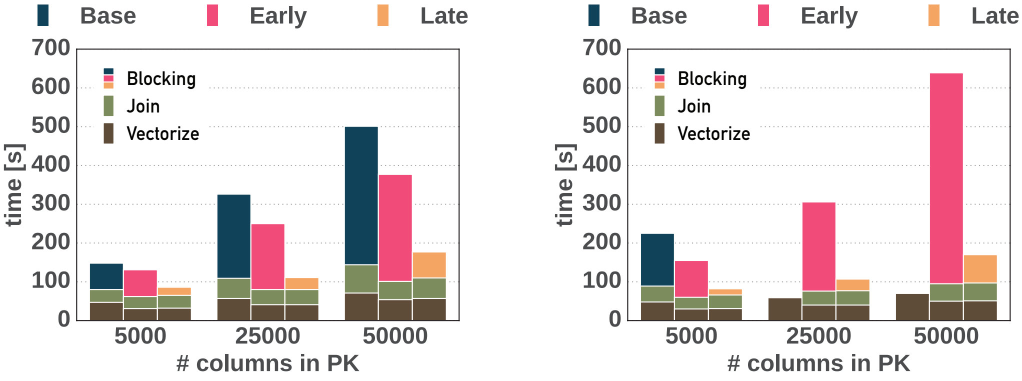


Figure 5.8: Estimated cost of the regression models, trained on the experiment results from Section 5.4.1. The number of rows correspond to the experiments (100K for PK and 1M for FK). The data points represent the experiment results for Late Materialization and Early Materialization on dense data with 5k, 25k, and 50k columns.

longer an issue, Early Materialization can be up to 1.8× faster than Late Materialization and 2× faster than the baseline. Finally, we were unable to conduct all experiments for the baseline in case of skewed data and the performance of Late Materialization is generally less affected by the data distribution.

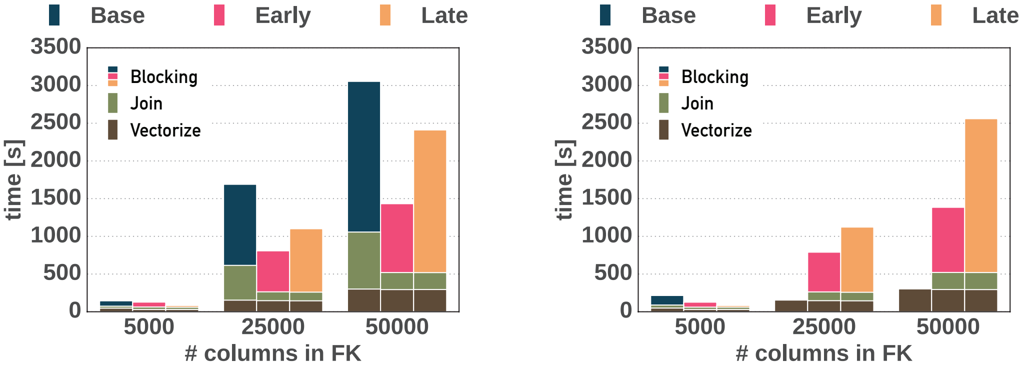
Cost Model Evaluation. We trained the regression models, described in Section 5.2.3, based on the experiment results from the previous section using dense input data. Figure 5.8 depicts the estimated runtime in relation to the number of columns in the two input relations. The number of rows is thereby the same as in the experiments (100K for PK and 1M for FK). We can observe that the model reflects the measured runtimes. While the model can serve as a binary classifier to select the best-suited strategy for other experiments, we are aware that we need more data to fit the model thoroughly. Another interesting observation is that we can use the column distribution as a simplified measure to select the strategies ($cols(PK) > cols(FK)$ favors Late Materialization and vise versa). This ratio turns out to be a pretty good estimation model and can be used as a fall-back in an optimizer, as long as not enough training data is available to fit the model.

Detailed runtimes of the different phases. In Figure 5.9 and Figure 5.10, we show the runtime of each of the phases – vectorize, join, and blocking – for the experiments with dense data in Figure 5.6 and Figure 5.7 respectively. Due to pipelining in Spark, we had to measure the phases in separated jobs to obtain their individual runtime. Thus, the results are indicating the runtimes of the different phases, but do not reflect the exact time spent.



(a) Uniform **(b)** Power-law

FIGURE 5.9: Split up execution times for scaling the number of columns in the PK table.



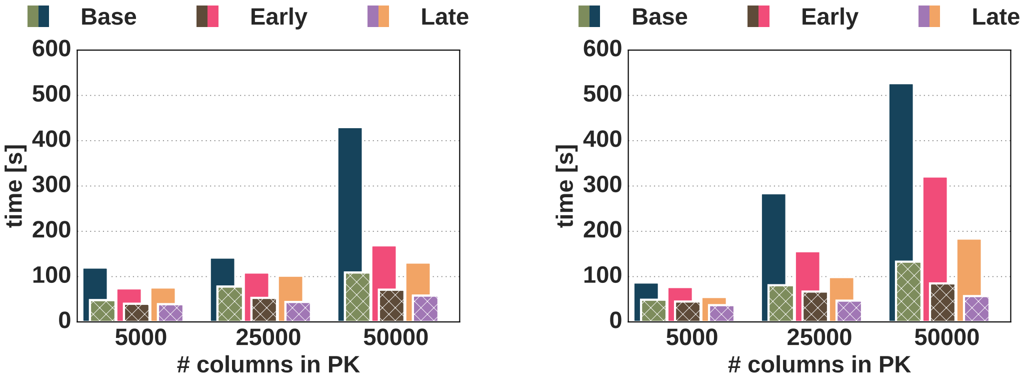
(a) Uniform **(b)** Power-law

FIGURE 5.10: Split up execution times for scaling the number of columns in the FK table.

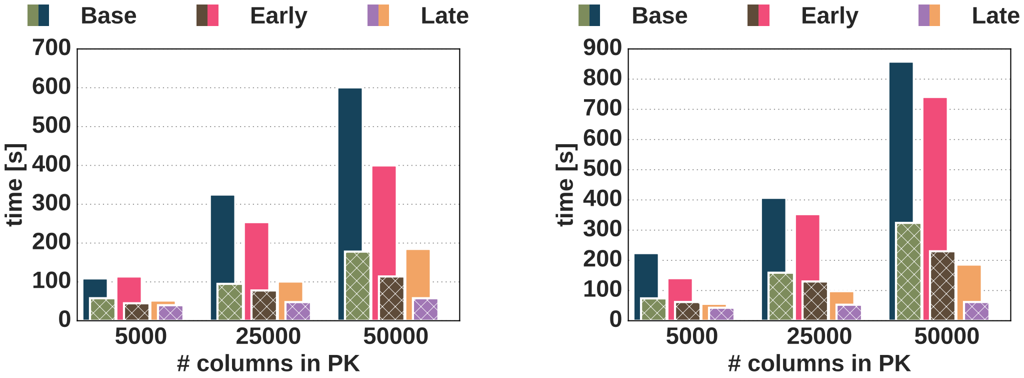
Vectorize – We observe roughly equal run times, which is expected, as the same vectorization function is performed for both the baseline and BlockJoin.

Join – We observe different behavior depending on whether we scale the PK or FK columns. Scaling the PK columns (cf. Figure 5.9), we see only a minor speedup for BlockJoin in case of uniform distributed keys. For power-law distributed keys, the baseline fails to execute the join after 5K columns. As expected, BlockJoin is not sensitive to skewed keys and the join times are equal to the cases with uniformly distributed keys. Scaling the FK columns (cf. Figure 5.10), we observe a speedup of up to 3x. Compared to Figure 5.9, we have to shuffle much more data, as we increase the FK columns. BlockJoin degrades gracefully with an increasing number of columns, as we have to read the data to project the join keys. Again, the baseline fails to execute the join for power-law distributed keys, while BlockJoin is not affected by skew.

