UNIVERSITY OF CALIFORNIA SAN DIEGO

Write once, rewrite everywhere:
A Unified Framework for Factorized Machine Learning

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

David Justo

Committee in charge:

Professor Arun Kumar, Chair
Professor Ranjit Jhala
Professor Nadia Polikarpova

2019

The thesis of David Justo is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

Chair

University of California San Diego

2019

DEDICATION

To my cousin Diego, who first introduced me to computing and inadvertently led me to this wild journey; *muchas gracias!*

EPIGRAPH

We have also obtained a glimpse of another crucial idea about languages and program design. This is the approach of stratified design, the notion that a complex system should be structured as a sequence of levels that are described using a sequence of languages. Each level is constructed by combining parts that are regarded as primitive at that level, and the parts constructed at each level are used as primitives at the next level. The language used at each level of a stratified design has primitives, means of combination, and means of abstraction appropriate to that level of detail.

—H. Abelson and G. Sussman (in ”The Structure and Interpretation of Computer Programs)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I want to use this space to reflect back on all the people who have helped me get to this point. Looking back, and considering the circumstances of my family and upbringing, I'm humbled and thankful for my experience at UC San Diego. I've been constantly overwhelmed, surprised, and excited by the wealth of opportunities that were made available to me here from day one; I am now certain that I made the right decision when I bought a one-way ticket to California just five years ago.

As an undergraduate, I made many good friends with whom I shared some good laughs and ridiculous memes. I was lucky to find great housemates in Todd Tang, Ethan Vander-Horn, Jacqui Bontigao, Steven Truong, Natalie Nguyen, Julia Kapich, and Igen Foreman. Outside the apartment, I spent a lot of time hanging out with *daBoise*: Arvind Kalathil, Jonathan Perapalanunt, Karan Lala, Todd Tang (again), Ethan Brand, Alex Kenji Barcenas, and Youngjin Yun, with whom I spent many long nights calculating derivatives. Many of these friends I met thanks to the SPIS program, so I want to thank Mohan Paturi for organizing it.

At that time, I was also really invested in the Data Science Student Society, so I have to thank Daniel Maryanovsky, for running the workshops with me and being a friendly mentor, and to Liz Izhikevich, for running the club logistics with me. After I left the org, Sim Bhatia stepped up to manage the club and took it to a whole other level, thank you. Bradley Voytek was key in making the club successful and he continues to play a part in that, so I'm grateful for his continued support.

I got involved in research very early on as an undergrad. Those early experiences taught me a lot and helped me identify key habits that I needed to incorporate in order to be productive as a researcher. So I want to thank Julian McAuley, Zhouwen Tu, and the San Diego Supercomputer Center for giving me a chance and for their patience when I still had much to learn.

I embarked on the Master's program to give research another shot, to prove to myself that I could be successful in this environment and to determine if academia was a path I wanted to

pursue in the long term. I'm immensely thankful to Nadia Polikarpova's mentoring and advice throughout this whole experience. Working with her, I understood what it takes to bring a research project from early stages to publication, and that experience will remain with me as a guiding model. I was also very kindly welcomed at UCSD's PL group and greatly enjoyed getting to know everyone in the community. I want to thank Valentin Robert for introducing me to the group and giving me access to many resources when I was just getting started. I also want to thank Ranjit Jhala, whose feedback has made me much more cognisant of what a good presentation should be like. In no particular order, I'm also thankful for many insightful conversations and feedback from Anish Tondwalkar, Tristan Knoth, Alex Sanchez Stern, John Sarracino, Matthew Kolosick, Shravan Narayan, Rose Kunkel, John Renner, Peter Amidon, and Dylan Lukes. Finally, I want to give special thanks to Zheng Guo, Michael James, Ziteng Wang and Jiaxiao Zhou; the other students of the Hoogle+ team. I'm humbled to have worked with y'all, thank you *so much*.

In the later half of my research adventure, I got involved with UCSD's Database group, which led me to this thesis. I want to thank Arun Kumar, my advisor for this project, whose work I built on top of and whose feedback has shaped my thinking when reasoning about data-intensive workflows. If I were to pursue a PhD in the future, exploring the intersection between database theory and programming languages would certainly be a large part of it. Finally, this project would not have been possible without the support of Oracle Labs and, in particular, that of Lukas Stadler. I want to thank them for the mentoring and support they provided me and for giving me my "research in industry" experience as a grad student.

I also made many good friends in my MS adventure. I'm grateful to have met Alex Hancock and Naveen Kashyap, who accompanied me throughout this experience, spending long nights in the CSE Buildilng writing logical proofs and showing the convexity of one too many functions. There are many more people I could thank but I'm running out of space. UC San Diego has helped shape me as a person, a researcher, and as an engineer. Thank y'all for the many opportunities, lessons, and experiences. Through it all, I aimed to make the most of it.

This thesis, in full, is currently being prepared for submission for publication of the material. Justo, David; Stadler, Lukas; Kumar, Arun. The thesis author is the primary investigator and author of this material.

ABSTRACT OF THE THESIS

Write once, rewrite everywhere:
A Unified Framework for Factorized Machine Learning

by

David Justo

Master of Science in Computer Science

University of California San Diego, 2019

Professor Arun Kumar, Chair

This thesis describes TRINITY, a framework to optimize linear algebra algorithms operating over relational data in GraalVM. The framework implements a host-language-agnostic version of the optimizations introduced by the Morpheus project, meaning that a single implementation of the Morpheus rewrite rules can be used to optimize linear algebra algorithms written in arbitrary GraalVM languages. We evaluate its performance when hosted within FastR and GraalPython, GraalVM's R and Python implementations respectively. In doing so, we also show that TRINITY can optimize across languages, meaning that it can execute and optimize an algorithm written in one language, such as Python, while using data originating from another language, such as R.

Chapter 1

Motivating Example

Machine Learning systems frequently operate over massive relational datasets, leading them to often require hours, if not days, to terminate. In this setting, memory efficiency also becomes an obstacle as datasets require increasingly powerful machines to host them. This project aims to mitigate these concerns by providing a *language-agnostic* abstraction that automatically improves the runtime and memory-usage of learning algorithms operating over large multi-table datasets. To introduce and motivate our approach, we begin by presenting a hypothetical Data Science task, ground it on a real dataset, and use it to highlight the inefficiencies we aim to solve. That scenario goes as follows: A Data Scientist is tasked to explain what business characteristics generally lead to good ratings on Yelp, a popular crowd-sourced review forum. A good place to start this analysis is with Kaggle's Yelp dataset, using the same version as in [16], which was designed for this very task.

1.1 Dataset Description and Pre-processing

This specific Yelp dataset is multi-table, as are many freely-available datasets found in the wild. It is comprised of three tables: the first containing the rating provided by a user to a business, the second one providing metadata for each user, and the third one containing metadata

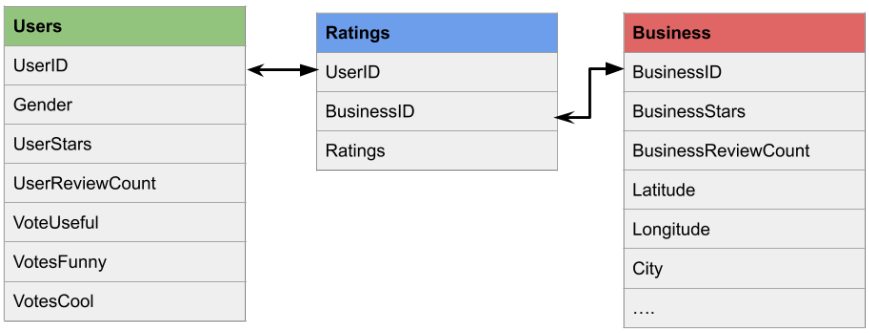


Figure 1.1: The Yelp dataset schema. The Ratings table contains two foreign keys associating it to each of the other two tables: UserID and BusinessID

for each business. We refer to these tables as **Ratings**, **Users**, and **Businesses** respectively.

Figure 1.1 exhibits the structure, i.e schema, of the dataset at hand. The **Users** table contains tuples of primary key (PK) UserID followed by attributes: Gender, UserStars, UserReviewCount, VoteUseful, VotesFury, and VotesCool. Meanwhile, the **Businesses** table contains tuples with their PKs in BusinessID and columns titled: BusinessStars, BusinessReivew, Count, Lattitude, Longitude, and others that we omit for this example. Finally, UserID and BusinessID are foreign keys (FK) in **Ratings**, enabling the tables to be linked together via table-joins and matched with the **Ratings**'s Rating column. This is convenient because many Machine Learning algorithm implementations expect their input data to be in a single-table format [12], so the Yelp dataset needs to be joined before we can train most off-the-shelf models with it.

1.2 Trapped by the matrix: How joins slow down models

To establish a baseline predictive model, our Data Scientist trains a Logistic Regression model to predict, for any user-business pair, if the user would rate the business higher than 2.5 stars, in a scale ranging from 1 to 5 stars. To train this model using the Yelp dataset, as implemented in Listing 1, with hyper-parameters `gamma0` $= 0.000001$ and `Max_Iter` $= 100$, it would takes around 30 seconds in a modern laptop.

```
LogisticRegression <- function(Data, Max_Iter, winit, gamma0, Target)
{
  w = winit;
  for( k in 1: Max_Iter )
  {
    w = w - gamma0 * (t(Data) %*% (Target / (1 + exp(Data%*%w))));
  }
  return(list(w));
}
```

Listing 1: A sample Logistic Regression implementation in R

The training time of this model could have been much less. Our current workflow is inefficient, in large part, because we are working with a redundant representation of our dataset: the materialized matrix resulting from a join operation. While joining multi-table datasets is a common, often expected, pre-processing step in Machine Learning programs, doing so tends to introduce redundancy to the resulting matrix. In our Yelp dataset, the attributes of popular businesses are likely to be repeated multiple times in the new matrix because popular businessIDs have many reviews. The same applies to very active users, whose corresponding rows in Users will be repeated multiple times. This means that the resulting matrix's cell count will be larger than the sum of cells of the original matrices.

Users (9 cells)

UserID	Gender	Age
Joe	M	20
Agent Smith	M	30
Maria	F	40

Ratings (18 cells)

UserID	Stars	BusinessID
Joe	3	Bob's Burgers
Agent Smith	2	Bob's Burgers
Maria	4	Primos Tacos
Agent Smith	5	Primos Tacos
Agent Smith	2	Vallarta Tacos
Agent Smith	3	McDonald's

Materialized Join (30 cells)

UserID	Stars	BusinessID	Gender	Age
Joe	3	Bob's Burgers	M	20
Agent Smith	2	Bob's Burgers	M	30
Maria	4	Primos Tacos	F	40
Agent Smith	5	Primos Tacos	M	30
Agent Smith	2	Vallarta Tacos	M	30
Agent Smith	3	McDonald's	M	30

Figure 1.2: Example of a join introducing redundancy. Joining Ratings and Users by UserID creates a matrix with more cells than the sum of the cells in the original matrices

Redundancy in the input data leads to wasted computation in ML algorithms. Given that ML pipelines are already resource-hungry and time-demanding, how can our Data Scientist circumvent these inefficiencies? One alternative would be to ”factorize” the algorithm: to rewrite the Logistic Regression model to operate directly on the original three tables [14, 21, 15, 20]. With enough effort and algebraic prowess, one may stumble upon an equivalent implementation that exploits the relational nature of the data. Unfortunately, this is time consuming to do and hard to scale.

1.3 Escape the matrix: The Morpheus alternative

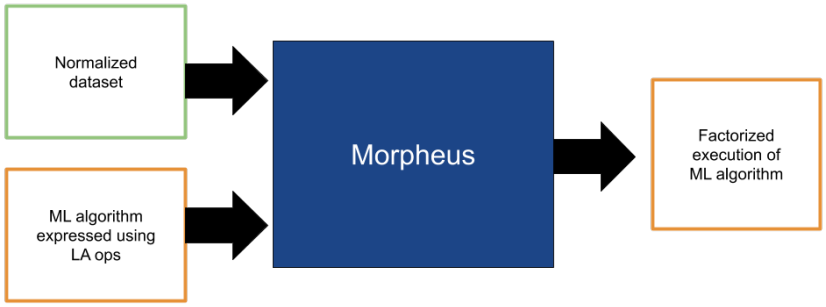


Figure 1.3: High-level description of the Morpheus system. Given a normalized (multi-table) dataset and some ML algorithm implemented with linear algebra operators, Morpheus executes the ML algorithm as-if it had been written to operate directly over the normalized dataset

The Morpheus project suggests a solution. It enables pre-existing ML algorithm implementations to execute *as if* they had been factorized; therefore avoiding the need for redundancy-inducing joins as a pre-processing step. The framework works by exporting an alternative matrix datatype, the Normalized Matrix, which serves to represent the result of a join operation. However, the Normalized Matrix is lazy, so it avoids eagerly performing the join. Instead, it stores the multi-tabel dataset in an alternative representation for which overloaded LA operator semantics are provided. In doing so, the Normalized Matrix automatically yields a factorized execution of an otherwise ”standard” Logistic Regression algorithm implementation [12, 17].

1.4 Meet TRINITY: A VM-level agent of Morpheus

The Morpheus project remains in active development. At the time of writing this thesis, new rewrite rules have been introduced to facilitate optimizing models that require or generate feature interactions over a multi-table dataset [17]. While open-source implementations of Morpheus exist and are a free to use [9, 8, 10, 7, 6], keeping all of them up-to-date with all the latest research progress is a daunting task; so they may not be available for our Data Scientists language or LA system of choice. In fact, because of the ever rapidly expanding list of linear algebra systems, languages, and libraries, it can be difficult to export the Morpheus framework to all relevant players in the field. Wouldn't it be wonderful to have a single, unified specification of the Morpheus rewrite rules that many language toolkits and toolchains could embed with low-to-zero developer burden? In this scenario, the Morpheus core logic could evolve while still supporting many front-ends. This is precisely the goal of the TRINITY system, the work presented in this thesis.

We build TRINITY using the GraalVM infrastructure, a virtual machine (VM) providing a unified runtime for many languages including R, Python, and JavaScript [24, 23]. In working with this framework, we build on top of its language-interoperability features, and contribute some of our own, to support a language-agnostic implementation of Morpheus that is easy to embed inside arbitrary Truffle-supported programming languages. Finally, this thesis' contributions are best understood by considering the outputs for each of its three main stakeholders.

1. **For the Data Scientist**: Morpheus semantics made available for the R and Python implementations in GraalVM.

2. **For the Truffle developer**: the introduction of a host-language-agnostic interface to manipulate matrix datatypes in Truffle.

3. **For the GraalVM language ecosystem**: a new, first-of-its-kind, Truffle language: the MorpheusDSL.

Chapter 2

Background

2.1 The GraalVM project

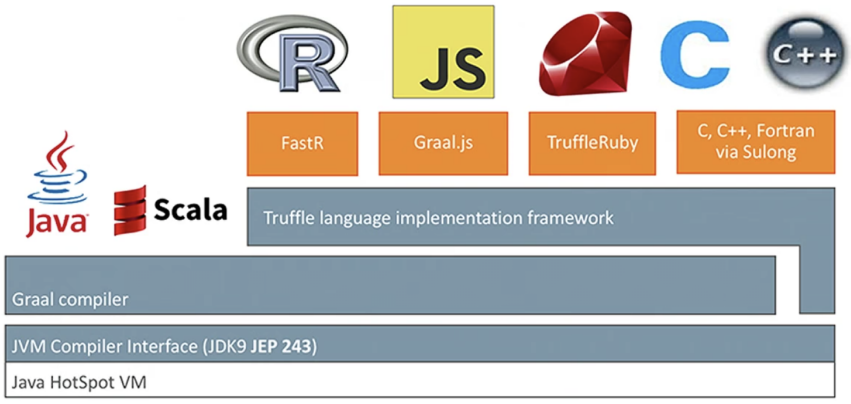


Figure 2.1: Architectural overview of GraalVM. Languages are implemented using the Truffle framework and are compiled to run on top of the Java HotSpot VM

The GraalVM project aims to accelerate the development of programming languages by amortizing the cost of building language-specific virtual machines [24]. GraalVM languages are implemented in a framework named Truffle which enables them to run on GraalVM, a modified version of Java's HotSpot VM. During execution, GraalVM captures runtime information to optimize and compile the Truffle ASTs to Java bytecode. This approach to language development

has lead to the speedy development production-ready re-implementations of many dynamic languages including JavaScript, R, and Ruby [22, 25].

