Towards A Polyglot Framework for Factorized ML

David Justo
University of California, San Diego
djusto@ucsd.edu

Shaoqing Yi
University of California, San Diego
shy218@ucsd.edu

Lukas Stadler
Oracle Labs
lukas.stadler@oracle.com

Nadia Polikarpova
University of California, San Diego
npolikarpova@eng.ucsd.edu

Arun Kumar
University of California, San Diego
arunkk@eng.ucsd.edu

ABSTRACT

Optimizing machine learning (ML) workloads on structured data is a key concern for data platforms. One class of optimizations called "factorized ML" helps reduce ML runtimes over multi-table datasets by pushing ML computations down through joins, avoiding the need to materialize such joins. The recent Morpheus system *automated* factorized ML to *any* ML algorithm expressible in linear algebra (LA). But all such prior factorized ML/LA stacks are restricted by their chosen programming language (PL) and runtime environment, limiting their reach in emerging industrial data science environments with many PLs (R, Python, etc.) and even cross-PL analytics workflows. Re-implementing Morpheus from scratch in each PL/environment is a *massive developability overhead* for implementation, testing, and maintenance. We tackle this challenge by proposing a new system architecture, *Trinity*, to enable factorized LA logic to be *written only once* and easily *reused across many PLs/LA tools in one go*. To do this in an extensible and efficient manner without costly data copies, Trinity leverages and extends an emerging industrial polyglot compiler and runtime, Oracle's GraalVM. Trinity enables factorized LA in multiple PLs and even cross-PL workflows. Experiments with real datasets show that Trinity is significantly faster than materialized execution (> 8x speedups in some cases), while being largely competitive to a prior single PL-focused Morpheus stack.

PVLDB Reference Format:
David Justo, Shaoqing Yi, Lukas Stadler, Nadia Polikarpova, and Arun Kumar. Towards A Polyglot Framework for Factorized ML. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at http://vldb.org/pvldb/format_vol14.html.

1 INTRODUCTION

Optimizing machine learning (ML) workflows over structured data on various data platforms is a major focus for the database community. In particular, a recent line of work optimized ML over datasets that are joins of multiple tables. Instead of forcing data scientists to always denormalize and materialize a bloated single table, *factorized ML* techniques rewrite and push ML computations down to the base tables [23, 35, 48, 49]. This is a form of cross-algebraic query optimization bridging relational algebra and linear algebra (LA) [36]. To enable factorized ML for many statistical ML algorithms in one go, the Morpheus project [23] *generalized* factorized ML, devising a framework that automatically factorizes any statistical ML algorithm expressed in terms of a subset of linear algebra (LA) operators.

Problem: Polyglot Industrial Data Science Landscape. The industrial data science landscape has recently exploded in its linguistic variety. Cross-PL *polyglot* workflows also now arise in practice, e.g., data preparation in Python, model building in R, and model deployment in Javascript [51]. All this PL diversity leads to a new practical bottleneck: *How to efficiently develop extensible factorized ML systems for many PLs at once?* This is a *novel developability challenge* from 3 standpoints. (1) Reimplementing factorized ML/LA code stacks *from scratch* in each new PL/LA system is a highly labor-intensive, tedious, error-prone, and ultimately costly process. (2) *Fragmentation* of factorized ML/LA code stacks across PLs also *duplicates efforts* and complicates testing, deployment, and maintenance. (3) As DB+ML researchers keep devising novel LA+relational algebraic optimizations (e.g., like [37]), prototyping them across PLs leads to tedious development grunt work for researchers. Figure 1(A) illustrates these three issues.

Motivation: Implementing Morpheus for Python. Morpheus was originally prototyped in R, informing its design with knowledge of matrix datatype representations, the efficiency of LA operators, and other PL-specific idiosyncrasies. When we wanted to port Morpheus to Python, we expected it to be a straightforward re-implementation of the same optimizations in Python, i.e., *same ideas, different syntax*. NumPy, Python's *de-facto* LA framework, replicates much of R's functionality in Python. But we found NumPy's design imposes some implementation restrictions that led to more dramatic runtime performance issues in Python than in R. Indeed, the original re-implementation in Python was slower than no optimizations at all! The culprit was not at the syntax level but rather a combination of wasteful memory allocation at the C-level and NumPy's eager conversion to dense matrices from sparse matrix operations. To make Morpheus in Python fast, it took us over two months to learn the idiosyncrasies of NumPy's LA performance and re-implement and tweak Morpheus optimizations many times over. Overall, this experience exemplifies the massive developability nightmare inherent in bringing modern data systems optimizations to multi-PL industrial data science environments.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