Blocking – We observe performance gains of up to 3x for the best-suited materialization strategy. This applies mainly for late materialization, as the benefits are rather small in cases early materialization is better. The gains in performance for early materialization are due to the



(a) Result rows: 100K, 1:1 relation **(b)** Result rows: 200K, 1:2 relation



(c) Result rows: 400K, 1:4 relation **(d)** Result rows: 1M, 1:10 relation

FIGURE 5.11: Effect of scaling the number of columns in PK table: **(a)** has a 1:1 relation. **(b)** – **(d)** depict M:N relations with 2, 4, and 10 duplicates per key.

block-size aware partitioning. Late materialization gains performance due to the receiver-side materialization of duplicates. Thus, we observe a huge performance gain when scaling the PK columns. The behavior reflects the assumptions of our cost model: When scaling the PK columns, Late Materialization is superior as it avoids the materialization of the duplicates in the PK table and thus, shuffles considerably less data. When we scale the FK columns, Late Materialization can not gain much from receiver-side materialization as the majority of data resides in the FK table, but has to shuffle way more tuples. The experiments show that BlockJoin gains performance with both, an efficient, skew resistant join and the right choice of the materialization strategy.

5.4.2 1:1 and M:N Relations

In this experiment, we analyze the effects of 1:1 and M:N relations between the keys in the two relations. Therefore, we fix the number of rows in both tables to 100K and use sequential keys in both relations, but vary the range we draw the keys from. Figure 5.11a depicts a 1:1 relation; each key appears once per table. Late Materialization and Early Materialization gain up to 2× speedup compared to the baseline (both for sparse and dense data). As there are no duplicates, Early Materialization is only slightly slower than Late Materialization. Figure 5.11b – Figure 5.11d illustrate M:N relations with 2, 4, and 10 duplicates per key, and therefore, 200K, 400K, and 1M rows in the matrix. Throughout the experiments, sparse data is indicated with *patched* bars in the front, whereas dense data is indicated with *solid* bars. While the baseline has the worst performance throughout the series, we can observe a declining performance of Early Materialization with an increasing number of duplicates for dense data. The runtime of Late Materialization is almost not affected by the number of duplicates and gains up to 4× speedup compared to the baseline for dense and sparse data.

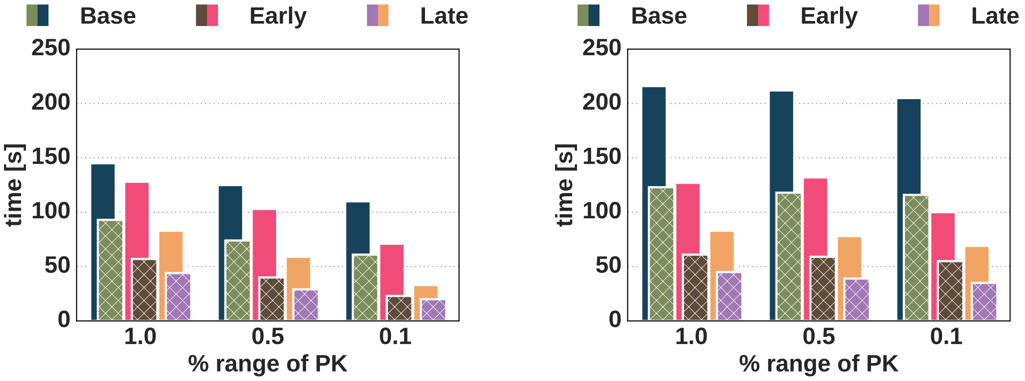
5.4.3 Effect of Selectivity

In this experiment, we investigate the performance implications of the join selectivity. Therefore, we can observe the impact of the semi-join reduction in the fetch-kernel. We start with the same number of rows in the PK and FK table as in the previous experiment (cf. Section 5.4.1), but we restrict the number of tuples in PK table. As a result, not all foreign keys match. This reflects a common use case, where only certain values, e.g., products of a given category, are of interest. Sparse data is indicated with *patched* bars in the front, whereas dense data is indicated with *solid* bars.

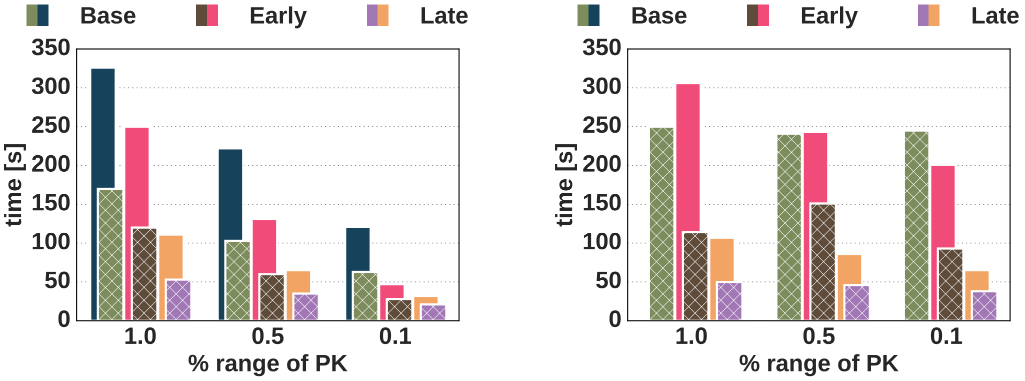
Scaling PK Columns. Figure 5.12 shows the experiment with fixed FK columns (5K) and scaling PK columns. On the x-axis, we increase the selectivity of the filter on the PK table. The selectivity not only defines the number of rows in the PK table (from 100K to 10K rows), but also the number of matching foreign keys, and thereby the size of the join result/matrix. Again, Late Materialization outperforms Early Materialization, but the benefits of late duplicate materialization decrease with increasing selectivity. Nevertheless, we achieve up to 4× speedups, due to pruning non-matching tuples in the fetch-kernel. For power-law distributed keys (cf. Figure 5.12b), the baseline approach fails for PK tables with more than 5K columns of dense data and the skew resistant Late Materialization gains up to 6× speedups for sparse data.

Scaling FK Columns. Figure 5.13 depicts the experiments with a scaling number of columns in the FK table. Again, we can observe the performance degradation of Late Materialization, compared to the experiments in Figure 5.12, as the number of FK columns increases. Note that increasing selectivity mitigates the performance impact of row splitting for Late Materialization due to pruning in the fetch-kernel and we see an almost equal performance for Early Materialization and Late Materialization in case of 0.1 selectivity. The semi-join reduction thereby increases the speedups from 2× for 1.0 up to 6× for 0.1 selectivity. Figure 5.13b shows the experiment with power-law distributed keys. While Late Materialization can outperform Early Materialization in the smallest configuration, pruning cannot mitigate the exploding number of tuples for a larger number of columns in the dense case.

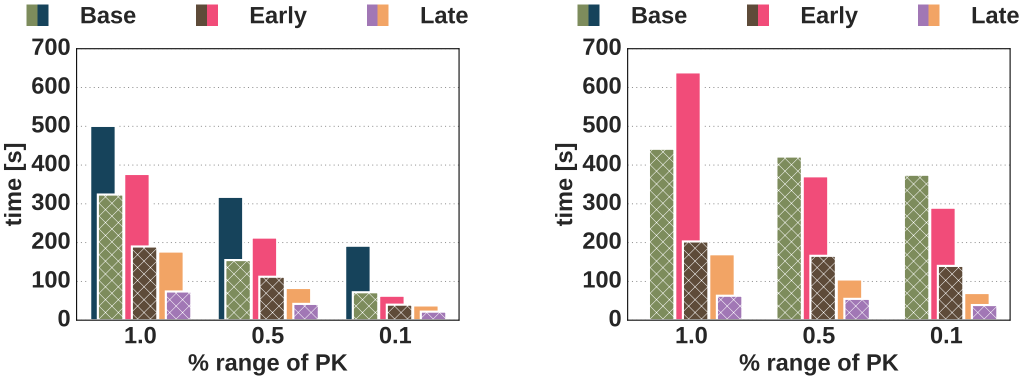
Experiment Conclusion. Filter predicates on the primary key table are a common use case, e.g., to consider only a certain category of products. The meta-data in BlockJoin enables it