2.1.1 The Truffle language implementation framework

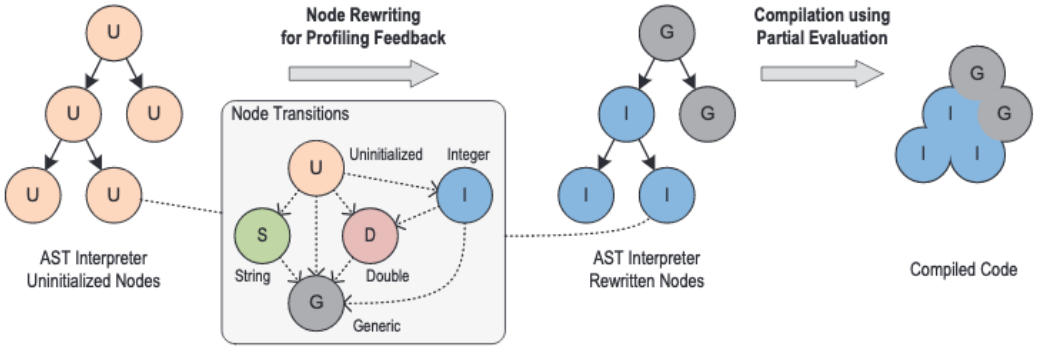


Figure 2.2: Truffle uses profiling feedback to speculate on future inputs and compile AST nodes into specialized variants. A more generic implementation always needs to be available in case the assumptions are invalidated.

In Truffle, a language is implemented by specifying an AST interpreter for it. Using the AST interpreter as input, GraalVM uses a technique called Partial Evaluation to derive a Java bytecode-producing compiler for the language [22].

In addition to a default implementation for each AST node, language designers are encouraged to provide alternative implementations that provide high-performance when operating over a subset of the inputs [25]. This is because, at runtime, the VM will make speculative assumptions about future inputs and use them to compile AST nodes into their optimized variants. If these assumptions are ever invalidated, the code is *deoptimized* and the node replaced with a more general implementation. Therefore, it is always necessary that a default implementation of each AST node, one that handles all inputs, is provided. Finally, Truffle AST nodes are implemented using a subset of Java and by relying on an annotation pre-processor to handle boilerplate around the interpreter.

2.1.2 Polyglot Programs and Interoperability

```
array <- eval.polyglot("python", "[1,2,42,4]");
print(array[3L]) # prints 42
```

Listing 2: R using its *polyglot-eval* function to create a python list and access its elements

A salient feature of the GraalVM runtime is that, by having multiple language share the same implementation framework, Truffle can seamlessly combine nodes from different languages within a single AST. This, among other mechanisms, enables GraalVM users to combine multiple languages in the same script, enabling cross-language partial evaluation and for the dynamic compiler to optimize across languages [13].

At the Truffle-level, language developers have access to the INTEROP protocol, an API to inspect and interact with foreign language datatypes when implementing AST nodes [5]. Unfortunately for us, INTEROP requires its users to know what method and attribute names are exported by the foreign object. As a result, a Truffle node implementation expecting foreign language inputs will often need to handle each foreign datatype as a separate case, even when the inputs all represent the same underlying data structure such as a tree, a hashmap, or a matrix. Our work alleviates this problem by providing an API for interacting with matrix datatypes in the same way regardless of their language of origin.

For the end-user, languages have access to a built-in *polyglot-eval* function that allows them to syntactically-embed fragments of other programming languages within the same program [4]. With it, users can, for instance, load some dataset as an R vector but send it over to some Python function to calculate summary statistics about the data. For some key datatypes, Truffle automatically translates them when mixing languages. For example, an R vector will automatically get transformed into a Python list. However, a specialized datatype such as a Python NumPy array cannot be translated directly into an R matrix, so it becomes an opaque foreign object with methods and attributes that can be reached from R if the user already knows its interface.

2.2 Factorized Machine Learning with Morpheus

Almost all Machine Learning (ML) algorithms assume that its input data exists in a single table format. Since many real-world data-sets are multi-table, programmers are often expected to join tables as a pre-processing step. Since data is more compact when normalized, ML algorithms often perform redundant computations by operating over the materialized joins of multi-table datasets. In this setting, "factorized" ML refers to alternative implementations of ML algorithms that operate directly on multi-table datasets [14, 21, 15, 20]. Providing alternative implementations for the ocean of ML algorithms is a daunting task which the Morpheus project aims to tackle by automatically factorizing otherwise "regular" ML algorithm implementations. This is achieved by introducing a new abstraction over joins, called the Normalized Matrix, which provides alternative semantics, i.e rewrite rules, for LA operators common in ML algorithms [12, 17]. With this approach, the Morpheus project aims to unlock the benefits of factorized Machine Learning without needing drastic edits to a pre-existing codebase or an LA system.

2.2.1 Notation

Table 2.1: Notation used in this thesis and shared by the Morpheus lineage of projects

Symbol	Explanation
R / R	Attribute table / feature matrix
S / S	Entity table / feature matrix
T / T	Materialized Join / feature matrix
K	Indicator matrix for PK-FK join
n_S / n_R	Number of rows in S / R
Y	Prediction Target
d_S / d_R / d	number of features in S / R / T

We begin by introducing the notation and naming conventions shared across the Morpheus lineage of projects, as introduced in [12]. For simplicity, consider the PK-FK join (a primary

key to foreign-key join) between the tables $\mathbf{R}(\underline{RID}, X_R)$ and $\mathbf{S}(Y, X_S, K)$ such that X_R and X_S are *feature vectors*, Y is the prediction *target*, K is the foreign key and *RID* is the primary key in **R**. We also refer to **R** as the *attributes* table and to **S** as the *entity* table. The materialized join is denoted as $\mathbf{T}(\mathrm{Y}, [X_S, X_R]) \leftarrow \pi(\mathbf{S} \bowtie_{K=RID} \mathbf{R})$ where $[X_S, X_R]$ is the column-wise concatenation of the feature matrices. Finally, we use a standard data representation notation for the feature matrices: R as shorthand for $\mathbf{R}.X_R$, and similarly for **S** and **T**. For multi-table joins, we always have multiple *attributes* tables but a single *entity* table, so we identify each **R** table with a subscript as in $\mathbf{R}_i$. Our complete notation is summarized in table 2.1.

Example. Consider the join between **Users**(UserID, Gender, Age) and **Ratings**(UserID, Stars, BusinessID) depicted in figure 1.2. In this case, **Ratings** is **R**, **Users** is **S**, **Users**.UserID is *RID*, **Ratings**.UserID is K, Y is Stars (remember our running example aims to predict ratings), X_S is empty and X_R is {Gender, Age} .

2.2.2 The Normalized Matrix: An abstraction for table joins

The Normalized Matrix is a Morpheus datatype representing a matrix constructed from table joins [18, 12, 17]. Consider the materialized feature matrix T resulting from the PK-FK join between entity table **S**(Y S,K) and attribute table **R**(*RID*, R). We show how to construct a Normalized Matrix for this join operation. First, note that $\mathbf{S}.K$ is a foreign key field mapping to some value in $\mathbf{R}.RID$. Second, observe that since each $\mathbf{R}.RID$ can be mapped to its sequential row number in **R**, we can construct a sparse indicator matrix K as follows:

$$K[i,j] = \begin{cases} 1, & \text{if } i^{th} \textit{ row of } \mathbf{S}.K = j \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

The Normalized Matrix alternative to T is then simply the matrix triple $T_N \equiv (S, K, R)$. Observe that the materialized matrix alternative corresponds to $[S, KR]$ so K can be thought of as

a a kind of ”row-selector” for the attributes table.

Even though the Normalized Matrix is formally just a matrix triple, users interact with it using the same LA operators that they would use to manipulate any standard matrix. In other words, LA operators are overloaded to operate efficiently over the matrix triple so that existing ML algorithms would need zero or minimal edits to operate on this datatype as its input. When a normalized matrix is passed as the input to some ML algorithm, its overloaded LA operators effectively factorize the algorithm's execution.

In this work, we focus on implementing a Normalized Matrix for PK-FK joins, although alternative implementations exist for M:N joins as well. Next, we describe LA operator semantics for simple PK-FK joins; the generalization to multi-table joins may be found in [17].

2.2.3 Normalized Matrix: LA operator rewrites

By interacting with the Normalized Matrix, ML algorithms execute as if they had been written to operate directly on the normalized dataset. This works because the Normalized Matrix exports rewrites of common LA operators as its interface. Here, we describe those rewrites, as presented in [12].

Element-wise Scalar Operators

$$\begin{aligned} T \oslash x &\rightarrow (S \oslash x, K, R \oslash x) \\ x \oslash T &\rightarrow (x \oslash S, K, x \oslash R) \\ f(T) &\rightarrow (f(x), K, f(R)) \end{aligned} \tag{2.2}$$

This rewrite rule describes how to execute unary element-wise operations such as *log* and *exp*, and binary operations with scalars like scalar addition and scalar multiplication. Observe that

this rewrite rule returns another matrix-tuple, another normalized matrix. This is special because the remaining rewrite rules return just standard matrices instead.

Aggregation Operators

$$\begin{aligned} sum(T) &\rightarrow sum(S) + colSums(K)rowSums(R) \\ rowSums(T) &\rightarrow rowSums(S) + KrowSums(R) \\ colSums(T) &\rightarrow rowSums(S) + KrowSums(R) \end{aligned} \tag{2.3}$$

Left- and Right- Matrix Multiplication

$$\begin{aligned} TX &\rightarrow SX[1:d_S,] + K(RX[d_S+1:d,]) \\ XT &\rightarrow [XS, (XK)R] \end{aligned} \tag{2.4}$$

2.2.4 Normalized Matrix: Complexity

In Factorized ML, performance gains result from avoiding redundant computations. These occur when a join results in multiple rows of **S** to match repeatedly with the same row in **R**. If so, the dimensionality of the resulting matrix is larger than the sum of its parts, the dimensionality of **S** and each $\mathbf{R}_i$.

Prior work in factorized ML identifies the *tuple ratio* (TR) and *feature ratio* (FR) as useful dimensions for evaluating the efficiency gains of the overloaded LA operators in the Normalized Matrix [15, 12, 17]. We compute TR as $\frac{n_S}{n_R}$ while FR equates to $\frac{d_R}{d_S}$. In the same vein, prior work expresses the asymptotic runtime efficiency gains of Morpheus linear algebra (LA) operators by measuring the number of arithmetic operations for each operator, which are expressed in terms of the dimensions of **S** and **R**. We present those asymptotic complexities in the table below.

Table 2.2: Asymptotic runtime complexity of the overloaded LA operators

Operator	**Standard**	**Factorized**
Scalar Op	$n_S(d_S + d_R)$	$n_S d_S + n_R d_R$
Aggregation	$n_S(d_S + d_R)$	$n_S d_S + n_R d_R$
LMM	$d_X n_S(d_S + d_R)$	$d_X(n_S d_S + n_R d_R)$
RMM	$n_X n_S(d_S + d_R)$	$n_X(n_S d_S + n_R d_R)$
crossprod	$\frac{1}{2}(d_S + d_R)^2 n_S$	$\frac{1}{2}d_S^2 n_S + \frac{1}{2}d_R^2 n_R + d_S n_R n_R$

Morpheus-style LA operators are not always faster than their counterparts. This occurs when a join does not introduce enough redundancy or the distribution of foreign keys in **S** is overly selective. More formally, this frequently occurs when either *TR* or *FR* are less than 1. In addition, depending on the underlying LA system, the rewrite in itself may introduce some minimal overhead even when both redundancy ratios are larger than 1. To combat this issue, Morpheus systems can incorporate a *heuristic decision rule* to decide whether or not *TR* and *FR* are large enough such that the Normalized Matrix is likely to improve performance. Calculating those thresholds is a matter of tuning and is LA-system-dependent [12].

2.2.5 The Morpheus implementation burden

Morpheus is not a standalone library written in a high-performance language like C++ that provides language bindings to LA systems. Instead, it is a *hosted* optimization system: it is re-implemented *within* some LA system (eg. NumPy or R) and uses its host's methods and constructs to implement a NormalizedMatrix. As a result, each re-implementation of Morpheus has to re-discover host-specific internals, which is time-consuming. Additionally, as the rewrite rules change or expand, and the number of supported LA systems grows, it would take increasingly more effort to keep each independent re-implementation of Morpheus up-to-date.

2.3 Non-invasive embedded DSLs

A domain-specific language (DSL) is a programming language designed to operate on a specialized domain. Notable examples of DSLs include: SQL, Prolog, CSS, and TensorFlow. These tools are often less computationally expressive than a general purpose programming language, albeit in exchange for desirable properties such as improved performance and increased programmer productivity.

DSLs may be implemented from scratch or otherwise re-use the infrastructure of an existing language. The latter is called an *embedded* DSL because they are executed via a syntactic subset of a more general programming language. Some embedded DSLs are originally introduced as libraries and only later envisioned as embedded languages — such is the case with TensorFlow — but the label of embedded DSL versus library in such cases tends to be merely a matter of opinion and computational background.

Some embedded DSLs also have the goal of being *non-invasive*. This means that the end-user should be able to trigger the optimized semantics of the DSL while minimizing the visible changes to their usual programming model with the host language. The latter description is borrowed from ParallelAccelerator, a non-invasive embedded DSL for Julia [11]. We believe that re-interpreting the Morpheus rewrite rules as the semantics of a non-invasive embedded DSL is useful in helping us achieve the automatic ”factorizing” of pre-existing Machine Learning algorithms. Additionally, thinking of Morpheus as a kind of limited programming language allows us to justify our uses of the tooling provided by the Truffle language platform.

Chapter 3

A Unified Framework for Factorized ML

3.1 Design Vision and Overview

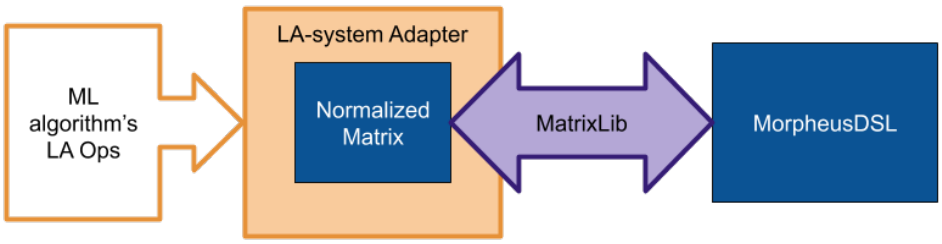


Figure 3.1: High-level components of TRINITY. The MorpheusDSL exports NormalizedMatrix objects, which are wrapped in LA-system-aware wrappers for compatibility with pre-existing ML algorithm implementations. Then, when LA operators are called, the NormalizedMatrix requests the overloaded semantics from MorpheusDSL via MATRIXLIB

Above all, we aim to make Morpheus' optimization logic widely available across languages and libraries while minimizing its implementation burden. In doing so, we broke from the tradition of re-implementing Morpheus with its host linear algebra framework in mind, instead choosing to make a distinction between the programmatic description of Morpheus' formal rewrite rules and the process of *embedding* them in some LA framework. Moreover, we wished to make the embedding process easy to execute, requiring few lines of code and little-to-no expertise about the underlying theory.

We realized these requirements in TRINITY, our framework for automatic factorized machine learning in GraalVM languages. We make novel use of GraalVM's language interoperability tooling by encoding the rewrite rules in a host-agnostic way and as a Truffle domain specific language that we call MorpheusDSL. At execution time, MorpheusDSL uses GraalVM's interoperability services to alter the abstract syntax tree (AST) of a linear algebra algorithm in some target language, optimizing its runtime over normalized data. To power this technique, we also developed a matrix-aware interoperability library that we call MATRIXLIB and that serves as a *unified* interface to manipulate arbitrary matrix datatypes, regardless of their programming language of origin. Finally, we argue that embedding MorpheusDSL optimizations within new LA systems requires no more than providing a simple adapter to map MorpheusDSL's Matrix interface to the corresponding method names in some target LA system.

Chapter 4

MATRILIB: A Foreign Matrix Interface

4.1 Overview

While Truffle has long provided APIs to facilitate language interoperability, interacting with foreign datatypes required knowing the specific details of their interface. For our purposes, this would have meant we needed to implement MorpheusDSL by handling **each** LA-system we wished to support as a separate case. That would defeat the purpose of this work, because we hoped to keep a separation between the description of Morpheus rewrite rules and host-system-specific concerns. What we needed was a *uniform* interface describing matrix manipulations, therefore enabling MorpheusDSL's optimization logic to be agnostic of its hosts.

We implemented MATRIXLIB to help bridge that gap. MATRIXLIB is a Truffle Library that provides a singular interface to execute common operations on foreign-language matrices. To achieve this, matrices sent to a foreign language context first need to be wrapped with an adapter that conforms to the MATRIXLIB API. This way, foreign languages have a reliable way to call upon common matrix operations on foreign matrices regardless of origin.