5.000 PK columns



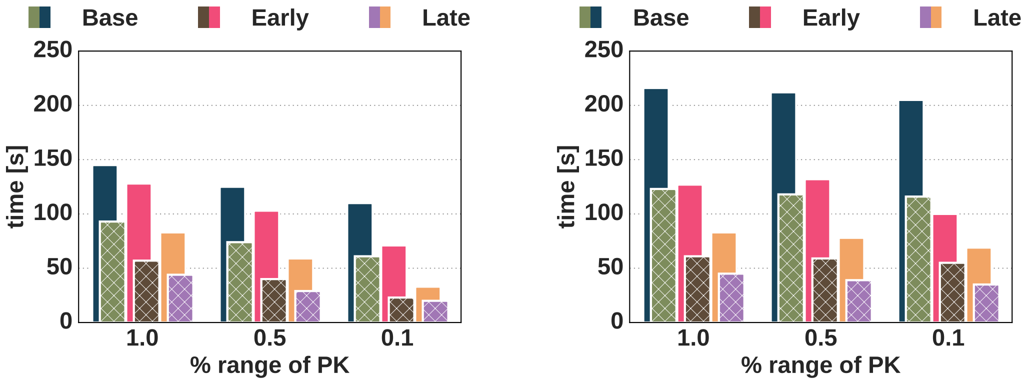
25.000 PK columns



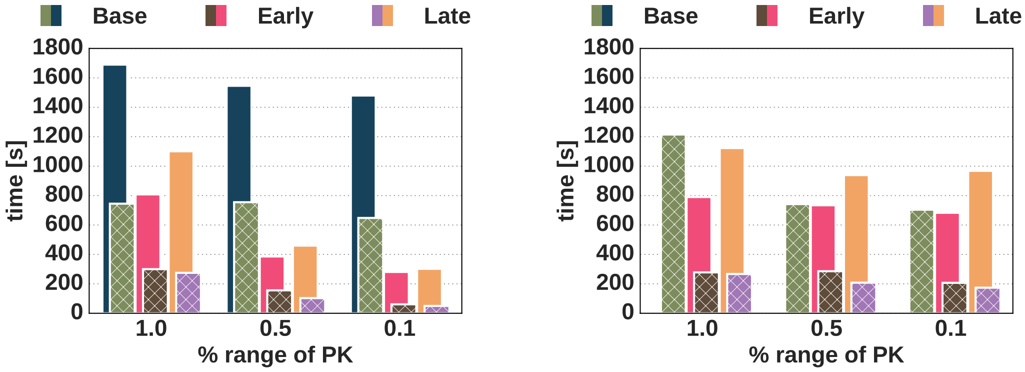
(a) Uniform **(b)** Power-law

50.000 PK columns

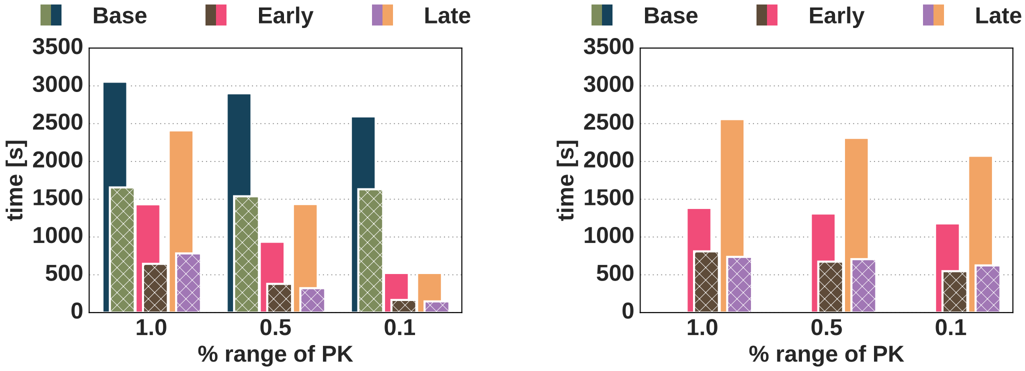
FIGURE 5.12: Effect of selectivity for varying number of columns in the PK table. The number of FK columns is fixed to 5K.



5.000 FK columns



25.000 FK columns

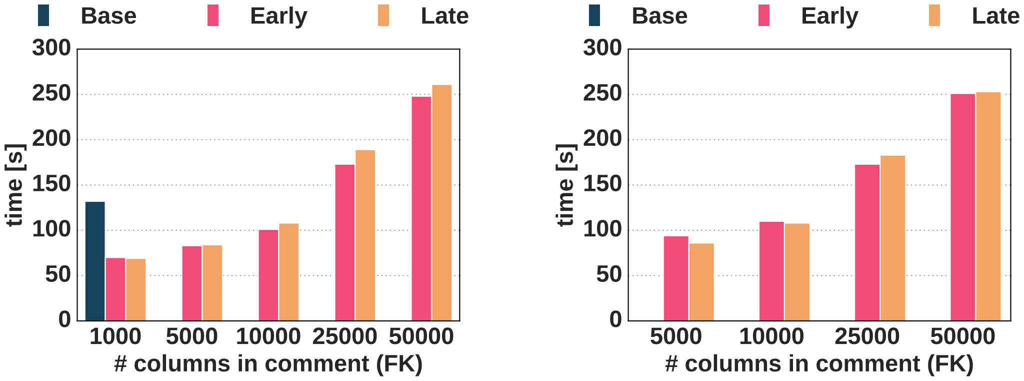


(a) Uniform

(b) Power-law

50.000 FK columns

Figure 5.13: Effect of selectivity for varying number of columns in the FK table. The number of PK columns is fixed to 5K.



(a) User vector dim. = 1000

(b) User vector dim. = 5000

Figure 5.14: Effect of scaling the number of columns for the comment relation.

to prune non-matching FK tuples in the fetch kernel, whereas the baseline implementation has to shuffle all FK tuples to determine the matching tuples. Thus, the experiments show the benefits of the semi-join reduction BlockJoin can perform in the fetch kernel as it does not have to materialize the join result. This semi-join reduction improves the performance benefits of BlockJoin compared to the baseline by up to 6×.

5.4.4 Reddit Comments Dataset

In this experiment, we evaluate our BlockJoin on the Reddit Comments dataset. In order to obtain the full feature set, we join the comments and authors CSV input files. To create a vector representation, we apply *feature hashing* to the author's name and the comments text. We split the name by camel case, white space, and other delimiters and hash the words to a fixed size feature space. For the comments, we split the text into words and hash them as described before. Figure 5.14 depicts the results of the experiment. We fix the dimensions of the author name feature vector to 1000 and 5000 and increase the dimensions of the comments vector. The first observation is that the baseline implementation fails after the first scaling factor. This is due to an out of memory exception in the blocking phase. The large number of comments (~30 million tuples) exceeds the available memory in the `mapPartitions` operators that create partial blocks within each partition. This limitation in the baseline reflects a general problem in dataflow engines introduced by the allocation of objects in UDFs: if the memory consumption in an UDF exceeds the available memory in the Java Heap the processing fails. This is in contrast to data managed by the dataflow engine, e.g., to join two relations, which is gradually spilled to disk to avoid out-of-memory errors. While we also create partial blocks in the Early Materialization approach, we execute the blocking on the two relations separately, without prior joining. This leads to less memory pressure, compared to the baseline. Late materialization is not affected by memory pressure. This leads, in combination with the huge difference in the size of the relations (1 : 30) and the relatively small sparse feature vectors, to an almost equal runtime for Late Materialization and Early Materialization.

5.5 Related Work

Join Optimization. Optimized join algorithms have been well studied in the area of distributed database systems [Mul90, RK91, SY93, RIKN16, AKN12] and parallel dataflow systems [PSR14, WLMO11, AU10, OR11, ZLC10] like Hadoop [Apa] and Spark [ZCF⁺10], with the aim of reducing network traffic and dealing with skewed data. Efficient join implementations in main-memory databases are based on TID-joins [MR94, DKO+84] and late materialization [THS+09, LR99] to achieve cache efficiency and to delay the materialization of the data to the latest possible time. In BlockJoin, we apply and enhance these techniques for the domain of distributed matrix computation by using *index-joins* to create the matching tuples without re-partitioning the tables. More specifically, we apply a *semi-join* reduction to prune tuples before creating the blocks and we introduce *late materialization* to avoid sending rows resulting from duplicated join keys.

Array Databases. RasDaMan [BDF+98] is an array DBMS for multidimensional discrete data with an extended SQL query language. It stores its data as *tiles*, i.e., possibly non-aligned subarrays, as blobs in an external DBMS. While their optimizer provides a rich set of heuristic-based rewrites, to the best of our knowledge, RasDaMan does not perform joint optimization over relational and array backed data. SciDB [Bro10] is another array database that, in contrast to RasDaMan, provides its own shared-nothing storage layer. This allows SciDB to store and query tiles more efficiently. It provides a variety of optimizations, like overlapping chunks and compression. We see BlockJoin as complementary to the research in array databases and its ideas could be implemented to enhance their data loading and transformation.

Algebra Unifying Approaches. Kumar et al. [KNP15] introduce learning generalized linear models over data residing in a relational database. The authors push parts of the computation of the ML model into joins over normalized data, similar to [CS94]. These works target generalized linear models only, while our approach subsumes a more generic optimization that can be used in arbitrary machine learning pipelines over normalized data. MLBase [KTD+13] 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 the relational operators into account nor does it optimize the physical representation of the data among different operators. Cohen et al. [CDD+09] execute linear algebra operations in a relational database but do not present optimizations for block-partitioning the operands. Kumar et. al [KNPZ16] discuss whether it is beneficial to execute the join at all. They argue that the foreign key determines the additional features in the foreign key table making them *irrelevant* in certain cases. The authors propose several rules to determine the right feature set: the whole join result, the join result without the foreign key, and the primary key table only. In BlockJoin, the user defines the feature set in the user-defined vectorization function and thus, can remove the foreign key.

ML Libraries & Languages. SystemML's DML [BBE+14, SSM+15, BDE+16, EBH+16], Mahout's Samsara [SPQ+16], provide R-like linear algebra abstractions. SystemML executes locally or distributed on Hadoop and Spark, while Samsara targets Spark, Flink and $H_2 0$. As there is no dedicated support for relational operators, ETL has to be executed using a different set of abstractions, and both systems lose the potential for holistic optimization. MLlib [MBY+16, ZMU+16], MLI [STS+13], Cumulon [HBY13] and Pegasus [KTF09] employ different strategies to efficiently execute matrix operations on distributed dataflow systems, but again do not target holistic optimization over relational and linear algebra operators. We presented recently the potential for optimizations across relational and linear algebra in the context of the Lara [KAKM16] language, based on Emma [AKK+15].

5.6 Conclusion

In this chapter, we introduce a scalable join algorithm for analytics that mix relational and linear algebra operations. Our technique reduces the re-partitioning overheads, which stem from the different physical representations of relations and matrices. To this end, we propose BlockJoin, an optimized join algorithm, which fuses relational joins with blocked matrix partitioning, avoiding costly re-partitioning steps. We discuss different block materialization strategies of this join operator and their cost-model driven application, depending on the shape of the input data. In an extensive experimental evaluation, we show that BlockJoin outperforms the current state of the art implementation for dataflow systems up to a factor of six, and demonstrated that BlockJoin is scalable and robust on highly skewed data.

Future work. BlockJoin and other physical operators can be integrated in a common intermediate representation, e.g., as presented in Chapter 4 and [AKK+15, KAKM16, SHG+15]. Moreover, BlockJoin can be extended to generate a variety of block-partitioned matrices for model selection workloads that are commonly employed to find well-working features and hyperparameters for machine learning models [SSM+15].

6

CONCLUSION

In this section, we revisit our contributions before we discuss them with respect to the problems stated in Section 1.1. To conclude, we reflect on the possibilities for research transfer and give pointers regarding future work in the broad field of optimizations over end-to-end ML pipelines.

In Chapter 3, we presented ScootR, a DSL that can be used as drop-in replacement for the `dataframe` type in R. ScootR offloads the execution of R programs to the distributed dataflow system Apache Flink. Users can transparently scale their existing R `dataframe` code on large amounts of data. The tight integration of Flink and the R language in a shared runtime provided by the Truffle framework and Graal compiler enables efficient execution of complex UDFs. Compliance with GNU-R provides a rich set of native language features. ScootR achieves performance comparable to systems based on source-to-source translation, even though it executes UDFs in an R language runtime. Compared to inter-process communication-based systems, ScootR achieves up to an order of magnitude higher performance.

In Chapter 4, we presented Lara, a quotation-based DSL for ML model training pipelines. It provides dedicated types for collections and matrices that can be interleaved with calls to UDFs. Its IR gives access to the whole AST of the pipeline, including UDFs and native control flow. Two additional views on top of the low-level IR increase the abstraction level to enable diverse optimizations: a monadic representation for both collections and matrices allows to pushdown and fuse UDFs over type boundaries. The combinator view represents high-level operators, e.g., joins or matrix multiplication, as single entities in an operator tree. Similar to database optimizers, this enables the generation of logical and physical plan variants.

In Chapter 5, we presented BlockJoin, a logically fused operator chain at the intersection of row-partitioned preprocessing and block-partitioned matrices for ML. It improves on common *join-vectorize-training* patterns in end-to-end pipelines and directly creates block-partitioned results from normalized data. BlockJoin is inspired by join techniques that use late materialization and applies them in a distributed setting. It separates the creation of the block indexes from the materialization of the block data. This avoids data shuffle and materialization of row-partitioned join results and BlockJoin is thereby not affected by skewed data. Furthermore, this separation enables a cost-based decision for the materialization strategy of the matrix blocks based on the shape of the normalized input.

6.1 Discussion

Dynamic guest languages extensions on top of dataflow engines lower the barrier to dataflow systems for non-expert programmers. ScootR acts as a drop-in replacement for existing preprocessing pipelines based on dataframes. ScootR supports complex UDFs, and we showed that their integration does not necessarily introduce a significant loss in performance. While the approach can be extended to support ML algorithms (e.g., by the `matrix` type in R), it still lacks a common IR to optimize over different types: ScootR relies on the type-based IR of Flink and inherits its limitations, e.g., black-box UDFs and no control flow. To this end, holistic optimizations rely on an IR that combines domain-specific optimizations for relational and linear algebra, but also provides access to UDFs and control flow of the host language to reason about iterative algorithms. We showed that such an IR not only allows to reason about operator fusion over type and loop boundaries but is also capable of optimizations that require domain knowledge, e.g., the semantics of linear algebra. Such an IR can also be used to detect operator patterns that one can implement as fused operators. The results that we have achieved with our fused operator for normalized data shows that a holistic view overcomes inherent limitations of isolated operators by applying join techniques in a distributed setting.

Research Transfer

We want to conclude with a perspective on the applicability of the presented contributions in real-world scenarios. During the timespan of the creation of this thesis, ML and in particular deep learning became omnipresent in almost all areas of computer science. This development not only changed the way analytics are conducted, but also attracted a broad audience with a diverse skill set. In the following, we describe how one could integrate our research contributions into a system that covers the diverse demands of users:

Data Scientists. Dynamic languages, such as R and Python, are the *de facto* standard for data exploration. Open-source libraries provide a rich set of predefined feature transformations and machine learning algorithms that lowered the barrier for non-expert programmers significantly. Data scientists apply cascades of library functions for data transformation and model training. To this end, an API with library support for popular preprocessing and ML algorithms in a dynamic programming language is a crucial requirement. For efficiency, the system must offload the actual computation to an underlying execution engine written in a systems language. Thus, a guest language integration, as described in Chapter 3 is required. We see an ongoing trend in systems design towards code generation to complement JVM-bases systems with native kernels for performance-critical aspects [BRH$^+$18, ETD$^+$18] and languages that generate machine code, e.g., C++ and Rust [ABC$^+$16, GSB$^+$18, MNW$^+$18]. This is due to shortcomings of the JVM, e.g., managed memory and limited support for modern hardware. Even though ScootR is based on the JVM, one can integrate the described techniques into a low-level IR for hardware targeted code, e.g., LLVM [LA04].