4.2 Motivation: Why we need a matrix-aware library

```
@GenerateUncached
abstract static class ExampleNode extends Node {

    protected abstract Object execute(Morpheus receiver)
        throws UnsupportedMessageException;

    @Specialization
    Object doDefault(Morpheus receiver, Object matrix,
        @CachedLibrary("matrix") InteropLibrary interop)
        throws UnsupportedMessageException {

        Object output = null

        // Need to handle each kind of matrix separately
        boolean isPythonMatrix = interop.isMemberInvocable(matrix, "__mul__");
        if(isPythonMatrix){
            // Python (NumPy) case
            output = interop.invokeMember(matrix, "__mul__", 42);
        }
        else{
            // R case
            output = interop.invokeMember(matrix, "*", 42);
        }
        return output;
    }
}
```

Listing 3: Implementation of a polyglot node for multiplying R and Numpy matrices by 42. Observe that we have to handle each case separately, even though they are both a kind of matrix

GraalVM exports a message protocol for Truffle languages to communicate with one another, appropriately named the Interoperability Protocol, alternatively INTEROP, for short. INTEROP messages serve to enable language developers to inspect, transform, and trigger behavior from foreign datatypes. To call methods from a foreign datatype, one may use the message `invokeMember` and provide it with a method name and its arguments.

Suppose we meant to implement an AST node receiving a matrix as its input and that outputs the result of multiplying that matrix by 42. Assuming we were expecting NumPy and R matrices as input, we would ultimately write something similar to listing 3. In listing 3, we utilize

InteropLibrary, Truffle's programatic means of sending INTEROP messages, to calls methods from the input matrix. Even though our task is embarrassingly simple, we need to handle a NumPy matrix differently from an R matrix, each requiring a different method name to be called. This is undesirable, because it means we could not support a wider set of matrix datatypes without extending this procedure; in this case it's easy because we simply perform a multiplication, but it would quickly become unwieldy if we were implementing a sequence of LA operations such as in a Morpheus rewrite rule, for instance. What we want is to turn matrices into first-class citizens of Truffle interoperability by giving ourselves access to a uniform means of interacting with them. This is exactly what MATRIXLIB solves.

4.3 The MATRIXLIB Interface

```
@GenerateUncached
abstract static class ExampleNode extends Node {

    protected abstract Object execute(Morpheus receiver)
        throws UnsupportedMessageException;

    @Specialization
    Object doDefault(Morpheus receiver, Object matrix,
        @CachedLibrary("matrix") MatrixLibrary matrixlib)
        throws UnsupportedMessageException {

         Object output = interop.matrixlib(matrix, "scalarMultiplication", 42);
        return output;
    }
}
```

Listing 4: Implementation of a polyglot node for multiplying R and Numpy matrices by 42 via MATRIXLIB, much simpler than the INTEROP alternative

MATRIXLIB is a Truffle Library that simplifies the implementation of Truffle AST nodes operating on foreign datatypes representing matrices. Its key benefit is eliminating the need to know, in advance, the interface details of the foreign input matrix. For an example of this,

compare list 4 with listing 3. MATRIXLIB users are given a singular interface supporting a variety of common matrix operations that foreign input matrices are expected to support. For this to work, a foreign matrix needs to be wrapped with an adapter to MATRIXLIB's interface prior to interacting with MATRIXLIB-enabled Truffle nodes; a process which is done automatically for the end-user.

The MATRIXLIB interface currently supports the following list of messages: `scalarAddition`, `scalarMultiplication`, `scalarExponentiation`, `leftMatrixMultiplication`, `rightMatrixMultiplication`, `rowWiseSum`, `columnWiseSum`, `elementWiseSum`, `splice`, `columnWiseAppend`, `rowWiseAppend`, `transpose`, `getNumColumns`, `getNumCols`, and a special method named `unwrap`. The `unwrap` method removes the adapter from the underlying matrix, and the remaining messages are self-descriptive. In the future, we would like for Truffle developers to support MATIXLIB messages directly, therefore removing the need to wrap the datatype in an adapter when sent to a foreign language context.

4.4 Implementation Summary: Truffle-level

MATRIXLIB is a contribution to the Truffle core API, which means it would be callable from within any Truffle language if compiled with our modified version of Truffle. To our knowledge, it is one of the first usages the of new Library ecosystem and, as a result, our usage of it helped identify and fix bugs in its implementation.

Implementation-wise, the messages of the MATRIXLIB protocol are declared as abstract methods of `MatrixLibrary`, an abstract class extending Truffle's Library object, the superclass of Truffle libraries. Of course, this only serves to specify the valid messages of the protocol, which need to be implemented by concrete classes. In the future, we would like Truffle language developers to provide the implementation of these messages for their matrix datatypes. In the meantime, we provide a solution for dynamically providing the implementation of these messages

```java
@GenerateLibrary
public abstract class MatrixLibrary extends Library {

    public abstract Object scalarAddition(Object receiver, Object scalar)
        throws UnsupportedMessageException;
    public abstract Object scalarMultiplication(Object receiver, Object scalar)
        throws UnsupportedMessageException;
    public abstract Object scalarExponentiation(Object receiver, Object scalar)
        throws UnsupportedMessageException;
    public abstract Object rowWiseAppend(Object receiver, Object tensor)
        throws UnsupportedMessageException;
    public abstract Object columnWiseAppend(Object receiver, Object tensor)
        throws UnsupportedMessageException;
    public abstract Object splice(Object receiver, Integer rowStart, Integer rowEnd,
        Integer colStart, Integer colEnd)
        throws UnsupportedMessageException;
    /* Continues ... */
}
```

Listing 5: Portion of the MatrixLibrary declaration

for user-identified matrices.

Our solution is InteropMatrix, a Truffle-level wrapper over foreign-language objects that uses INTEROP to export MATRIXLIB messages. With the exception of unwrap, all MATRIXLIB messages exported by this class resolve to executing a method from the inner object via INTEROP. In other words, the class assumes that the inner object already exports the MATRIXLIB messages, which may be called via invokeMember, and it simply leads them to type-check at the Truffle-level by re-exporting them directly as MATRIXLIB calls.

4.5 Implementation Summary: Truffle-language level

For InteropMatrix to function correctly, we require its inner object to already export methods corresponding to MATRIXLIB messages. Therefore, to enable MATRIXLIB-managed interoperability, we require matrices to be wrapped in an MATRIXLIB-conforming adapter prior to their usage in a polyglot call; a process done automatically without end-user intervention. We

```java
@ExportLibrary(MatrixLibrary.class)
public class InteropMatrix implements TruffleObject {

    Object adaptee;
    public InteropMatrix(Object adaptee) {
        this.adaptee = adaptee;
    }

    @ExportMessage Object unwrap(@CachedLibrary(limit = "3") InteropLibrary interop) {
        return adaptee;
    }

    @ExportMessage
    public Object crossProduct(@CachedLibrary(limit = "3") InteropLibrary interop)
        throws UnsupportedMessageException {
        Object[] arguments = {};
        return doCallObj("crossProduct", arguments, interop);
    }

    @ExportMessage
    public Object columnWiseAppend(Object tensor,
        @CachedLibrary(limit = "3") InteropLibrary interop,
        @CachedLibrary(limit="10") MatrixLibrary matrixlib)
        throws UnsupportedMessageException {
        Object[] arguments = {matrixlib.unwrap(tensor)};
        return doCallObj("columnWiseAppend", arguments, interop);
    }

    public Object doCallObj(String memberName, Object[] arguments,
    @CachedLibrary(limit = "3") InteropLibrary interop)
        throws UnsupportedMessageException {
        try {
           Object boundFunction = interop.readMember(adaptee, memberName);
           Object tensor = new TensorAdapter(interop.execute(boundFunction, arguments));
           return tensor;
        }
        catch (ArityException | UnknownIdentifierException |
        UnsupportedMessageException | UnsupportedTypeException exception) {
            throw UnsupportedMessageException.create();
        }
        /* Continues ... */

    }
```

Listing 6: Preview of the InteropMatrix declaration

provide such adapters for NumPy and R matrices in their respective languages. The listing below previews `TensorFromNumpy`, the Python-provided adapter for NumPy arrays. The details on when and how to wrap matrices in their Truffle-language-level adapters are discussed in later chapters.

```
class TensorFromNumpy(object):

    def __init__(self,  npArr):
        self.npArr = npArr

    def unwrap(self):
        return self.npArr

    def rightMatrixMultiplication(self, otherArr):
        if isinstance(otherArr, TensorFromNumpy):
            otherArr = otherArr.npArr
        return TensorFromNumpy(np.matmul(self.npArr, otherArr))

    def leftMatrixMultiplication(self, otherArr):
        if isinstance(otherArr, TensorFromNumpy):
            otherArr = otherArr.npArr
        return TensorFromNumpy(np.matmul(otherArr, self.npArr))

    def scalarAddition(self, scalar):
        return TensorFromNumpy(self.npArr + scalar)

    def scalarMultiplication(self, scalar):
        return TensorFromNumpy(self.npArr * scalar)

    # Continues ...
```

Listing 7: Preview of the Python language-level adapter

Chapter 5

MorpheusDSL: Host-agnostic Rewrites

5.1 Overview

Using MATRIXLIB, Truffle AST nodes can manipulate foreign matrices with a singular API. Without it, a developer would need to handle each foreign matrix's interface independently, leading to code duplication and maintainability concerns. Therefore, we can use MATRIXLIB as the key mechanism to enable a single implementation of Morpheus to operate with matrices from many languages, which we did.

We implemented the Morpheus system as a Truffle language called MorpheusDSL. This language serves as an embeddable linear algebra DSL that factorizes the execution of LA algorithms written for Truffle languages. The language implements its AST nodes fully in terms of MATRIXLIB calls, meaning that it can operate with arbitrary foreign matrices as input and, more notably, that it delegates back to its host language's LA system to perform computations: the LA operations are achieved using the hosts *own* AST nodes. Therefore, just like prior Morpheus implementations, the rewrite rules sit on top of its host's semantics. To our knowledge, this is the first Truffle language to function like this.

5.2 An Interoperable NormalizedMatrix

```
@ExportLibrary(InteropLibrary.class)
public final class NormalizedMatrix implements TruffleObject {

    // entity table S, indicators Ks, attributes Rs
    public Object S = null;
    public Object[] Ks = null;
    public Object[] Rs = null;

    // isTransposed and is `S` empty
    boolean T = false;
    boolean Sempty = false;

    // Implement InvokeMember message to
    //enable its methods to be called by
    //foreign languages
    @ExportMessage
    @GenerateUncached
    static class InvokeMember {
        static final protected String build = "build";
        @Specialization(guards = {"member.equals(build)", "arguments.length == 4"})
        static Object doBuild(Morpheus receiver, String member, Object[] arguments,
        @Cached BuildNode node)
        throws UnsupportedMessageException {
            return node.execute(receiver, arguments[0], arguments[1],
            arguments[2], arguments[3]);
        }

        static final protected String scalarAddition = "scalarAddition";
        @Specialization(guards = {"member.equals(scalarAddition)",
        "arguments.length == 1"})
        static Object doScalarAddition(Morpheus receiver, String member,
        Object[] arguments,
        @Cached ScalarAdditionNode node)
        throws UnsupportedMessageException {
            return node.execute(receiver, arguments[0]);
        }

        /* Other messages exported below */
    }

}
```

Listing 8: The essentials to make MorpheusDSL's NormalizedMatrix interoperable: export the InteropLibrary protocol and implement INTEROP's invokeMember

In Morpheus, the NormalizedMatrix is an abstraction over the result of joining matrices; its interface providing more efficient implementations of LA operators. In MorpheusDSL, the NormalizedMatrix is an interoperable entity that can be requested by other languages and, once obtained, its interface also exports more efficient implementations of LA operators. We will describe how other languages request and obtain the NormalizedMatrix later and focus, for now, on describing how the NormalizedMatrix is declared as an interoperable datatype.

The interface exported by the NormalizedMatrix corresponds to the LA operators for which Morpheus provides rewrite rules. First, we export the following self-descriptive interface methods: `scalarAddition`, `scalarExponentiation`, `scalarMultiplication`, `leftMatrixMultiplication`, `rightMatrixMultiplication`, `crossProduct`, `rowWiseSum`, `columnWiseSum`, and `elementWiseSum`. Two extra methods are exported as part of the interface. The first is `build`, which serves to initialize the NormalizedMatrix. Finally, we export a `tranpose` method which simply flips a binary flag in the NormalizedMatrix to signal that subsequent LA operators should execute their transposed variant.

An interoperable Truffle object must implement some INTEROP protocol messages. To specify INTEROP as a supported protocol, we annotate our declaration of the NormalizedMatrix with an `@ExportLibrary(InteropLibrary.class)`. Then, we implement the `invokeMember` method, where we specify which methods are callable from our object, its interface. This is enough to make the NormalizedMatrix a valid interoperable entity with a callable interface.

5.3 Rewrite rules as AST nodes

The Morpheus rewrite rules, and all of the NormalizedMatrix's interface, are implemented as Truffle AST nodes that utilize MATRIXLIB to manipulate, generate, and inspect matrices. As examples, we review the implementation of `scalarAddition`, `rightMatrixMultiplication`, and the `build` method.

5.3.1 The ScalarAddition Node

```
@GenerateUncached
abstract static class ScalarAdditionNode extends Node {

    protected abstract Object execute(NormalizedMatrix receiver, Object scalar)
    throws UnsupportedMessageException;

    @Specialization(limit = "3", guards="!receiver.Sempty")
    Object doDefault(NormalizedMatrix receiver, Object scalar,
                     @CachedLibrary("receiver.S") MatrixLibrary matrixlibS,
                     @CachedLibrary(limit = "3") MatrixrLibrary matrixlibGen)
                     throws UnsupportedMessageException {

        int size = receiver.Rs.length;
        Object[] newRs = new Object[size];
        for(int i = 0; i < size; i++) {
            newRs[i] = matrixlibGen.scalarAddition(receiver.Rs[i], scalar);
        }
        Object newS = matrixlibS.scalarAddition(receiver.S, scalar);
        return createCopy(newS, receiver.Ks, newRs, receiver.T, receiver.Sempty);
    }
```

Listing 9: Implementation of the scalarAddition method

According to equation 2.2, element-wise scalar operators should be implemented by applying the corresponding operator, element-wise, to the entity matrix S and all attribute matrices R_i. All element-wise scalar operators are implemented very similarly, so the implementation of element-wise addition in listing 9 should be representative of the whole class.

5.3.2 The Build Node

Consider listing 10. Our NormalizedMatrix expects entity matrix S, an array of indicator matrices Ks, and an array of attribute tables Rs to build itself. The constructor merely wraps matrix objects in these three parameters with InteropMatrix and stores them as member variables, waiting for later use. As said when describing MATRIXLIB, it is necessary to wrap foreign inputs in Truffle-level InteropMatrix wrappers for MATRIXLIB to typecheck.

```
@GenerateUncached
abstract static class BuildNode extends Node {

    protected abstract Object execute(NormalizedMatrix receiver, Object S, Object Ks,
    Object Rs, Object Sempty);

    @Specialization
    Object doDefault(NormalizedMatrix receiver, Object S, Object Ks,
    Object Rs, Object Sempty,
        @CachedLibrary(limit = "10") InteropLibrary interop) {
        receiver.S = new InteropMatrix(S);
        try {
            //TODO: casting to int is unsafe here! this returns longs
            int sizeKs = interop.getArraySize(Ks);
            int sizeRs = interop.getArraySize(Rs);
            boolean SemptyBool = (boolean) interop.asBoolean(Sempty);

            receiver.Ks = new Object[sizeKs];
            receiver.Rs = new Object[sizeKs];

            Object currK = null;
            Object currR = null;
            for(int i = 0; i < sizeKs; i ++){
                currK = interop.readArrayElement(Ks,i);
                currR = interop.readArrayElement(Rs,i);
                receiver.Ks[i] = new InteropMatrix(currK);
                receiver.Rs[i] = new InteropMatrix(currR);
            }
            receiver.Sempty = SemptyBool;
        }

        catch (Exception e) {
        System.out.println(e.toString());
        }
        return receiver;
    }
}
```

Listing 10: Implementation of the build method

5.3.3 The RightMatrixMultiplication Node

The formal description of the rightMatrixMultiplication rewrite may be found in equation 2.4. Note the following key steps in listing 11: first, that the foreign input matrix is wrapped within the Truffle-level InteropMatrix and, second, that the result is ”unwrapped” prior to being

```
@GenerateUncached
abstract static class RightMatrixMultiplicationNode extends Node {

    protected abstract Object execute(NormalizedMatrix receiver, Object vector)
    throws UnsupportedMessageException;

    @Specialization(limit = "3", guards = {"!receiver.T", "!receiver.Sempty"})
    Object doDefault(NormalizedMatrix receiver, Object vector,
                     @CachedLibrary("receiver.S") TensorLibrary tensorlibS,
                     @CachedLibrary(limit = "3") TensorLibrary tensorlibGen)
                     throws UnsupportedMessageException {
        Object vectorAdapter = new InteropMatrix(vector);
        Object leftmostColumns = tensorlibS.leftMatrixMultiplication(receiver.S,
        vectorAdapter);
        Object result = leftmostColumns;
        Object vecByK = null;
        Object rightmostColumns = null;
        int size = receiver.Rs.length;
        for(int i = 0; i < size; i++) {
            vecByK = tensorlibGen.leftMatrixMultiplication(receiver.Ks[i],
            vectorAdapter);
            rightmostColumns = tensorlibGen.leftMatrixMultiplication(receiver.Rs[i],
            vecByK);
            result = tensorlibGen.columnWiseAppend(result, rightmostColumns);
        }
        Object resultUnwrapped = tensorlibGen.unwrap(result);
        return resultUnwrapped;
    }
```

Listing 11: Implementation of the rightMatrixMultiplication method

returned. These two steps are always necessary when a foreign matrix is taken in as input as they are required for MATRIXLIB to typecheck. Remember from the build method, that the NormalizedMatrix already wrapped S, every K_i, and every R_i with InteropMatrix as well.

5.3.4 How the transposed-version rewrites are selected

Our Morpheus rewrite rule implementations are guarded, meaning that we specificy a set of boolean checks that should hold true for the corresponding AST node to execute; a check automatically enforced by the compiler. This means that we can provide alternative AST node implementations for different guard combinations and it is via this mechanism that we implement

the Morpheus rewrite rules for when the NormalizedMatrix is transposed. In other words, the Morpheus rewrite rules whose definition changes if the NormalizedMatrix is transposed will have alternative AST node implementations guarded by whether or not the NormalizedMatrix is transposed. These guards can be seen in listing 11 and listing 9.

5.3.5 Rewrites for when S is empty

When processing real-world datasets, it is possible that our entity table S will end up empty. This occurs when the entity table contained only foreign keys instead of also containing some metadata columns of their own; which occurs in some of our own datasets. Please refer to table 7.1 for a listing of which datasets exhibit this behaviour.

When S ends up empty, we need to adjust our rewrite rules accordingly. Just like for the transposed versions of the rewrites, we also provide alternative implementations of the rewrite rules for when S is empty and guard them by a size check on S.

Chapter 6

Embedding MorpheusDSL

6.1 Overview

With MorpheusDSL, Truffle languages gain access to the NormalizedMatrix, an interoperable matrix abstraction that enables the factorized execution of linear algebra algorithms. However, the default interface exported by MorpheusDSL's NormalizedMatrix may not correspond to the expected matrix interface for some linear algebra system. For example, in NumPy, scalar multiplication is done by calling the `__mul__` method but, in TRINITY, that operation corresponds to the method name `scalarMultiplication`. With this state of affairs, pre-existing linear algebra algorithms would not be able to operate on a NormalizedMatrix without first adjusting their matrix method calls to account for this alternative matrix interface! We would like for the NormalizedMatrix to *feel* like a NumPy matrix and, more generally, to have the appropriate interface for whichever linear algebra system that we aim to optimize.

To circumvent this problem, we turn again to adding a layer of indirection. We provide NormalizedMatrix adapters to wrap the NormalizedMatrix in Python and R, exporting the appropriate matrix interface in each. This is very easy to do because it corresponds to little more than merely establishing a mapping between interfaces and yet, it is sufficient to enable

pre-existing algorithms to operate on the NormalizedMatrix *as if* they were dealing with their native matrix datatype. This is ideal because it means that enabling Morpheus-style optimizations in some new linear algebra system should be as easy as writing just another matrix adapter, thus simply *embedding* and re-using the pre-existing implementation of Morpheus logic.

6.2 Mapping LA operations

Table 6.1: Rough mapping between Morpheus operators signatures and their names in Numpy and R

Morpheus call-signature	**R prefixed-signature**	**Python call-signature**
`scalarAddition(receiver, numeric)`	`"+"(numeric, receiver)` `"+"(receiver, numeric)`	`__sum__(matrix, numeric)` `__sum__(matrix, numeric)`
`scalarMultiplication(receiver, numeric)`	`"*"(numeric, receiver)` `"*"(receiver, numeric)`	`__mul__(matrix, numeric)` `__mul__(matrix, numeric)`
`scalarExponentiation(receiver, numeric)`	`"^"(numeric, matrix)` `""^"(matrix, numeric)`	`__sum__(matrix, numeric)` `__sum__(matrix, numeric)`
`leftMatrixMultiplication( receiver, matrix)`	`%*%(matrix,`	`__mul__(matrix, matrix)`
`rightMatrixMultiplication( receiver, matrix)`	`%*%(matrix, matrix)`	`__mul__(matrix, matrix)`
`rowWiseSum(receiver)`	`rowSums(receiver)`	`numpy.sum(receiver, axis=1)`
`colWiseSum(receiver)`	`colSums(receiver)`	`numpy.sum(receiver, axis=0)`
`elementWiseSum(receiver)`	`sum(matrix)`	`numpy.sum(receiver)`

Interface adapters are meant to be lightweight wrappers over an object, re-exporting its interface to conform to alternative call signatures expected by some system, a pre-existing

algorithm in our case. We begin by identifying the mapping between the call signatures of the Morpheus operators to the operators and method names of R's Matrix and Python's NumPy.

Table 6.1 describes a rough mapping between the rewrite rules of Morpheus, as made available by MorpheusDSL's NormalizedMatrix, and their corresponding method names in NumPy and R. We say that this is a rough mapping because we have simplified the call signatures across languages to all appear in prefix notation and have the same argument names, for ease of comparison. Observe that, in NumPy, the `__mul__` method handles Morpheus' `leftMatrixMultiplication`, `rightMatrixMultiplication`, and `scalarMultiplication` operations. This simply means that our wrapper will need to inspect the types of the arguments of this method in order to determine which NormalizedMatrix method to call. We will exemplify that later in this chapter.

Additionally, the interface of some Matrix datatype is likely to contain more operations that just those overloaded by Morpheus, For instance, matrix slicing is not a supported Morpheus operator, for performance reasons, but it is available in the matrix interface of Numpy and R. In the general case, we encourage unsupported operators to throw exceptions as means of signaling that the Morpheus system is ill-prepared to handle those use-cases. Nonetheless, our set of supported LA operators is already big enough to support a large class of machine learning algorithms.

6.3 Implementation Summary: The constructor

The constructor expects three arguments: an entity matrix S, an array of attribute matrices Rs, and the corresponding array of indicator matrices Ks. In the future, we may instead overload the host's matrix-join operator and automatically construct a normalized matrix that way. We currently avoid doing that so that end-users may explicitely decide whether or not they want to use a normalized matrix.

The construction procedure is simple. We begin by wrapping the argument matrices with host-level MatrixAdapters, to enable interoperability, and pass them as arguments to a

```r
# Adapter, stores NM as member variable
NormalizedMatrix <- setClass(
   "NormalizedMatrix",
   slot = c(morpheus = "ANY")
)

# Adapter constructor
asNormalizedMatrix <- function(S, Ks, Rs) {

    # Wrap tensors in adapters
    tensorS <- TensorFromMatrix(matrix=S)
    tensorKs <- lapply(Ks, function(x){TensorFromMatrix(matrix=x)})
    tensorRs <- lapply(Rs, function(x){TensorFromMatrix(matrix=x)})

    # Obtain NM constructor, execute it, store it in adapter object,
    # return adapter
    morpheusBuilder <- eval.polyglot("MorpheusDSL", "build")
    # TODO: check is S is empty
    Sempty <- FALSE
    if(nrow(S)*ncol(S) == 0){
        Sempty <- TRUE
    }
    morpheus <- morpheusBuilder@build(tensorS, tensorKs, tensorRs, Sempty)
    normMatrix <- NormalizedMatrix(morpheus=morpheus)
    return(normMatrix)
}
```

Listing 12: The constructor for Normalized Matrix adapter, in R

NormalizedMatrix `build` method, which can be invoked via that the host's `polyglot-eval` function. Finally, after calling the `build` method, the adapter stores resulting normalizedMatrix as a member variable; waiting to call methods from it when LA operators are invoked.

6.4 Implementation Summary: Deferring to MorpheusDSL

Writing these classes is extremely simple. When implementing the element-wise scalar operators, the adapter needs only to call upon the corresponding method from its inner NormalizedMatrix object. This returns another NormalizedMatrix from MorpheusDSL, which is then wrapped by its own adapter and returned.

```r
setMethod("+", c("numeric", "NormalizedMatrix"), function(e1, e2) {
    result <- e2@morpheus@scalarAddition(e1)
    newNormalizedMatrix <- NormalizedMatrix(morpheus=result)
    return(newNormalizedMatrix)
})

setMethod("+", c("NormalizedMatrix", "numeric"), function(e1, e2) {
    result <- e1@morpheus@scalarAddition(e2)
    newNormalizedMatrix <- NormalizedMatrix(morpheus=result)
    return(newNormalizedMatrix)
})

setMethod("-", c("numeric", "NormalizedMatrix"), function(e1, e2) {
    result <- e2@morpheus@scalarAddition(e1)
    newNormalizedMatrix <- NormalizedMatrix(morpheus=result)
    return(newNormalizedMatrix)
})

setMethod("-", c("NormalizedMatrix", "numeric"), function(e1, e2) {
    result <- e1@morpheus@scalarAddition(e2)
    newNormalizedMatrix <- NormalizedMatrix(morpheus=result)
    return(newNormalizedMatrix)
})

setMethod("*", c("numeric", "NormalizedMatrix"), function(e1, e2) {
    result <- e2@morpheus@scalarMultiplication(e1)
    newNormalizedMatrix <- NormalizedMatrix(morpheus=result)
    return(newNormalizedMatrix)
})

setMethod("*", c("NormalizedMatrix", "numeric"), function(e1, e2) {
    result <- e1@morpheus@scalarMultiplication(e2)
    newNormalizedMatrix <- NormalizedMatrix(morpheus=result)
    return(newNormalizedMatrix)
})
```

Listing 13: Scalar methods exported by the Normalized Matrix adapter, in R

In other cases, some lightweight processing of inputs and outputs is required. For instance, when passed a matrix argument, we need to wrap them in a language-level adapter conforming to MATRIXLIB. The reason why no adapter was needed in the element-wise scalar operators is that interop understands and transforms numeric types directly, and our element-wise scalar operators all take in numbers as arguments. The other case is when a Normalized Matrix operation returns

either a matrix or a number. Since the implementation of the NormalizedMatrix is language-agnostic, it can only receive and return foreign-language-native objects by wrapping them in MATRIXLIB-adapters. As a result, these need to be removed before returning values. These two processing steps, to apply and remove MATRIXLIB-adapters, are the only extra responsabilities of the NormalizedMatrix

```
setMethod("%*%", c("NormalizedMatrix", "ANY"), function(x, y) {
    tensorArg <- TensorFromMatrix(y)
    result <- x@morpheus@leftMatrixMultiplication(tensorArg)
    return(removeMatrixAdapter(result))
})

setMethod("sum", c("NormalizedMatrix"), function(x) {
    result <- x@morpheus@elementWiseSum()
    return(as.vector(removeMatrixAdapter(result)))
})
```

Listing 14: The pre- and post- processing needed for leftMatrixMultiplication and element-wise sum

6.5 How TRINITY optimizes polyglot programs

Since end-users running programs in GraalVM can mix multiple languages within the same program, we would like for TRINITY to optimize polyglot programs as well. Consider the example of executing a NumPy algorithm with matrices represented in R. MorpheusDSL is able to optimize this because it interacts with input matrices using the same API, regardless of the input's origin. As a result, so long as the input matrices are from the same language, we can operate on them and have them interact with one another using MATRIXLIB. Then, since the NormalizedMatrix is a foreign object wrapped to fit some host matrix interface, the wrapper will take care of translating the NumPy LA operators into Morpheus-recognized method calls, which MATRIXLIB resolves to interop calls to manipulate the R matrices. With this wrapper-managed translation, a NumPy algorithm dictates what operations to perform over R matrices.

Chapter 7

Experimental Evaluation

We present an empirical evaluation of TRINITY's performance using synthetic and real-world datasets. We consider the performance of materializing the join as the baseline. Additionally, we compare TRINITY against the pre-existing Morpheus implementation for R to explore if our *host-agnostic* implementation suffers from excesive runtime overheads from all the indirection. More concretely, we aim to answer the following two questions:

1. **RQ1:** How do TRINITY rewrites scale with join-induced redundacy?
2. **RQ2:** How does TRINITY compare to prior Morpheus implementations?

Experiment Computing Setup. All the experiments were run a machine with 47 Intel Xeon CPU E5-2690 v3 2.60GHz 12-cores CPUs, 512 GB of RAM, and over 7TB of disk available, running on Oracle Linux Server 7.3 as the OS. We use OpenJDK version 1.8.0_151 and `a3a4c35a38a0128eeba1bef863c00f4a620356d0`, `6ba17f6cb2a3bfa44eb0d7237c7eec82d57fffbb`, and `3d5bc06a753cd65cdcde2b5ed9c89bd68a277bc7` as the commit hashes for the GraalVM [3], FastR [1], and GraalPython [2] repositories respectively.

Platform Limitations. We have completed adapters for R and NumPy, and are actively working to support more hosts. Unfortunately, the Python implementation for GraalVM is still

in its early stages and does not currently support a native sparse matrix datatype. Since a sparse indicator matrix K is key for the Normalized Matrix, we cannot evaluate TRINITY on NumPy without quickly running out of memory. As a result, we limit our evaluation to TRINITY's performance in R. Nonetheless, we will test the NumPy integration with a polyglot-performance test, such that the algorithms are defined in NumPy but execute with R matrices as input.

Synthetic Datasets. To measure the impact of join-induced dataset redundancy, we generate synthetic 2-table datasets for a range of dataset dimensions. We fix $n_S = 10^5$ and $d_S = 20$, and vary TR, the *tuple ratio* $\frac{n_S}{n_R}$, and FR, the *feature ratio* $\frac{d_R}{d_R}$, to control for redundancy. Unless otherwise specified, we increase TR and FR in increments of one, within ranges $[1,20]$ and $[1,5]$ respectively.

Real-word Datasets. We also re-use 6 of the real-world normalized datasets from the original Morpheus paper. As per standard practice, the datasets are pre-processed by one-hot-encoding all categorical features, dropping primary keys from the entity table, and by *whitening* (subtracting the mean and dividing by the standard deviation) all numeric features. Finally, all these datasets contain a prediction target column Y which is separated from the original dataset and fed appropriately to the algorithms. The Y column is *binarized* for logistic regression to 0/1 values but kept intact for all other algorithms. Relevant dataset statistics may be found in 7.1.

Table 7.1: Real-world dataset dimensionalities

Dataset	(n_S, d_S)	$\#\ tables$	(n_{Ri}, d_{Ri})
Expedia	$(942142, 27)$	2	$(11939, 12013)$ (37021,40242)
Movies	$(1000209, 0)$	2	$(6040, 9509)$ (3706,3839)
Yelp	$(215879, 0)$	2	$(11535, 11706)$ (43873,43900)
LastFM	$(343747, 0)$	2	$(4099, 5019)$ (50000,50233)
Books	$(253120, 0)$	2	$(27876, 28022)$ (49972,53641)
Flights	$(66548, 20)$	3	$(540, 718)$ (3167,6464) (3170,6467)

7.1 Operator-level Results in FastR

We begin by assessing TRINITY's performance on individual LA operators executing over synthetic datasets. As shorthand, we sometimes refer to right matrix multiplication and left matrix multiplication as RMM and LMM respectively.

Experiment Setup. For these experiments, we recorded the duration, in milliseconds, of 30 consecutive executions for selected LA operators. We consider the first five as warm-up and report the average of the remaining 25. For comparison, we ran this experiment with three different approaches. The first is the materialized approach, our primary baseline, where we materialize the join of tables. The second is the TRINITY approach. Finally, we also ran these experiments with MorpheusR, the original host-aware implementation of Morpheus introduced in [12]. For a fair comparison, all these experiments ran on the GraalVM runtime instead of GNU-R.

7.1.1 Discretized speed-ups over the materialized approach

We first evaluate TRINITY's performance compared to the materialized approach on six representative LA operators: scalar addition, left-matrix multiplication, right-matrix multiplication, row-wise sum, column-wise sum, and element-wise sum. We report TRINITY's relative speed-up when discretized over four classes: a fractional speed-up (a slowdown), a speed-up larger than 1x but fewer than 2x, a speed-up larger than 2x but fewer than 3x, and a speed-up larger 3x. In increasing order of relative speed-up, these classes are presented visually as: a black circle, a green cross, a blue square, and a red diamond. In theory, we expect the speed-ups to grow alongside the redundancy ratios; they did.