Domain Experts. ML researchers provide their newest results as *plug-and-play* implementations in the high-level APIs of Tensorflow and PyTorch [VBB+18]. Data engineers write preprocessing pipelines as transformations on `dataframes`, which are adopted by most libraries [McK12] and systems [ZCF+10, CKE+15]. This separation prevents inter-library and domain optimizations.
A quotation-based DSL based on a common IR representation would provide several advantages: (*i*) the DSL can reuse native control flow and UDFs are reflected in the IR; (*ii*) a common IR provides intra and inter-domain optimizations as discussed in Chapter 4; (*iii*) it provides imperative *and* optimized, symbolic execution. This provides fast development cycles and efficient debugging; (*iv*) compared to a standalone DSL, quotation-based DSLs can reuse the infrastructure of the host language, e.g., IDEs and tooling. Even though we implemented our prototype in Scala, hardware tailored languages, such as Rust, provide similar meta-programming facilities.

System Experts. ML algorithms frequently combine certain linear algebra and tensor operations. System experts hard-code such operator chains as optimized kernels to provide fast execution [LG16]. Hard-coded kernels have two major disadvantages: (*i*) system experts that also have expertise in ML are rare, but are required to implement such kernels; (*ii*) as these operator chains are hard-coded, there is no opportunity for inter-kernel optimizations, and experts have to implement each new operator chain from scratch.
We cannot solve the first problem, but a common IR can perform physical operator selection based on the executed pipeline. Thus, system experts can provide fine-grained physical operator implementations, which can be integrated and combined during the plan variant generation.

6.2 Future Work

In addition to the future work mentioned in the chapters of the individual contributions, we will now discuss some more general directions for future work:

Prediction. We restrict ourselves to pipelines for model training in this thesis. Similar ideas to the ones presented can be applied for the prediction. Predictions pose a new set of requirements and therefore, opportunities for holistic optimizations. The preprocessing phase of the model training has to be applied to a new data item before it is used for prediction. In contrast to the training phase, the preprocessing steps and model access have to be optimized for low latency – in contrast to the high throughput required in model training. Furthermore, prediction pipelines are likely to apply relational operators on the results retrieved from the model, e.g., to obtain the top-k recommended items for a particular user. This presents even further potential for inter-domain optimizations.

Hyperparameter Tuning. We did not fully investigate the possible optimizations for hyperparameter tuning. In automated settings, users only have to specify certain ranges for the hyperparameters of an ML algorithm. The tuning algorithms find the best-suited parameter combinations. Current solutions employ heuristics and pruning techniques to reduce the massive search space [BB12, STH+15, KMNP15], or use task parallelism to search for different configurations in parallel [BTR+14]. A suitable IR can augment these optimizations to detect loop invariant code, potential for shared intermediate data, and algorithmic optimizations similar to our proposed optimizations for CV (cf. Section 4.2.4).

Code Generation for Modern Hardware. Modern hardware and in particular GPUs are used to accelerate linear algebra and tensor operations. Operators are mostly implemented as isolated kernels [GJ+10, LG16]. Query compilation techniques for main-memory databases on CPUs [Neu11, HM12] have recently been extended for heterogeneous processors [BKF+18]. We think that these techniques can be extended to cover end-to-end pipelines in a similar way as we presented in Chapter 4.

APPENDIX

A: ScootR

```
df <- flink.readdf(...)
df <- flink.filter(df, df$cancelled == 0 &&
                       df$dep_delay >= 10 && df$carrier %in% c("AA", "HA"))

df <- flink.select(df, carrier, origin, dest, dep_delay, arr_delay)
df$avgDelay <- (df$arr_delay + df$dep_delay) / 2
df$delay <-
  if (df$avgDelay > 30) "High"
  else if (df$avgDelay < 20) "Low"
  else "Medium"

df$head(5)
```

LISTING 1: Data transformation pipeline proposed in [YZP14].

```
df  <- flink.readdf(...)
df  <- flink.filter(df, df$origin == 'JFK')
grp <- flink.groupBy(df, 'dest')
max <- grp$max('arr_delay')
cat(max$head(5))
```

LISTING 2: The maximal arrival delay per destination for flights starting from New York.

```
df <- flink.readdf(...)
df <- flink.filter(df, df$cancelled == 0 &&
  df$dep_delay >= 10 && df$carrier %in% c("AA", "HA"))

df <- flink.select(df,
  carrier, dep_delay, arr_delay, distance)

fastR_df <- flink.collect(df)
model <- glm(
  arr_delay ~ dep_delay + distance,
  data = fastR_df,
  family = "gaussian")

summary(model)
```

Listing 3: A end-to-end pipeline with distributed ETL on Flink and local model training.

```
df <- flink.readdf(...)
ngrams <- function(tpl, collector) {
    splits    <- strsplit(tpl$body, " ")[[1]]
    numSplits <- length(splits)

    srtIdx <- 1
    endIdx <- 2
    while (endIdx <= numSplits) {
        twoGram <- paste(splits[srtIdx:endIdx],
                         collapse = " ")
        srtIdx <- srtIdx + 1
        endIdx <- endIdx + 1
        collector$collect(flink.tuple(twoGram, 1))
    }
}

df <- flink.apply(df, ngrams)
flink.writedf(df, outFile)
flink.execute()
```

Listing 4: Calculating the 2-grams of the *body* column in the Reddit comments dataset.

B: Lara

```
trait Matrix extends Serializable {
  val numRows: Int
  val numCols: Int
  val transposed: Boolean

  // element-wise M o scalar (+,-,*,/)
  def +(that: Double): Matrix
  // element-wise M o vector (+,-,*,/)
  def +(that: Vector): Matrix
  // element-wise M o M  (+,-,*,/)
  def +(that: Matrix): Matrix

  // M x M -> M,  M x V -> V, solve
  def **(that: Matrix): Matrix
  def **(that: Vector): Vector
  def  \(that: Vector): Vector

  def diag(): Vector
  def t: Matrix

  def row(rowIndex: Int): Vector
  def col(colIndex: Int): Vector

  // higher-order methods on dimensions (same for columns)
  def forRows(f: Vector => Double): Vector
  def forRows(f: Vector => Vector): Matrix
  def forRows(f: Vector => Boolean): Matrix

  def forRows(f: Idx[Int, Vector] => Double): Vector
  def forRows(f: Idx[Int, Vector] => Vector): Matrix
  def forRows(f: Idx[Int, Vector] => Boolean): Matrix

  // point-wise higher-order methods
  def map(f: Double => Double): Matrix
  def map(f: Idx[(Int, Int), Double] => Double): Matrix

  def reduce(f: (Double, Double) => Double): Double
  def fold[B](z: B)(s: Double => B, p: (B, B) => B): B
  def fold[B](z: B)(s: Idx[(Int, Int), Double] => B, p: (B, B) => B): B
}
```

Listing 5: Important methods of Lara’s matrix type.

```
trait Vector extends Serializable {
  val size: Int

  // element-wise vector o scalar (+,-,*,/)
  def +(that: Double): Vector
  // element-wise vector o vector (+,-,*,/)
  def +(that: Vector): Vector
  // inner product
  def dot(that: Vector): Double
  // outer product
  def **(that: Vector): Matrix
  // row vector x matrix
  def **(that: Matrix): Vector

  def diag(): Matrix
  def t: Vector

  // higher-order methods
  def map(f: Double => Double): Vector

  def reduce(f: (Double, Double) => Double): Double

  def fold[B](z: B)(s: Double => B, p: (B, B) => B): B
  def fold[B](z: B)(s: Idx[Int, Double] => B, p: (B, B) => B): B
}
```

Listing 6: Important methods of Lara’s vector type.

BIBLIOGRAPHY

[ABB+12] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. Nodb: efficient query execution on raw data files. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 241–252, 2012.

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 265–283, 2016.

[ABE+14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The stratosphere platform for big data analytics. *VLDB J.*, 23(6):939–964, 2014.

[AK01] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann, 2001.

[AKK+15] Alexander Alexandrov, Andreas Kunft, Asterios Katsifodimos, Felix Schüler, Lauritz Thamsen, Odej Kao, Tobias Herb, and Volker Markl. Implicit parallelism through deep language embedding. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 47–61, 2015.

[AKM19] Alexander Alexandrov, Georgi Krastev, and Volker Markl. Representations and optimizations for embedded parallel dataflow languages. *ACM Trans. Database Syst.*, 44(1):4:1–4:44, 2019.