Element-wise Scalar Operations. Figure 7.1 presents the speed-ups for scalar addition as representative for the class of element-wise scalar operations. In general, we observe that TRINITY achieves higher speed-ups over its materialized alternative as the feature- and tuple-ratios increase, which is expected. However, prior work on Morpheus has shown that the speed-

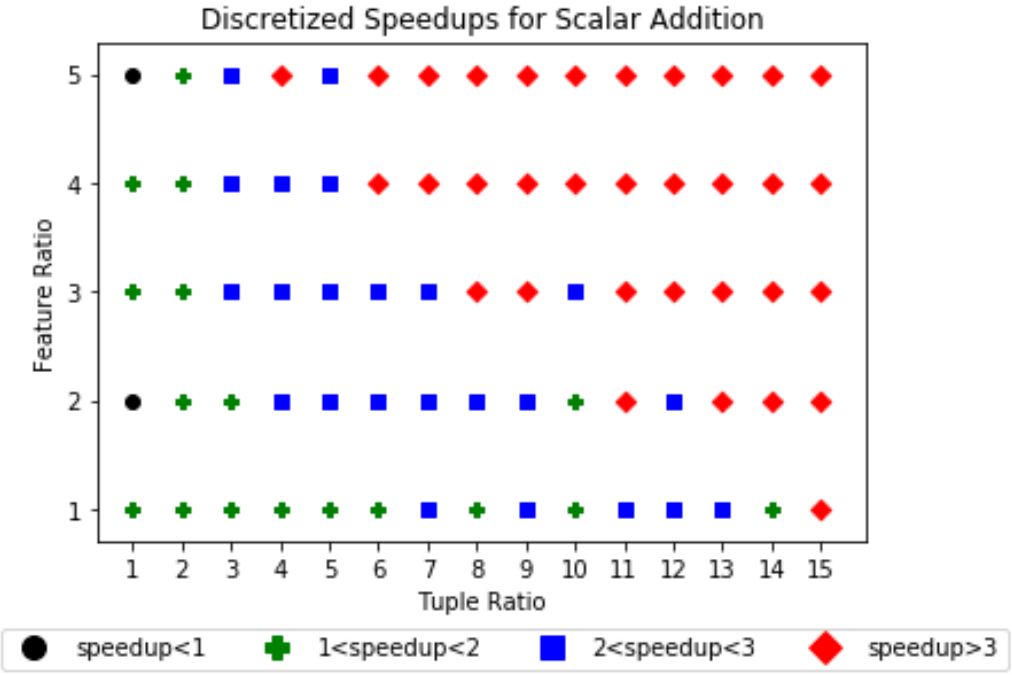
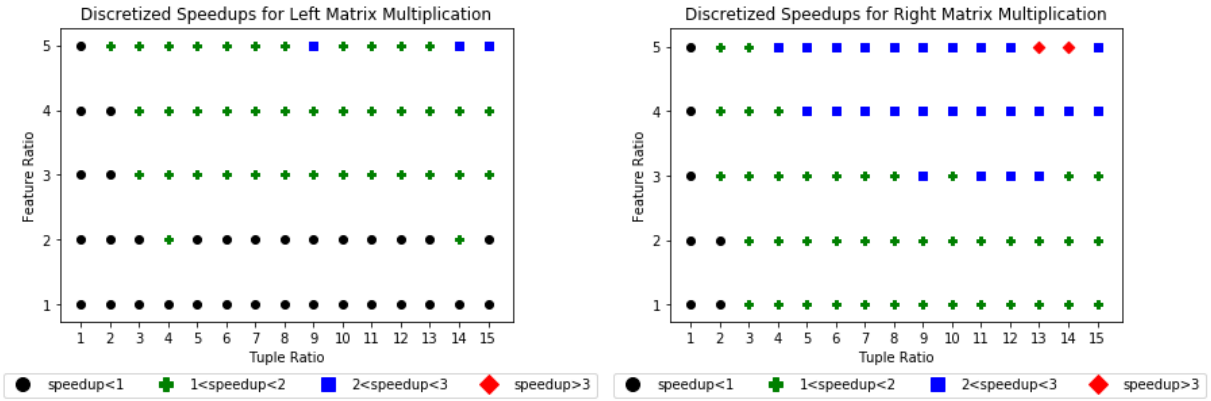


Figure 7.1: Discretized Speedups for Scalar Addition

ups should increase monotonically as the ratios increase, which is not the case for us. For instance, scalar addition at TR-FR configuration (1,1) achieves a speed-up larger than 1x but smaller than 2x, for which we should not have a slowdown at TR-FR configuration (1,2) and especially not in (1,5). We will soon see that these *performance blips* become more pronounced in other operations.

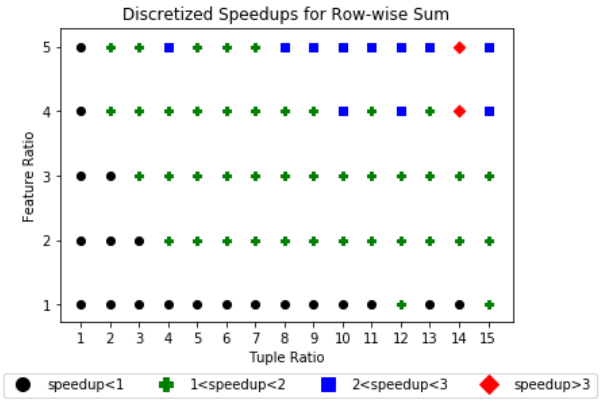


(a) Discretized Speedups for LMM

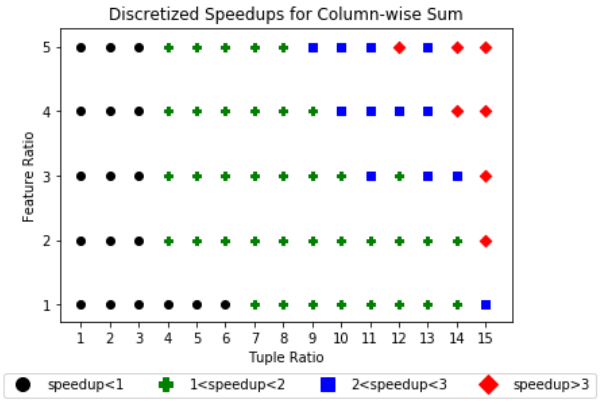
(b) Discretized Speedups for RMM

Figure 7.2: Matrix Multiplication Speed-ups

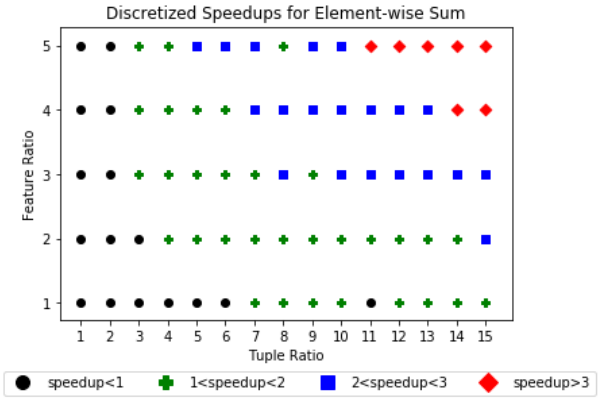
Matrix Multiplications. Figure 7.7b shows the speed-ups for the matrix multiplication operations. Clearly, TRINITY's LMM operator requires a larger redundancy before achieving speed-ups over the baseline. Furthermore, its speed-ups are minimal as they are hardly larger than 2x. In comparison, the RMM operation does appear to exhibit speed-ups at low FR-TR configurations while also achieving speed-ups larger than 2x with more frequency. As it is apparent, these two also exhibit the aforementioned *performance blips*.



(a) Discretized Speedups for Row-wise sum



(b) Discretized Speedups for Column-wise sum



(c) Discretized Speedups for Element-wise sum

Figure 7.3: Aggregation Operations Speed-ups

Aggregating Operations. Figure 7.8 displays the discretized speed-ups for the aggregating operations. In terms of performance, element-wise sum achieves the best speed-ups, followed by column-wise sum, with row-wise last. As before, they also exhibit *performance blips*.

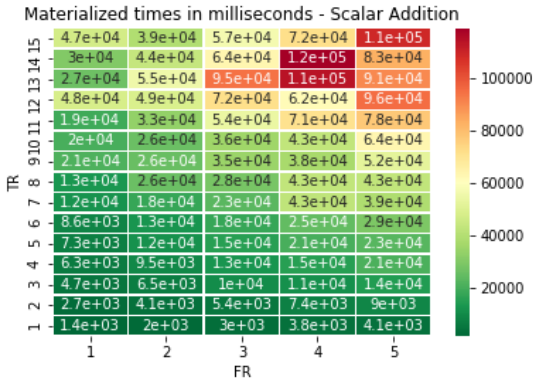
Summarizing remarks. In general, we observe that TRINITY achieves greater speed-ups over the materialized approach as the redundancy ratios grow, which is expected and desirable. However, we are worried about the sporadic drops in performance, which we're calling *performance blips*, because they go against our expectation that larger redundancy should *always* lead to greater speed-ups. Therefore, we have an incentive to look at the individual execution times for the materialized approach and TRINITY in hopes to find an explanation for what is behind these performance drops.

7.1.2 Inspecting the real execution times

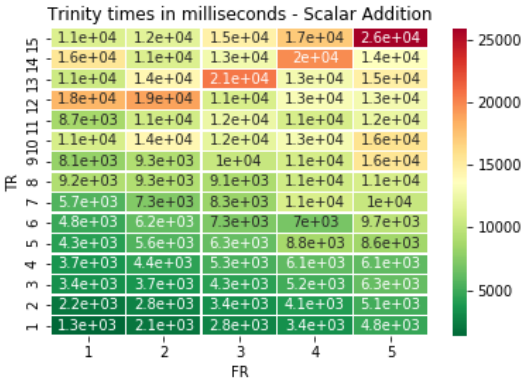
To get a sense of what may be causing the *performance blips*, we inspect the average execution times, in milliseconds, for the materialized approach and for TRINITY as TR and FR grow. We present the data as heatmaps ranging over TR and FR where, like before, we expect the execution times to grow monotonically with FR-TR; after all, larger matrices should take longer to process. This is not what we see. Instead, we report that *sometimes* a smaller matrix takes longer to process, on average, than a larger one.

Here, we focus on the execution times for scalar addition, left matrix multiplication, and element-wise sum, because they exhibit the most interesting behaviour. The remaining plots can be found in the Appendix.

Scalar Addition. Figure 7.4 displays the real execution times, in milliseconds, for TRINITY and the materialized approach for the scalar addition operation. The materialized execution times appear to mostly grow monotonically with FR-TR configurations, but with a few notable exceptions. For instance, we see that the average execution time for FR-TR (3,13) is larger than that of (3,14); the same happens with (4,14) compared to (4,15). Similarly, the heatmap for TRINITY exhibits *blips* as well. Visually, these can be identified as the areas of the heatmap where the a cell's color drastically changes to a higher color in the gradient but then falls back to a lower gradient color in the next cell.

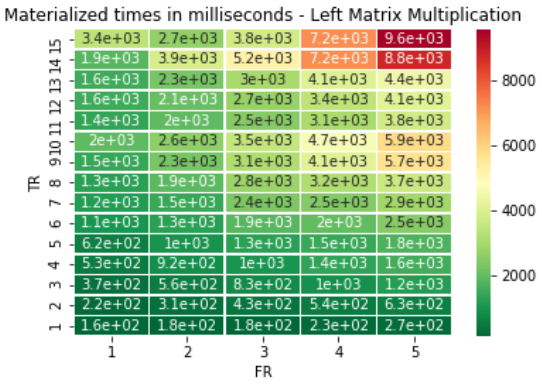


(a) Baseline Scalar Addition Execution Times in ms

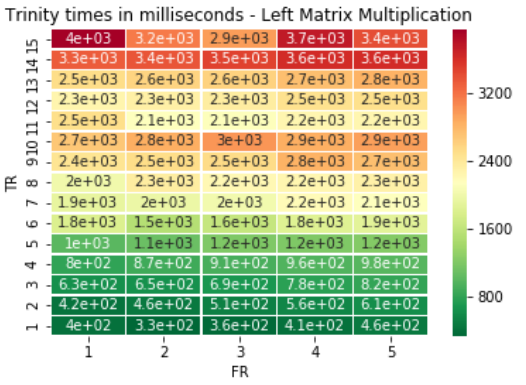


(b) Trinity Scalar Addition Execution Times in ms

Figure 7.4: Real Execution Times for Scalar Addition



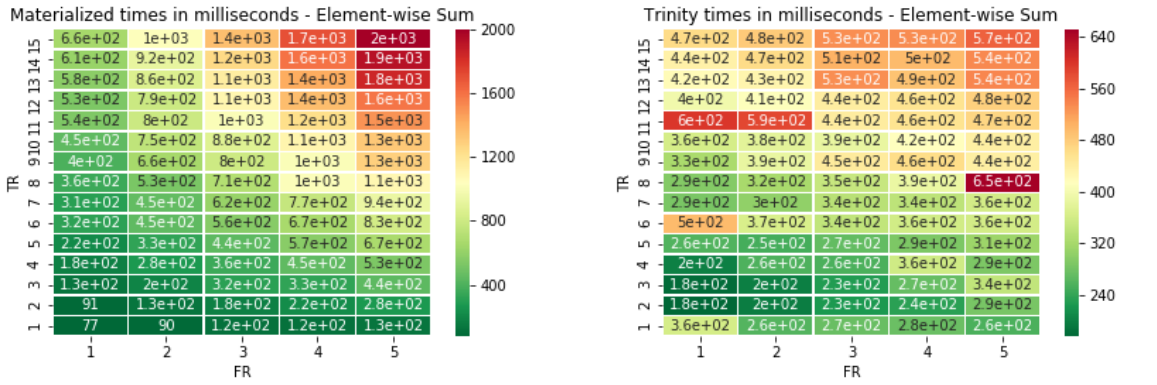
(a) Baseline LMM Execution Times in ms



(b) Trinity LMM Execution Times in ms

Figure 7.5: Real Execution Times for Left Matrix Multiplication

Left Matrix Multiplication. Figure 7.5 shows the average execution times for left matrix multiplication. We also observe some *blips*, most notably in TRINITY's results where the average processing time for FR-TR (1,15) is larger than (5,15). However, like before and as we'll see in all our experiments, larger TR-FR configurations to appear to correlate with larger average processing times.



(a) Baseline Row-wise Sum Execution Times in ms

(b) Trinity Row-wise Sum Execution Times in ms

Figure 7.6: Real Execution Times for Element-wise Sum

Row-wise Sum. The row-wise sum real execution times are displayed in Figure 7.6. Compared to Figure 7.4 and 7.5, *blips* can be observed more frequently here in the TRINITY results. In particular, we can quickly visually identify that FR-TR configurations (1,11), (2,11), and (5,8) to exhibit abnormally large processing times.

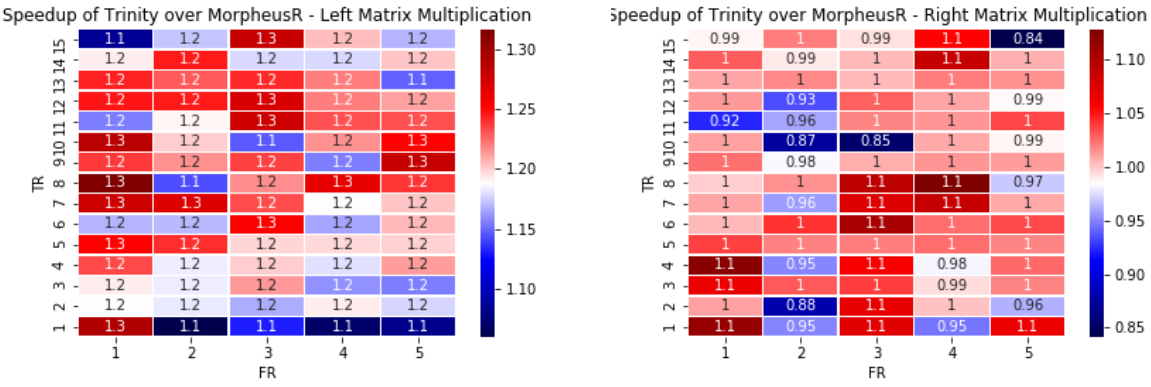
Summarizing Remarks. Unexpectedly, we see that LA operators sometimes took longer to execute over smaller matrices than over larger ones. In addition, when this occurs, it occurs without discernible pattern: the blips don't occur in the same FR-TR ratios. This suggest that perhaps there's some VM-level work occurring non-deterministically that's causing some experiments to take abnormally long to complete. We speculate on this next.

7.1.3 What could be causing these *performance blips*?

Looking at the heatmaps in the prior section, it would appear these aberrant drops of performance appear mostly in high FR-TR ratios, which also corresponds with more memory consumption. This leads us to believe that the drops of performance may be caused, in part, to garbage collection (GC) pausing the execution to free-up heap usage. This is something we are still actively investigating so we cannot provide a conclusive answer, for now.

7.1.4 TRINITY speed-ups relative to MorpheusR

We wanted to know how TRINTY's *host-agnostic* implementation of Morpheus compares to a pre-existing *host-aware* implementation, at the operator level. Therefore, we now compare MorpheusR, the prior implementation of Morpheus for R, with TRINITY on R. To do so, we display the relative speed-ups of TRINITY's average execution time over MorpheusR when running on synthetic datasets with different TR-FR configurations. Here, we focus on the matrix multiplications and aggregating operators, because they are the most expensive and we would expect them to showcase the most overhead.



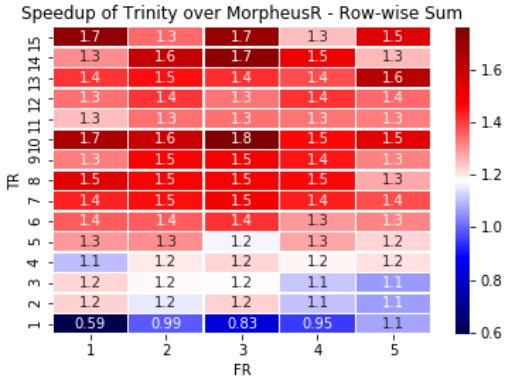
(a) Speedups for LMM

(b) Speedups for RMM

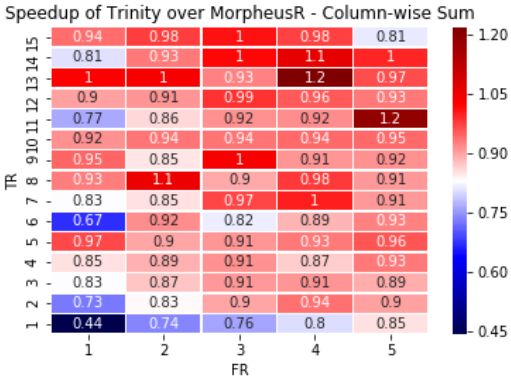
Figure 7.7: Matrix Multiplication Speed-ups

Matrix Multiplications. For RMM, it appears that TRINITY is, on average, between 20

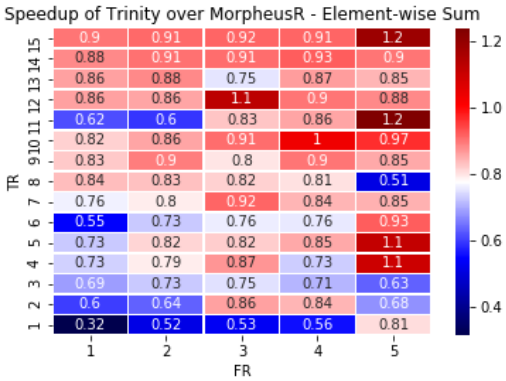
slower and 10 percent faster than MorpheusR. This suggests that TRINITY's use of indirection does not severely impact the performance of the rewrite rules. For LMM in particular, TRINITY appears to be often faster than Morpheus, frequently up to 20 percent faster. It's possible that MorpheusR implemented LMM with optimizations specifically tuned for GNU-R that are not as efficient for GraalVM's R implementation, therefore leading to TRINITY being faster in LMM.



(a) Speedups for Row-wise sum



(b) Speedups for Column-wise sum



(c) Speedups for Element-wise sum

Figure 7.8: Aggregation Operations Speed-ups

Aggregating Operations. For the aggregating operators, we see a similar trend as with the matrix multiplications: TRINITY's performance is comparable to that of MorpheusR. Especially at higher FR-TR ratios, TRINITY appears to be rarely significantly faster or slower than MorpheusR. At lower FR-TR ratios though, TRINITY appears to suffer some performance

penalties, most notably in element-wise sum.

Summarizing Remarks. It appears that TRINITY's use of indirection does not severely impact it's average performance compared to MorpheusR. While we see lower performance for the aggregator operators at low FR-TR ratios, TRINITY's performance often matches and sometimes surpasses that of MorpheusR. Therefore, we are confident that our *host-agnostic* solution is a viable means of optimizing languages with Morpheus rewrites.

7.2 Algorithm-level Results in FastR

7.2.1 Training time summary statistics

We compare how fast TRINITY trains ML models in comparison to the materialized approach and MorpheusR. We compare these three approaches using our selection of real-world datasets for the following three algorithms: Linear Regression (*LinReg*), Logistic Regression (*LogReg*), and K-Means Clustering (*KMeans*).

Experiment Setup. For these experiments, we recorded the duration, in milliseconds, of 30 consecutive training scripts for each algorithm. We consider the first five as warm-up and report the average of the remaining 25. We ran this experiment with three different approaches, like before: the materialized approach, TRINITY, and MorpheusR. Once more, these were all executed within the GraalVM runtime. Finally, we run each training loop for 20 iterations, and the number of centroids in KMeans is 10.

Linear Regression. Table 7.2 presents the results for *LinReg*. We see that, whenever MorpheusR achieves a speed-up, so does TRINITY; meaning that both TRINITY and MorpheusR achieved speed-ups for all datasets except for Flights, where they were both around 10 percent slower than the materialized approach. It would also appear that, in most cases, MorpheusR achieves larger speed-ups than TRINITY, close to 2 times larger.

Table 7.2: FastR Linear Regression Results. Linear Regression mean runtime (in ms) for the materialized baseline (**M**) and its standard deviation ($\mathbf{Sp}_b$), TRINITY's speed-ups relative to **M** ($\mathbf{Sp}_t$) and its runtime standard deviation($\mathbf{STD}_t$), and MorpheusR's speed-ups relative to **M** ($\mathbf{Sp}_m$) and its runtime standard deviation($\mathbf{STD}_t$). Y, M, E, L, B, and F correspond to the Yelp, Movies, Expedia, LastFM, Books, and Flights datasets respectively.

Dataset	M	$\mathbf{Sp}_t$	$\mathbf{Sp}_m$	$\mathbf{STD}_b$	$\mathbf{STD}_t$	$\mathbf{STD}_m$
Y	21272.28	10.23	17.7	641.25	48.03	75.13
F	1422.95	0.9	0.98	65.69	35.73	30.87
E	77435.36	3.15	2.84	1036.75	100.73	162.48
B	3429.31	1.46	2.46	149.76	50.93	38.31
L	9085.38	3.22	5.7	477.8	39.65	69.29
M	69986.09	12.1	19.01	525.11	58.35	87.04

Table 7.3: FastR Logistic Regression Results. Logistic Regression mean runtime (in ms) for the materialized baseline (**M**) and its standard deviation ($\mathbf{Sp}_b$), TRINITY's speed-ups relative to **M** ($\mathbf{Sp}_t$) and its runtime standard deviation($\mathbf{STD}_t$), and MorpheusR's speed-ups relative to **M** ($\mathbf{Sp}_m$) and its runtime standard deviation($\mathbf{STD}_t$). Y, M, E, L, B, and F correspond to the Yelp, Movies, Expedia, LastFM, Books, and Flights datasets respectively.

Dataset	M	$\mathbf{Sp}_t$	$\mathbf{Sp}_m$	$\mathbf{STD}_b$	$\mathbf{STD}_t$	$\mathbf{STD}_m$
Y	1078.46	7.41	12.34	22.57	19.96	2.1
F	84.39	0.76	0.88	2.84	31.72	3.79
E	4190.13	2.82	2.64	32.44	56.09	68.79
B	210.84	1.33	2.0	19.94	3.02	26.08
L	514.26	2.79	4.38	29.0	22.38	4.84
M	3882.91	8.98	13.49	62.79	19.84	2.7

Logistic Regression. Table 7.3 presents the results for *LogReg*. Here see similar results as before, MorpheusR and TRINITY achieve speed-ups for all datasets but Flights, and MorpheusR tends to be faster than TRINITY.

Table 7.4: FastR KMeans Clustering Results. KMeans Clustering mean runtime (in ms) for the materialized baseline (**M**) and its standard deviation ($\mathbf{Sp}_b$), TRINITY's speed-ups relative to **M** ($\mathbf{Sp}_t$) and its runtime standard deviation($\mathbf{STD}_t$), and MorpheusR's speed-ups relative to **M** ($\mathbf{Sp}_m$) and its runtime standard deviation($\mathbf{STD}_t$). M, E, and F correspond to the Movies, and Flights datasets respectively.

Dataset	M	$\mathbf{Sp}_t$	$\mathbf{Sp}_m$	$\mathbf{STD}_b$	$\mathbf{STD}_t$	$\mathbf{STD}_m$
F	3715.87	0.65	0.54	63.07	30.46	112.0
E	121118.78	0.54	0.99	537.21	28147.13	1863.36
M	111979.28	2.35	2.47	951.57	691.64	584.45

K-Means Clustering. Table 7.6 shows the results for K-Means Clustering. We display fewer results for it because FastR errored out for the Yelp, Books, and LastFM datasets. For Yelp, we got a segfault and, for the others, we got a `CHMOLD: problem too large` exception. These errors are not expected as the same experiment would succeed in GNU-R, so we believe these are GraalVM-internal exceptions. Note that GraalVM and FastR are rapidly evolving research codebases, so it's understandable that we would encounter this kind of problem. Exceptions aside, K-Means also breaks with the pattern because, unlike before, we see MorpheusR and TRINITY both experience slowdowns for most datasets except MovieLens. We refrain from further performance comments on K-Means because we could not test it extensively.

Summarizing Remarks While TRINITY achieves decent speed-ups in real world datasets, it simultaneously suffers from performance costs compared to MorpheusR. Namely, TRINITY is, for many of our tests, just over half as fast as MorpheusR. In future work, we hope to close this gap. On the other hand, platform-level errors prevent us from testing K-Means extensively, so that is something we will have to address in the future as well.

7.2.2 Visualizing the progression of training times

As previously described, we trained each algorithm 30 times in sequence, using the first 5 as warm-up and averaging the remaining 25. In this section, we plot the time it took to train the algorithms for each of those 30 trials. We do this to observe how GraalVM optimizes TRINITY's runtime. Additionally, we believe it may give us more insights on the *performance blips* discovered in our operator-level analysis. Here, we present 3 selected training time progressions, but the remaining ones may be found in the Appendix.

Figures 7.9 and 7.10 present the kind of behaviour we expected from our system. Namely, TRINITY's execution times lie somewhere between MorpheusR and the materialized approach. Note how TRINITY's performance dramatically improves, on both plots, from the first trial to the second; a common trend among the remaining visualizations from the Appendix. We believe

Figure 7.9: Linear Regression: Execution times per trial in LastFM

Figure 7.10: Logistic Regression: Execution times per trial in MovieLens

this suggests that GraalVM's JIT compiler has identified and collapsed (optimized) much of the indirection necessary for hosting the MorpheusDSL. Another interesting behaviour are the spikes in the execution time of the materialized approach in figure 7.9 at trials 8 and 14. Those may be another instance of the *performance blips* witnessed in the operator-level evaluation.

Figure 7.11 displays truly abherrant behaviour. As its clear from the visualization, TRINITY's performance varies dramatically from iteration to iteration with no discernible pattern. This may explain why we saw K-Means error out with some datasets: there may be something about the K-Means implementation that causes some underlying, platform-level, bugs to manifest. In any case, this is the "worst" behaviour of TRINITY that we on record and so we present this

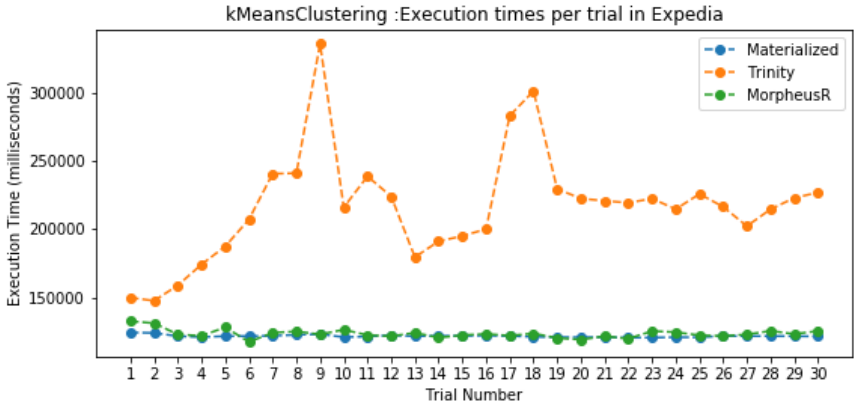


Figure 7.11: kMeansClustering: Execution times per trial in Expedia

graph to illustrate some that we still have much VM-performance-tuning work to do.

7.3 Preliminary Exploration of Polyglot Performance

Since the GraalVM runtime possesses polyglot execution capabilities, we would like to evaluate how TRINITY performs in polyglot programs. As a preliminary exploration of this feature, we evaluate TRINITY on Linear Regression and Logistic Regression implemented for NumPy but executing with R matrices as input.

Experiment Setup. For these experiments, we recorded the duration, in milliseconds, of 5 consecutive executions for Linear Regression and Logistic Regression. We consider the first trial as warm-up and report the average of the remaining 4. We ran this experiment using synthetically-generated datasets where we fix $n_S = 10^5$, $d_S = 20$, $TR = 5$, and vary only FR, the *feature ratio* $\frac{n_S}{n_R}$, from 1 to 5. We chose these experimental parameters are because we experienced runtime exceptions at higher memory loads and in long-running experiments.

General Observations. At a high-level, we see that TRINITY is able to optimize LA algorithms in polyglot programs as well. As expected, larger FR ratios appear to lead to monotonically greater speed-ups. We would like to test TRINITY in polyglot programs more extensively but we encountered C-level errors when our experiments dealt with larger matrices or ran for

Table 7.5: Polyglot Linear Regression results. **M**, **Sp**$_t$, **STD**$_b$, **STD**$_t$ refers to the mean runtime for the materialized approach, TRINITY's speed-up relative to the materialized alternative, the standard deviation of the materialized approach, and TRINITY's standard deviation respectively. The experiment threw a C-level exception for FR=3, which is why we do not show that result.

FR	**M**	**Sp**$_t$	**STD**$_b$	**STD**$_t$
1	208801.25	5.95	23836.72	575.30
2	233649.0	6.24	6410.88	355.45

Table 7.6: Polyglot Logistic Regression results. **M**, **Sp**$_t$, **STD**$_b$, **STD**$_t$ refers to the mean runtime for the materialized approach, TRINITY's speed-up relative to the materialized alternative, the standard deviation of the materialized approach, and TRINITY's standard deviation respectively.

FR	**M**	**Sp**$_t$	**STD**$_b$	**STD**$_t$
1	217470.25	4.61	18059.52	851.54
2	286053.0	5.72	39669.94	1541.39
3	360066.5	7.10	74777.20	1295.54

too long. That is why we show no results for Linear Regression with FR equal to 3; so this opens another dimension of VM-level work that we have to investigate further. Finally, these are to be taken as merely preliminary results suggesting that TRINITY may be a viable means of optimizing polyglot LA programs. Before making more a more confident conclusion, we will need to evaluate the performance of TRINITY executing on more LA-hosts while making use of GraalVM's polyglot features. Nonetheless, we are confident that our approach should work because of our unified matrix interface.

Chapter 8

Conclusions and Future Work

This thesis presents TRINITY, a host-agnostic implementation of the Morpheus optimization system for GraalVM languages. By making novel use of Truffle's interoperability features, we were able to encode Morpheus in such a way that the same implementation could be re-used for more than one language. Then, using GraalVM's R language implementation as a case study, we showed that TRINITY achieves decent speed-ups over materializing table-joins. Additionally, it appears that TRINITY has performance comparable to MorpheusR, a prior host-specific implementation of Morpheus for R. Case studies on other GraalVM languages are necessary before making further claims about the system, but current results suggest that we should be able to see similar benefits on other languages. Unfortunately, we could not do a case study on GraalVM's Python as the language is not mature enough to run our experiments. We are in the process of embedding TRINITY in GraalVM's Ruby and JavaScript implementations, which we understand to be more mature Truffle languages. After all, until at least another suitable host for TRINITY is evaluated, we cannot claim that TRINITY succeeds at optimizing multiple languages at once.