[AKN12] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10):1064–1075, 2012.

[Aly05] Mohamed Aly. Survey on multiclass classification methods. *Neural networks*, 19:1–9, 2005.

[AMH08] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 967–980, 2008.

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

[Apa18] Apache Foundation. Apache arrow. https://arrow.apache.org/, 2018. [Online; accessed 2018-8-27].

[App98] Andrew W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[ASK$^+$16] Alexander Alexandrov, Andreas Salzmann, Georgi Krastev, Asterios Katsifodimos, and Volker Markl. Emma in action: Declarative dataflows for scalable data analysis. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2073–2076, 2016.

[AU10] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, pages 99–110, 2010.

[AXL$^+$15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1383–1394, 2015.

[B$^+$08] Dhruba Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 53:1–13, 2008.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, 2012.

[BBC$^+$17] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. TFX: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*, pages 1387–1395, 2017.

[BBE$^+$14] Matthias Böhm, Douglas R. Burdick, Alexandre V. Evfimievski, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Shirish Tatikonda, and Yuanyuan Tian. Systemml's optimizer: Plan generation for large-scale machine learning programs. *IEEE Data Eng. Bull.*, 37(3):52–62, 2014.

[BBR96] Ronald F Boisvert, Ronald F Boisvert, and Karin A Remington. *The matrix market exchange formats: Initial design.* US Department of Commerce, National Institute of Standards and Technology, 1996.

[BDE$^+$16] Matthias Boehm, Michael Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, Prithviraj Sen, Arvind Surve, and Shirish Tatikonda. Systemml: Declarative machine learning on spark. *PVLDB*, 9(13):1425–1436, 2016.

[BDF$^+$98] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. The multidimensional database system rasdaman. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.*, pages 575–577, 1998.

[BdM96] Richard S. Bird and Oege de Moor. The algebra of programming. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*, pages 167–203, 1996.

[BEG$^+$11] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl-Christian Kanne, Fatma Özcan, and Eugene J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283, 2011.

[BFG$^+$17] Joos-Hendrik Boese, Valentin Flunkert, Jan Gasthaus, Tim Januschowski, Dustin Lange, David Salinas, Sebastian Schelter, Matthias W. Seeger, and Bernie Wang. Probabilistic demand forecasting at scale. *PVLDB*, 10(12):1694–1705, 2017.

[BKF$^+$18] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Generating custom code for efficient query execution on heterogeneous processors. *VLDB J.*, 27(6):797–822, 2018.

[BMK99] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 54–65, 1999.

[Bre05] Eric A Brewer. Combining systems and databases: A search engine retrospective. *Readings in Database Systems*, 4:247–261, 2005.

[BRH$^+$18] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. On optimizing operator fusion plans for large-scale machine learning in systemml. *PVLDB*, 11(12):1755–1768, 2018.

[Bro10] Paul G. Brown. Overview of scidb: large scale array storage, processing and analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 963–968, 2010.

[BTR$^+$14] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas Burdick, and Shivakumar Vaithyanathan. Hybrid parallelization strategies for large-scale machine learning in systemml. *PVLDB*, 7(7):553–564, 2014.

[Bur13] Eugene 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, SCALA@ECOOP 2013, Montpellier, France, July 2, 2013*, pages 3:1–3:10, 2013.

[BYM$^+$14] Peter Baumann, Jinsongdi Yu, Dimitar Misev, Kinga Lipskoch, Alan Beccati, P Campalani, and M Owonibi. Preparing array analytics for the data tsunami. In *Geographical Information Systems*, pages 11–29. CRC Press, 2014.

[CCS12] Hsinchun Chen, Roger H. L. Chiang, and Veda C. Storey. Business intelligence and analytics: From big data to big impact. *MIS Quarterly*, 36(4):1165–1188, 2012.

[CDD$^+$09] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. MAD skills: New analysis practices for big data. *PVLDB*, 2(2):1481–1492, 2009.

[CDPW92] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *FMPC*, pages 120–127. IEEE, 1992.

[CGD$^+$15] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Çetintemel, and Stan Zdonik. An architecture for compiling udf-centric workflows. *PVLDB*, 8(12):1466–1477, 2015.

[CKE$^+$15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[CKNP17] Lingjiao Chen, Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. Towards linear algebra over normalized data. *PVLDB*, 10(11):1214–1225, 2017.

[CL11] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM TIST*, 2(3):27:1–27:27, 2011.

[CLL$^+$15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[CLS07] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 315–326, 2007.

[CS94] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 354–366, 1994.

[CSB$^+$11] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 35–46, 2011.

[CT$^+$94] William B Cavnar, John M Trenkle, et al. N-gram-based text categorization. In *Proceedings of SDAIR-94, 3rd annual symposium on document analysis and information retrieval*, volume 161175. Citeseer, 1994.

[DDGR07] Abhinandan Das, Mayur Datar, Ashutosh Garg, and Shyamsundar Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 271–280, 2007.

[DG92] David J DeWitt and Jim Gray. Parallel database systems: The future of high performance database processing. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1992.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150, 2004.

[DGG$^+$86] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. GAMMA - A high performance dataflow database machine. In *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings.*, pages 228–237, 1986.

[DKO$^+$84] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, pages 1–8, 1984.

[DS08] David DeWitt and Michael Stonebraker. Mapreduce: A major step backwards. *The Database Column*, 1:23, 2008.

[DSB+10] Sudipto Das, Yannis Sismanis, Kevin S. Beyer, Rainer Gemulla, Peter J. Haas, and John McPherson. Ricardo: integrating R and hadoop. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 987–998, 2010.

[EBH+16] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. Compressed linear algebra for large-scale machine learning. *PVLDB*, 9(12):960–971, 2016.

[ELB+17] Tarek Elgamal, Shangyu Luo, Matthias Boehm, Alexandre V. Evfimievski, Shirish Tatikonda, Berthold Reinwald, and Prithviraj Sen. SPOOF: sum-product optimization and operator fusion for large-scale machine learning. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[ETD+18] Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, pages 799–815, 2018.

[ETKM12] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning fast iterative data flows. *PVLDB*, 5(11):1268–1279, 2012.

[Eve80] Howard Whitley Eves. *Elementary matrix theory*. Courier Corporation, 1980.

[FCW+18] Raul Castro Fernandez, William Culhane, Pijika Watcharapichat, Matthias Weidlich, Victoria Lopez Morales, and Peter R. Pietzuch. Meta-dataflows: Efficient exploratory dataflow jobs. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1157–1172, 2018.

[FM00] Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, 2000.

[FSSD17] Juan José Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. Just-in-time GPU compilation for interpreted languages with partial evaluation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*, pages 60–73, 2017.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 29–43, 2003.

[GHR+12] Saptarshi Guha, Ryan Hafen, Jeremiah Rounds, Jin Xia, Jianfu Li, Bowei Xi, and William S Cleveland. Large complex data: divide and recombine (d&r) with rhipe. *Stat*, 1(1):53–67, 2012.

[GJ$^+$10] Gaël Guennebaud, Benoit Jacob, et al. Eigen. *URl: http://eigen. tuxfamily. org*, 2010.

[GKP$^+$11] Amol Ghoting, Rajasekar Krishnamurthy, Edwin P. D. Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 231–242, 2011.

[GM93] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 209–218, 1993.

[GRS$^+$13] Matthias Grimmer, Manuel Rigger, Lukas Stadler, Roland Schatz, and Hanspeter Mössenböck. An efficient native function interface for java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, September 11-13, 2013*, pages 35–44, 2013.

[Gru99] Torsten Grust. *Comprehending queries.* PhD thesis, University of Konstanz, Germany, 1999.

[Gru15] Joel Grus. *Data science from scratch: first principles with python.* "O'Reilly Media, Inc.", 2015.

[GS99] Torsten Grust and Marc H. Scholl. How to comprehend queries functionally. *J. Intell. Inf. Syst.*, 12(2-3):191–218, 1999.

[GSB$^+$18] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Tappan Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, pages 213–231, 2018.

[GSS$^+$15] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. High-performance cross-language interoperability in a multi-language runtime. In *Proceedings of the 11th Symposium on Dynamic Languages, DLS 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 78–90, 2015.