While building TRINITY, we also implemented a Truffle Library that provides Truffle language developers with a unified interface for manipulating foreign matrices. We believe this could be helpful for future linear algebra-focused projects on the GraalVM and are currently

working with Oracle Labs to determine whether or not it could be incorporated in other projects.

Finally, we also uncovered some strange behaviour that requires attention. For starters, we observe that sometimes smaller matrices lead to longer average execution times than larger ones. In response, we speculate that some VM-level process, such as a garbage collector pause, could be non-determistically slowing down our experiments. Furthermore, we are triggering many unexpected errors and exceptions when evaluating the k-Means Clustering algorithm, so we will need to work with the Oracle Labs team to find those bugs in order for us to properly evaluate this algorithm. Addressing these issues will be key to make the case that GraalVM is a suitable platform for linear algebra and a viable means of executing Morpheus-style optimizations.

This thesis, in full, is currently being prepared for submission for publication of the material. Justo, David; Stadler, Lukas; Kumar, Arun. The thesis author is the primary investigator and author of this material.

Appendix A

Real Execution Time Heatmaps

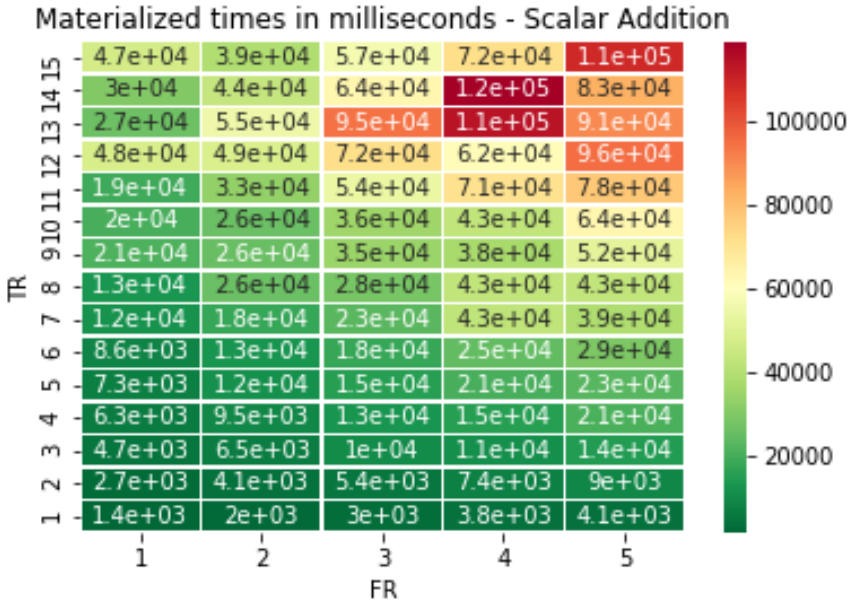


Figure A.1: Materialized times in milliseconds - Scalar Addition

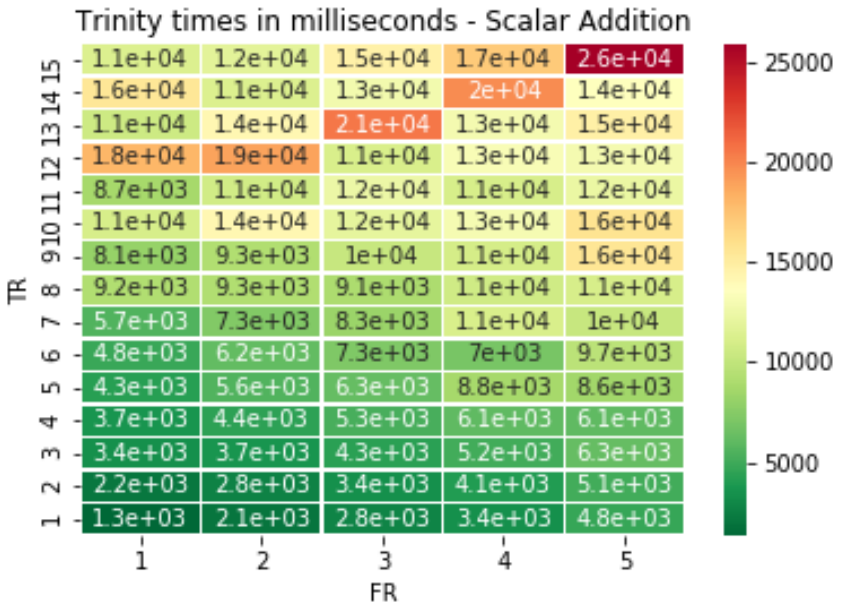


Figure A.2: Trinity times in milliseconds - Scalar Addition

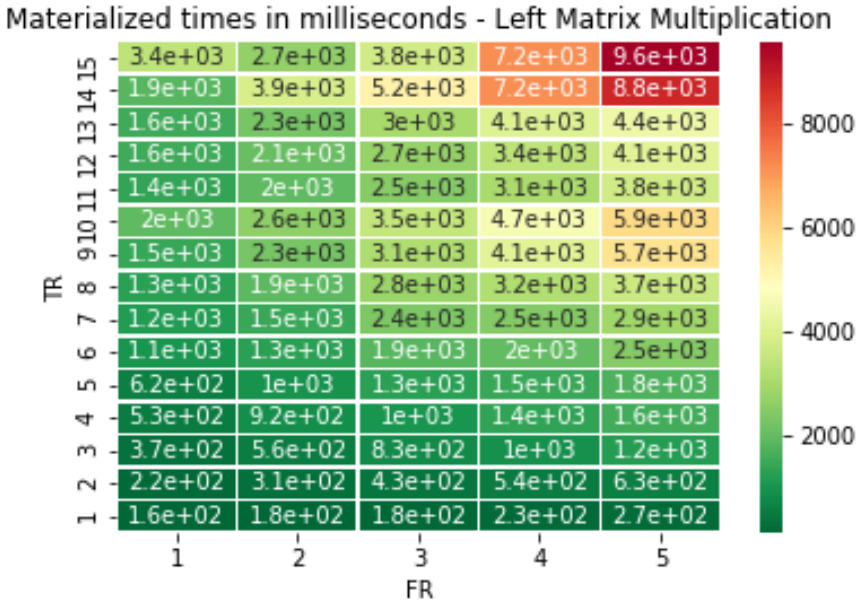


Figure A.3: Materialized times in milliseconds - Left Matrix Multiplication

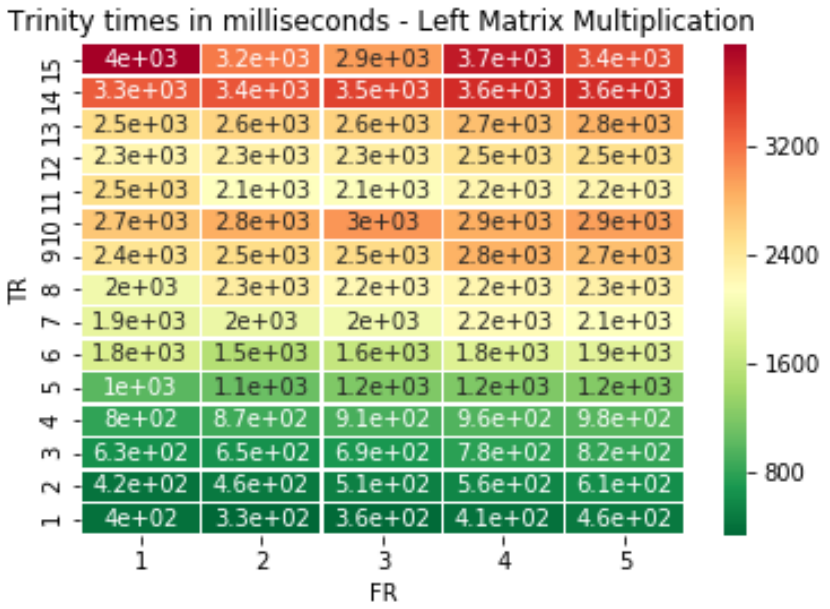


Figure A.4: Trinity times in milliseconds - Left Matrix Multiplication

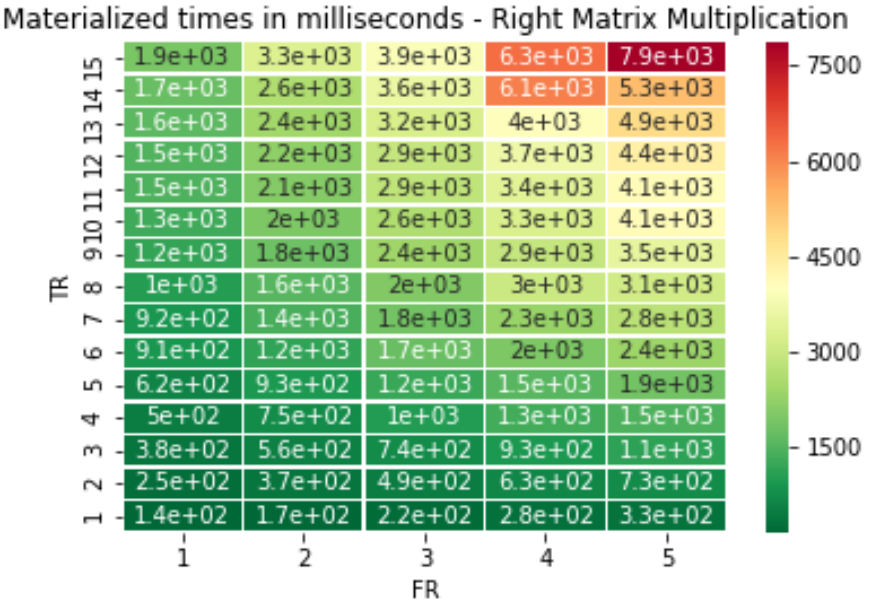


Figure A.5: Materialized times in milliseconds - Right Matrix Multiplication

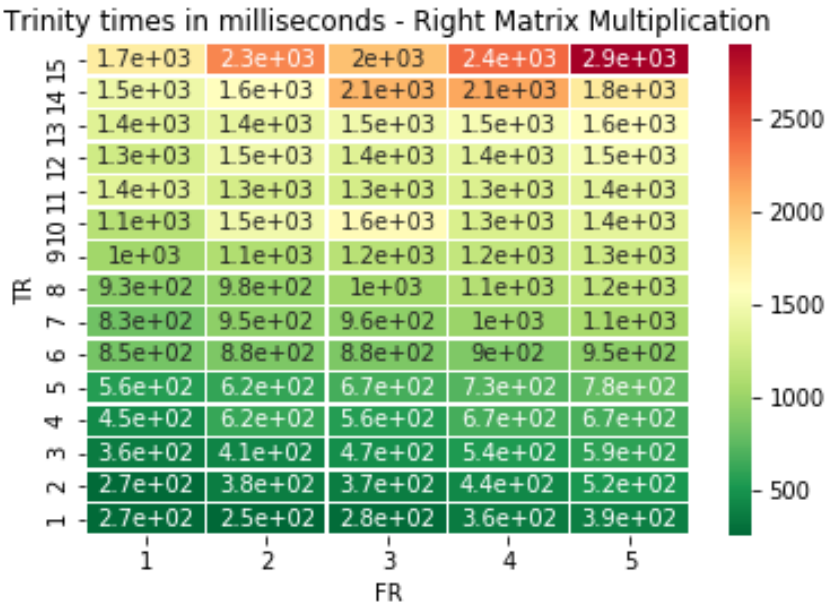


Figure A.6: Trinity times in milliseconds - Right Matrix Multiplication

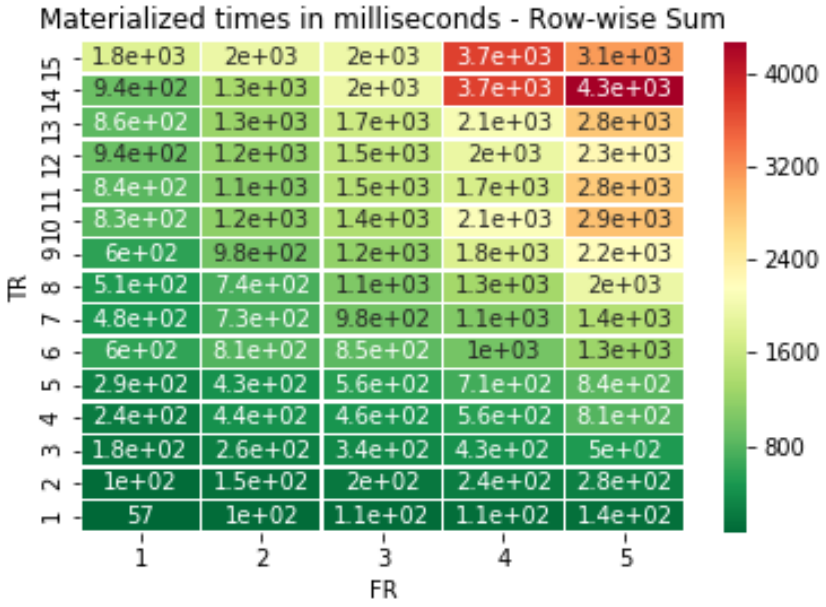


Figure A.7: Materialized times in milliseconds - Row-wise Sum

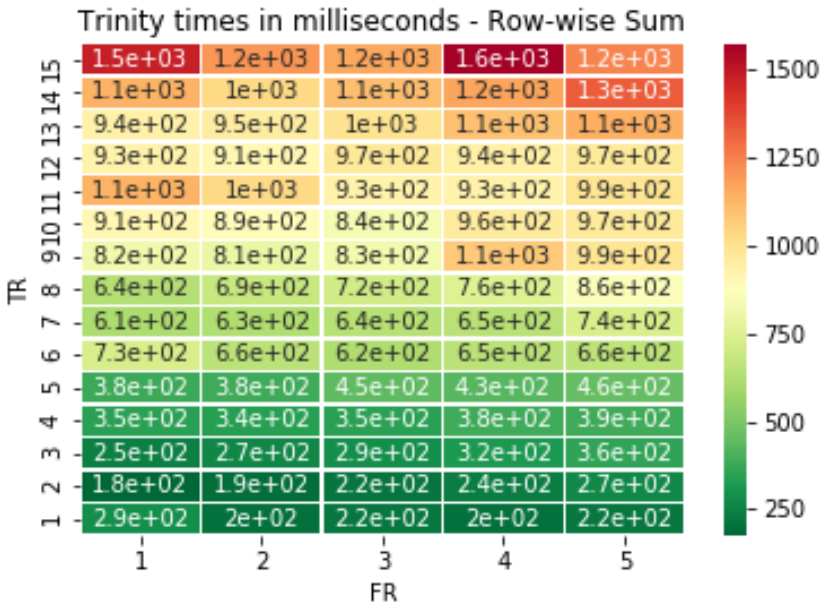


Figure A.8: Trinity times in milliseconds - Row-wise Sum

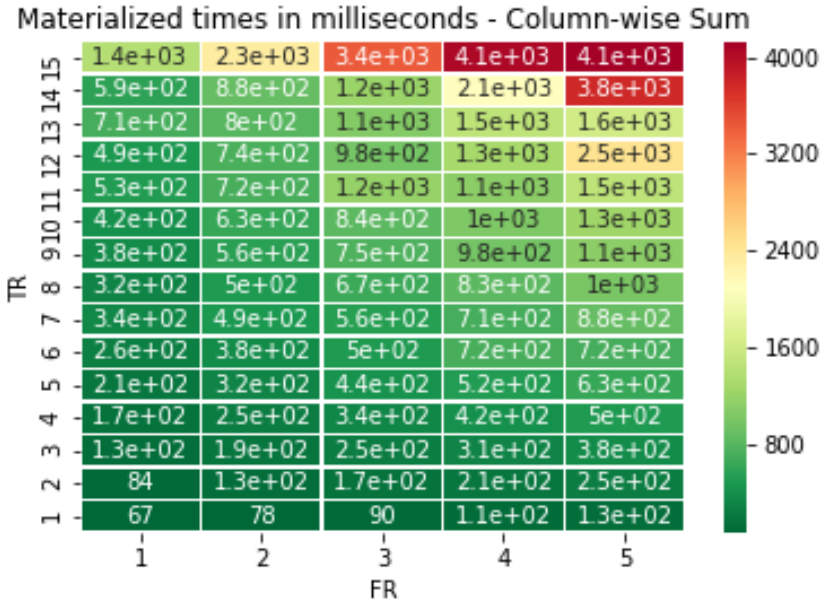


Figure A.9: Materialized times in milliseconds - Column-wise Sum

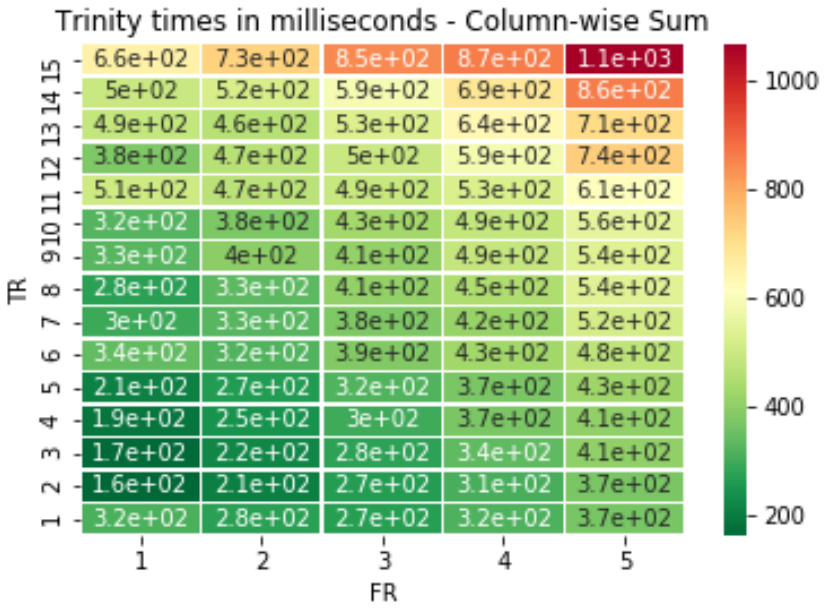


Figure A.10: Trinity times in milliseconds - Column-wise Sum

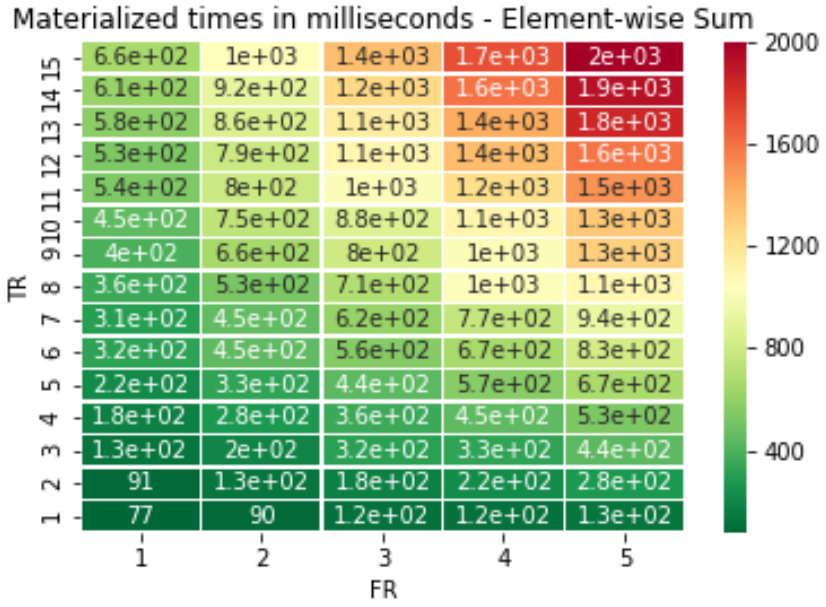


Figure A.11: Materialized times in milliseconds - Element-wise Sum

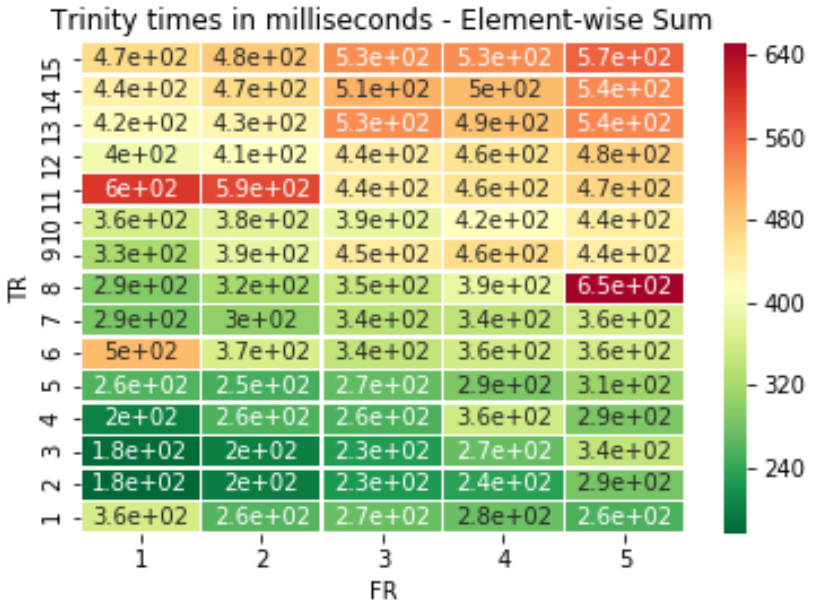


Figure A.12: Trinity times in milliseconds - Element-wise Sum

Appendix B

Training Time Progressions

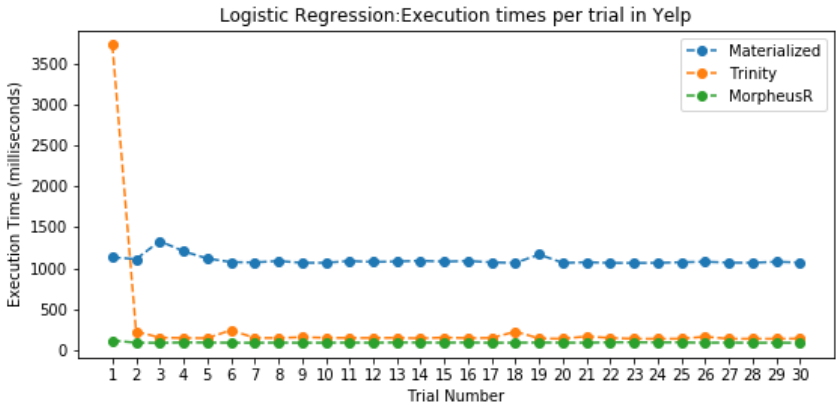


Figure B.1: Logistic Regression: Execution times per trial in Yelp

Figure B.2: Logistic Regression: Execution times per trial in Books

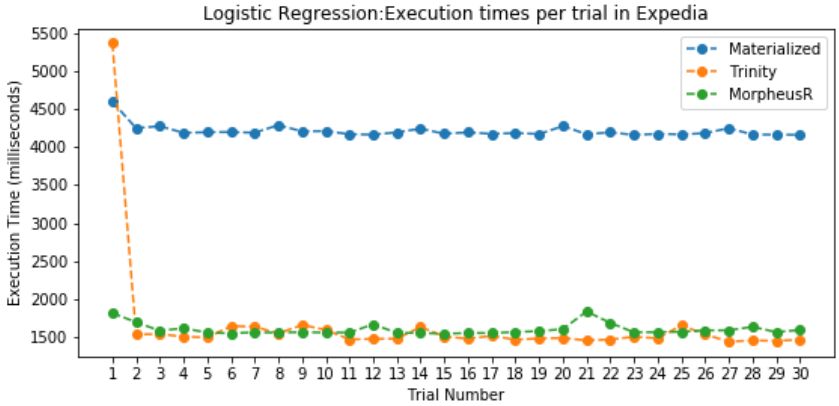


Figure B.3: Logistic Regression: Execution times per trial in Expedia

Figure B.4: Logistic Regression: Execution times per trial in Flights

Figure B.5: Logistic Regression: Execution times per trial in LastFM

Figure B.6: Logistic Regression: Execution times per trial in MovieLens

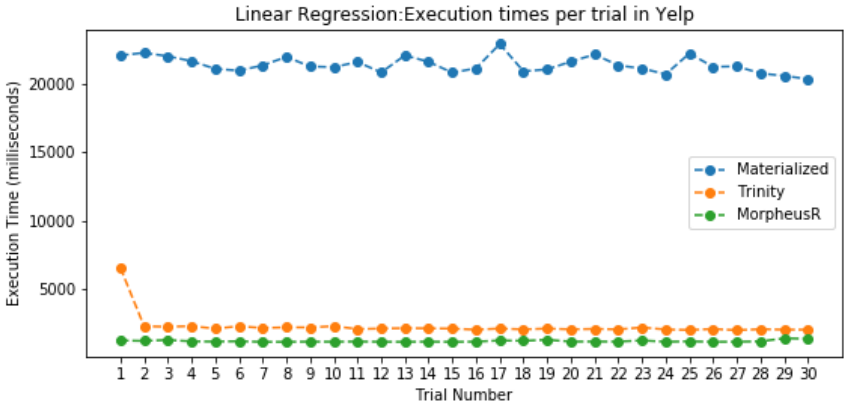


Figure B.7: Linear Regression: Execution times per trial in Yelp

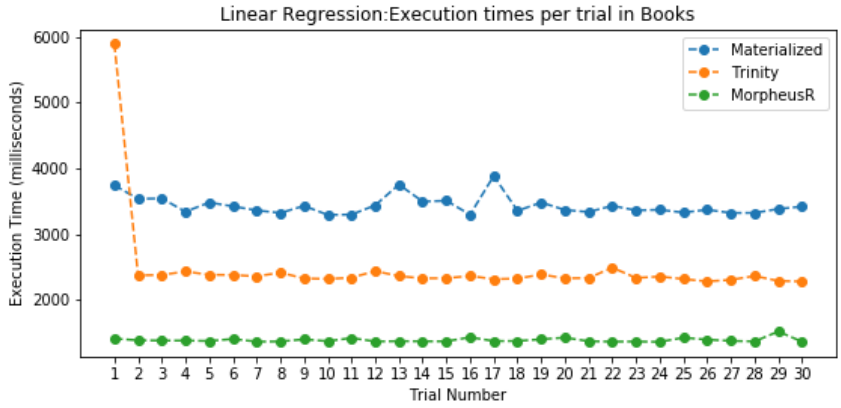


Figure B.8: Linear Regression: Execution times per trial in Books

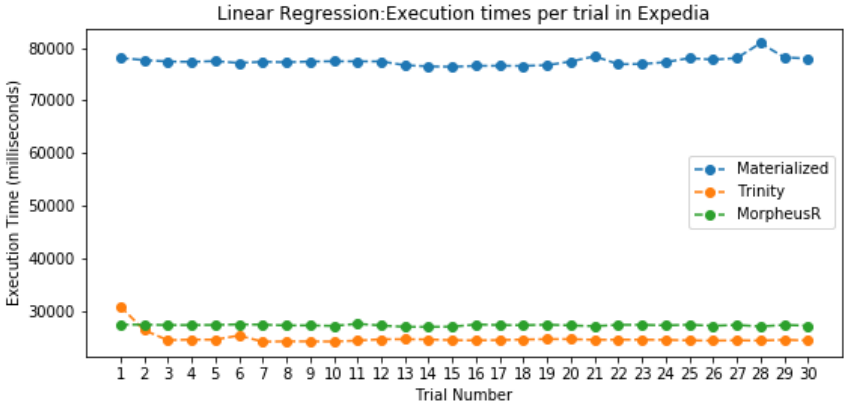


Figure B.9: Linear Regression: Execution times per trial in Expedia