[Guh10] Saptarshi Guha. Computing environment for the statistical analysis of large and complex data. 2010.

[GW14] Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 339–347, 2014.

[HAS$^+$14] Po-Sen Huang, Haim Avron, Tara N. Sainath, Vikas Sindhwani, and Bhuvana Ramabhadran. Kernel methods match deep neural networks on TIMIT. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2014, Florence, Italy, May 4-9, 2014*, pages 205–209, 2014.

[HBY13] Botong Huang, Shivnath Babu, and Jun Yang. Cumulon: optimizing statistical data analysis in the cloud. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1–12, 2013.

[HH10] David Harris and Sarah Harris. *Digital design and computer architecture.* Morgan Kaufmann, 2010.

[HJLS13] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*, volume 398. John Wiley & Sons, 2013.

[HLH$^+$11] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 1199–1208, 2011.

[HM12] Max Heimel and Volker Markl. A first step towards gpu-assisted query optimization. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2012, Istanbul, Turkey, August 27, 2012.*, pages 33–44, 2012.

[HNP09] Alon Y. Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[HPS$^+$12] Fabian Hueske, Mathias Peters, Matthias Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. Opening the black boxes in data flow optimization. *PVLDB*, 5(11):1256–1267, 2012.

[IEE17] IEEE Spectrum. Ieee spectrum, the 2017 top programming languages. https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages, 2017. [Online; accessed 2017-10-23].

[Jac05] J Edward Jackson. *A user's guide to principal components*, volume 587. John Wiley & Sons, 2005.

[JD88] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data.* Prentice-Hall, 1988.

[JYP$^+$17] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt

Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 1–12, 2017.

[KAKM16] Andreas Kunft, Alexander Alexandrov, Asterios Katsifodimos, and Volker 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, BeyondMR@SIGMOD 2016, San Francisco, CA, USA, July 1, 2016*, page 1, 2016.

[KBB+15] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A modern, open-source SQL engine for hadoop. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[KC14] Mijung Kim and K. Selçuk Candan. Tensordb: In-database tensor manipulation with tensor-relational query plans. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014*, pages 2039–2041, 2014.

[KKC+17] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman P. Amarasinghe. The tensor algebra compiler. *PACMPL*, 1(OOPSLA):77:1–77:29, 2017.

[KKS+17] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Tilmann Rabl, and Volker Markl. Blockjoin: Efficient matrix partitioning through joins. *PVLDB*, 10(13):2061–2072, 2017.

[KKS+19] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. An intermediate representation for optimizing machine learning pipelines. *PVLDB*, 12(11):1553–1567, 2019.

[KMNP15] Arun Kumar, Robert McCann, Jeffrey F. Naughton, and Jignesh M. Patel. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Record*, 44(4):17–22, 2015.

[Kna15] Jochen Knaus. *snowfall: Easier cluster computing (based on snow).*, 2015. R package version 1.84-6.1.

[KNP15] Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1969–1984, 2015.

[KNPZ16] Arun Kumar, Jeffrey F. Naughton, Jignesh M. Patel, and Xiaojin Zhu. To join or not to join?: Thinking twice about joins before feature selection. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 19–34, 2016.

[Koh95] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pages 1137–1145, 1995.

[KSB$^+$18] Andreas Kunft, Lukas Stadler, Daniele Bonetta, Cosmin Basca, Jens Meiners, Sebastian Breß, Tilmann Rabl, Juan José Fumero, and Volker Markl. Scootr: Scaling R dataframes on dataflow systems. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, pages 288–300, 2018.

[KTD$^+$13] Tim Kraska, Ameet Talwalkar, John C. Duchi, Rean Griffith, Michael J. Franklin, and Michael I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[KTF09] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A peta-scale graph mining system. In *ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009*, pages 229–238, 2009.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88, 2004.

[Lan13] Brett Lantz. *Machine learning with R.* Packt Publishing Ltd, 2013.

[LG16] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 4013–4021, 2016.

[LGG$^+$18] Shangyu Luo, Zekai J. Gao, Michael N. Gubanov, Luis Leopoldo Perez, and Christopher M. Jermaine. Scalable linear algebra on a relational database system. *SIGMOD Record*, 47(1):24–31, 2018.

[LGK$^+$12] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[LHB12] Harold Lim, Herodotos Herodotou, and Shivnath Babu. Stubby: A transformation-based optimizer for mapreduce workflows. *PVLDB*, 5(11):1196–1207, 2012.

[LHKK79] Charles L. Lawson, Richard J. Hanson, D. R. Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[LKC$^+$17] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison: A fast JSON parser for data analytics. *PVLDB*, 10(10):1118–1129, 2017.

[LKF05] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, Illinois, USA, August 21-24, 2005*, pages 177–187, 2005.

[LR99] Zhe Li and Kenneth A. Ross. Fast joins using join indices. *VLDB J.*, 8(1):1–24, 1999.

[LRU14] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014.

[LYF$^+$10] Chao Liu, Hung-chih Yang, Jinliang Fan, Li-Wei He, and Yi-Min Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 681–690, 2010.

[LZY$^+$13] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, volume 6, page 2, 2013.

[MAAC14] Stefan C. Müller, Gustavo Alonso, Adam Amara, and André Csillaghy. Pydron: Semi-automatic parallelization for multi-core and the cloud. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 645–659, 2014.

[MAB$^+$10] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on*

Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010, pages 135–146, 2010.

[MBY^{+}16] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.*, 17:34:1–34:7, 2016.

[MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.

[McK12] Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython.* " O'Reilly Media, Inc.", 2012.

[Mei11] Erik Meijer. The world according to LINQ. *Commun. ACM*, 54(10):45–51, 2011.

[ML13] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

[MMS99] Christopher D Manning, Christopher D Manning, and Hinrich Schütze. *Foundations of statistical natural language processing.* MIT press, 1999.

[MNW^{+}18] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, pages 561–577, 2018.

[MR94] Robert Marek and Erhard Rahm. TID hash joins. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94), Gaithersburg, Maryland, USA, November 29 - December 2, 1994*, pages 42–49, 1994.

[MRT18] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning.* MIT press, 2018.

[Mul90] James K. Mullin. Optimal semijoins for distributed database systems. *IEEE Trans. Software Eng.*, 16(5):558–560, 1990.

[Mur12] Kevin P. Murphy. *Machine learning - a probabilistic perspective.* Adaptive computation and machine learning series. MIT Press, 2012.

[MW17] Microsoft Corporation and Stephen Weston. *doSNOW: Foreach Parallel Adaptor for the 'snow' Package*, 2017. R package version 1.0.15.

[Neu11] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[NLSW16] Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip 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 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 25–36, 2016.

[NW72] John Ashworth Nelder and Robert WM Wedderburn. Generalized linear models. *Journal of the Royal Statistical Society: Series A (General)*, 135(3):370–384, 1972.

[OR11] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 949–960, 2011.

[ORS$^+$08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1099–1110, 2008.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[PPR$^+$09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 165–178, 2009.

[PSR14] Orestis Polychroniou, Rajkumar Sen, and Kenneth A. Ross. Track join: distributed joins with minimal network traffic. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1483–1494, 2014.

[PTN$^+$18] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. Evaluating end-to-end optimization for data analytics applications in weld. *PVLDB*, 11(9):1002–1015, 2018.

[PTS$^+$17] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Malte Schwarzkopf, Saman P. Amarasinghe, and Matei Zaharia. A common runtime for high performance data analysis. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspot tm server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine*

Research and Technology Symposium-Volume 1, pages 1–1. USENIX Association, 2001.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, 2011.

[PVSK18] Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. Unifying analytic and statically-typed quasiquotes. *PACMPL*, 2(POPL):13:1–13:33, 2018.

[R+11] Philip Russom et al. Big data analytics. *TDWI best practices report, fourth quarter*, 19(4):1–34, 2011.

[RAM+12] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. Scala-virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation*, 25(1):165–207, 2012.

[Ras18] Sebastian Raschka. Model evaluation, model selection, and algorithm selection in machine learning. *CoRR*, abs/1811.12808, 2018.

[RGM16] Manuel Rigger, Matthias Grimmer, and Hanspeter Mössenböck. Sulong - execution of llvm-based languages on the JVM: position paper. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, ICOOOLPS@ECOOP 2016, Rome, Italy, July 17-22, 2016*, pages 7:1–7:4, 2016.

[RIKN16] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 1194–1205, 2016.

[RK91] Nick Roussopoulos and Hyunchul Kang. A pipeline n-way join algorithm based on the 2-way semijoin program. *IEEE Trans. Knowl. Data Eng.*, 3(4):486–495, 1991.

[RO10] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*, pages 127–136, 2010.

[RRWN11] Benjamin Recht, Christopher Ré, Stephen J. Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural*

Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain., pages 693–701, 2011.

[SBJ⁺18] Sebastian Schelter, Felix Bießmann, Tim Januschowski, David Salinas, Stephan Seufert, and Gyuri Szarvas. On challenges in machine learning model management. *IEEE Data Eng. Bull.*, 41(4):5–15, 2018.

[SBO13] Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for scala. Technical report, 2013.

[SHG⁺15] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2503–2511, 2015.

[SKKR01] Badrul Munir Sarwar, George Karypis, Joseph A. Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the Tenth International World Wide Web Conference, WWW 10, Hong Kong, China, May 1-5, 2001*, pages 285–295, 2001.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10, 2010.

[SLB⁺11] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. Optiml: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, pages 609–616, 2011.

[Smi17] David Smith. R, then and now. useR!, Brussels, 2017, 2017.

[SMYT14] D.S. Starnes, D.S. Moore, D. Yates, and J. Tabor. *The Practice of Statistics.* W. H. Freeman, 2014.

[SPQ⁺16] Sebastian Schelter, Andrew Palumbo, Shannon Quinn, Suneel Marthi, and Andrew Musselman. Samsara: Declarative machine learning on distributed dataflow systems. In *Machine Learning Systems workshop at NeurIPS*, 2016.

[SS02] Bernhard Schölkopf and Alexander Johannes Smola. *Learning with Kernels: support vector machines, regularization, optimization, and beyond.* Adaptive computation and machine learning series. MIT Press, 2002.

[SSM+15] Sebastian Schelter, Juan Soto, Volker Markl, Douglas Burdick, Berthold Reinwald, and Alexandre V. Evfimievski. Efficient sample generation for scalable meta learning. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1191–1202, 2015.

[STH+15] Evan R. Sparks, Ameet Talwalkar, Daniel Haas, Michael J. Franklin, Michael I. Jordan, and Tim Kraska. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*, pages 368–380, 2015.

[STS+13] Evan R. Sparks, Ameet Talwalkar, Virginia Smith, Jey Kottalam, Xinghao Pan, Joseph E. Gonzalez, Michael J. Franklin, Michael I. Jordan, and Tim Kraska. MLI: an API for distributed machine learning. In *2013 IEEE 13th International Conference on Data Mining, Dallas, TX, USA, December 7-10, 2013*, pages 1187–1192, 2013.

[SVK+17] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 535–546, 2017.

[SWHJ16] Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. Optimizing R language execution via aggressive speculation. In *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*, pages 84–95, 2016.

[SY93] James W. Stamos and Honesty C. Young. A symmetric fragment and replicate algorithm for distributed joins. *IEEE Trans. Parallel Distrib. Syst.*, 4(12):1345–1354, 1993.

[T+15] R Core Team et al. R: A language and environment for statistical computing. 2015.

[TER18] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. How to architect a query compiler, revisited. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 307–322, 2018.

[THS+09] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. Query processing techniques for solid state drives. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 59–72, 2009.

[TK18] Anthony Thomas and Arun Kumar. A comparative evaluation of systems for scalable linear algebra-based analytics. *PVLDB*, 11(13):2168–2182, 2018.

[TÖZ+16] Yuanyuan Tian, Fatma Özcan, Tao Zou, Romulo Goncalves, and Hamid Pirahesh. Building a hybrid warehouse: Efficient joins between data stored in HDFS and enterprise warehouse. *ACM Trans. Database Syst.*, 41(4):21:1–21:38, 2016.

[TRLS16] Luke Tierney, A. J. Rossini, Na Li, and H. Sevcikova. *snow: Simple Network of Workstations*, 2016. R package version 0.4-2.

[TS97] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97), Amsterdam, The Netherlands, June 12-13, 1997*, pages 203–217, 1997.

[TSJ$^+$09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[VBB$^+$18] Ashish Vaswani, Samy Bengio, Eugene Brevdo, François Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Lukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. Tensor2tensor for neural machine translation. *CoRR*, abs/1803.07416, 2018.

[Voh16] Deepak Vohra. Apache parquet. In *Practical Hadoop Ecosystem*, pages 325–335. Springer, 2016.

[VYL$^+$16] Shivaram Venkataraman, Zongheng Yang, Davies Liu, Eric Liang, Hossein Falaki, Xiangrui Meng, Reynold Xin, Ali Ghodsi, Michael J. Franklin, Ion Stoica, and Matei Zaharia. Sparkr: Scaling R programs with spark. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1099–1104, 2016.

[W$^+$17] Hadley Wickham et al. A grammer of data manipulation. CRAN, 2017.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*, pages 344–358, 1988.

[WDL$^+$09] Kilian Q. Weinberger, Anirban Dasgupta, John Langford, Alexander J. Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, pages 1113–1120, 2009.

[Wes17] Stephen Weston. *doMPI: Foreach Parallel Adaptor for the 'Rmpi' Package*, 2017. R package version 0.2.2.

[WLMO11] Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. Query optimization for massively parallel data processing. In *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*, page 12, 2011.

[WWS$^+$12] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages, DLS '12, Tucson, AZ, USA, October 22, 2012*, pages 73–82, 2012.

[WWW$^+$13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 187–204, 2013.

[YAB$^+$18] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Gordon Murray, and Xiaoqiang Zheng. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 18:1–18:15, 2018.

[YZP14] Oscar D. Lara Yejas, Weiqiang Zhuang, and Adarsh Pannu. Big R: large-scale analytics on hadoop using R. In *2014 IEEE International Congress on Big Data, Anchorage, AK, USA, June 27 - July 2, 2014*, pages 570–577, 2014.

[ZCF$^+$10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, 2010.

[ZKR14] Ce Zhang, Arun Kumar, and Christopher Ré. Materialization optimizations for feature selection workloads. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 265–276, 2014.

[ZKR16] Ce Zhang, Arun Kumar, and Christopher Ré. Materialization optimizations for feature selection workloads. *ACM Trans. Database Syst.*, 41(1):2:1–2:32, 2016.

[ZLC10] Jingren Zhou, Per-Åke Larson, and Ronnie Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 1060–1071, 2010.

[ZMU$^+$16] Reza Bosagh Zadeh, Xiangrui Meng, Alexander Ulanov, Burak Yavuz, Li Pu, Shivaram Venkataraman, Evan R. Sparks, Aaron Staple, and Matei Zaharia. Matrix computations and optimization in apache spark. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 31–38, 2016.

List of Figures

List of Tables

LISTINGS

LIST OF ACRONYMS

API application programming interface

AST Abstract Syntax Tree

BGD batch gradient descent

BSP bulk synchronous parallel

CSC Compressed Sparse Column

CSE Common Subexpression Elimination

CSR Compressed Sparse Row

CSV comma-separated values

CV cross-validation

DAG directed acyclic graph

DOP degree-of-parallelism

DSL domain-specific language

ETL Extract, Transform, and Load

FK foreign-key

GFS Google File System

GPL general-purpose programming language

HDFS Hadoop Distributed File System

IDE integrated development environment

IPC inter-process communication

IR intermediate representation

JIT just-in-time

JSON JavaScript Object Notation

JVM Java Virtual Machine

LIR low-level intermediate representation

LNF let-normal form

ML machine learning

MPI Message Passing Interface

MR MapReduce

OLAP online analytical processing

PCA principal component analysis

PK primary-key

PVM Parallel Virtual Machine

RDBMS relational database management system

RDD resilient distributed dataset

SSA static single assignment form

STS source-to-source translation

SVM support-vector machine

TPU Tensor Processing Unit

UDAF user-defined aggregate function

UDF user-defined function