Figure B.10: Linear Regression: Execution times per trial in Flights

Figure B.11: Linear Regression: Execution times per trial in LastFM

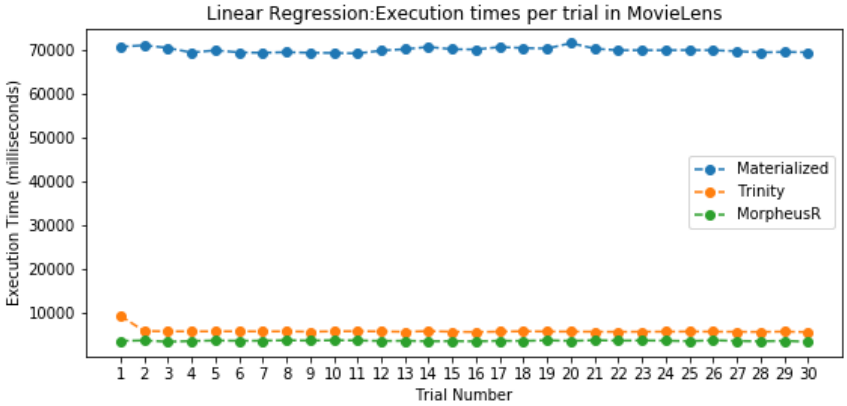


Figure B.12: Linear Regression: Execution times per trial in MovieLens

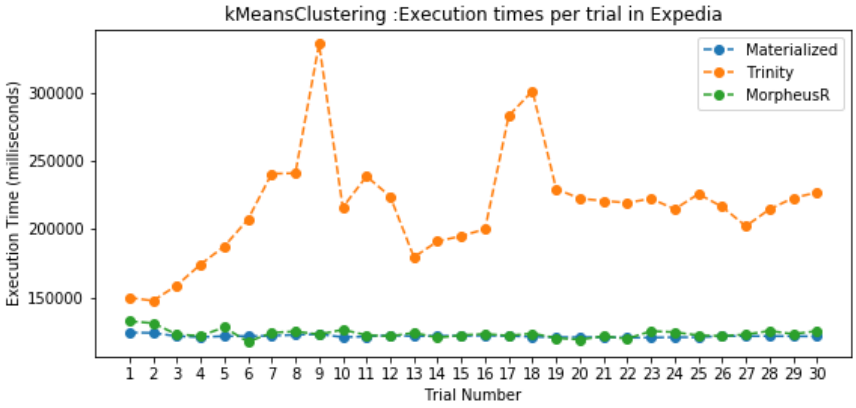


Figure B.13: kMeansClustering: Execution times per trial in Expedia

Figure B.14: kMeansClustering: Execution times per trial in Flights

Figure B.15: kMeansClustering: Execution times per trial in MovieLens

Bibliography

[1] Fastr github repository. https://github.com/oracle/fastr. Accessed: 2019-11-01.

[2] Graalpython github repository. https://github.com/graalvm/graalpython. Accessed: 2019-11-01.

[3] Graalvm github repository. https://github.com/oracle/graal. Accessed: 2019-11-01.

[4] Graalvm's polyglot tutorial. https://www.graalvm.org/docs/reference-manual/embed/. Accessed: 2019-11-01.

[5] Interop. graalvm.org/truffle/javadoc/com/oracle/truffle/api/interop/InteropLibrary.html. Accessed: 2019-12-01.

[6] Morpheus project main site. https://adalabucsd.github.io/morpheus.html. Accessed: 2019-11-01.

[7] Morpheusfi github repository. https://github.com/liside/MorpheusFI. Accessed: 2019-11-01.

[8] Morpheusflow github repository. https://github.com/ADALabUCSD/MorpheusFlow. Accessed: 2019-11-01.

[9] Morpheuspy github repository. https://github.com/ADALabUCSD/MorpheusPy. Accessed: 2019-11-01.

[10] Morpheusr github repository. https://github.com/lchen001/Morpheus. Accessed: 2019-11-01.

[11] Todd A. Anderson, Hai Liu, Lindsey Kuper, Ehsan Totoni, Jan Vitek, and Tatiana Shpeisman. Parallelizing Julia with a Non-Invasive DSL. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:29, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

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

[13] 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, pages 78–90, New York, NY, USA, 2015. ACM.

[14] Arun Kumar, Mona Jalal, Boqun Yan, Jeffrey Naughton, and Jignesh M. Patel. Demonstration of santoku: Optimizing machine learning over normalized data. *Proc. VLDB Endow.*, 8(12):1864–1867, August 2015.

[15] Arun Kumar, Jeffrey 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*, SIGMOD '15, pages 1969–1984, New York, NY, USA, 2015. ACM.

[16] Arun Kumar, Jeffrey 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 '16, pages 19–34, New York, NY, USA, 2016. ACM.

[17] Side Li, Lingjiao Chen, and Arun Kumar. Enabling and optimizing non-linear feature interactions in factorized linear algebra. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1571–1588, 2019.

[18] Side Li and Arun Kumar. Morpheuspy: Factorized machine learning with numpy. Technical report, 2018. Available at https://adalabucsd.github.io/papers/TR_2018_MorpheusPy.pdf.

[19] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. Towards polyglot adapters for the graalvm. In *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*, Programming '19, pages 1:1–1:3, New York, NY, USA, 2019. ACM.

[20] Dan Olteanu and Maximilian Schleich. F: Regression models over factorized views. *Proc. VLDB Endow.*, 9(13):1573–1576, September 2016.

[21] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 3–18, New York, NY, USA, 2016. ACM.

[22] Christian Wimmer and Thomas Würthinger. Truffle: A self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 13–14, New York, NY, USA, 2012. ACM.

[23] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. *SIGPLAN Not.*, 52(6):662–676, June 2017.

[24] 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.

[25] 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, pages 73–82, New York, NY, USA, 2012. ACM.