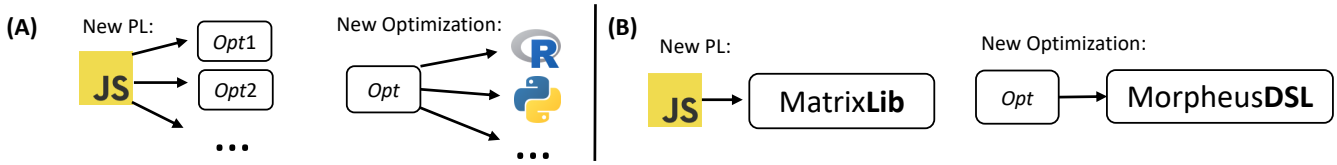

Figure 1: (A) Implementation burden *without* Trinity. Supporting a new PL requires re-implementing all optimizations in it; adding a new optimization (opt) requires porting it to each PL. This burden is "quadratic" (product) in the number of PLs and opts. (B) Implementation burden *with* Trinity. Supporting a new PL requires implementing only its MatrixLib interface; a new opt needs to be specified only once in MorpheusDSL (details later). The burden is "linear" (sum) in the number of PLs and opts.

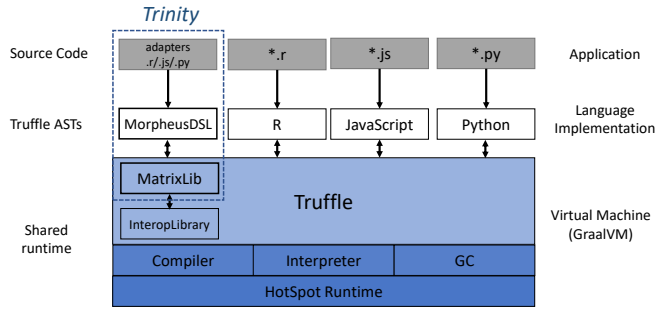

Figure 2: Trinity in and for GraalVM (Section 3 explains more). GraalVM supports many host PLs, they execute on a shared runtime Truffle, which provides services such as dynamic compilation and memory management. Truffle also provides interoperability services across PLs. We build two such services in Trinity to achieve our goals: MatrixLib (Section 4) and MorpheusDSL (Section 5).

Informed by our experience, in this paper we take a first step towards a first-of-its-kind polyglot framework for factorized ML to meet the above developability challenge. We unify ideas from 3 fields: DB, ML, and PL. Specifically, we exploit an emerging industrial polyglot compiler/runtime stack to architect a new data system that makes DB-style factorized ML optimizations easily *reusable* across PLs. That is, our approach imposes minimal cost for supporting new PLs and for updating the optimization techniques that get reused across PLs automatically. A key benefit is we rein in the engineering cost to be "linear" in the number of optimizations and PLs, as Figure 1(B) illustrates, rather than "quadratic."

System Desiderata and Challenges. We have 3 key desiderata for our polyglot framework. (1) *Genericity*: It should be generic enough to support many user-level PLs and LA systems. Existing stacks like Morpheus in R tightly couple factorized LA logic with the PL's runtime. Enabling PL-agnostic stacks and *interoperability* across PLs is challenging due to their differing object and memory management styles. (2) *Extensibility*: It should be relatively easy to extend support to new LA systems in PLs or even new PLs in the future. Today this requires deep knowledge on two fronts: Morpheus-style algebraic rewrite rules and the new PL/LA system's compiler/runtime environment. We want to mitigate this double whammy and enable a clean separation of concerns. (3) *Efficiency:* Ideally, the higher generality we seek will not sacrifice runtime performance too much relative to prior single PL-specific implementations.

Our Approach: Trinity. We present Trinity, the *first data system to offer factorized ML/LA as a "reusable service" that can be used in multiple PLs and LA systems in one go.* Trinity is built as a service in and for Oracle's GraalVM, a multi-PL virtual machine with good support for PL interoperability aimed at industrial data science applications [60]. We use GraalVM for two reasons. First, GraalVM lets us reuse the *same* underlying code stack for factorized ML across many user-level target PLs aka *host PLs* without needless data copies or movement.Second, GraalVM's advanced compiler and runtime capabilities also enable truly *polyglot* programs that span and *compose* PLs. Such capabilities are potentially beneficial for data scientists to be given more freedom to choose the idioms they prefer when writing LA scripts while also representing and transforming data with the right semantics. All of this is powered by an efficient interoperability protocol in a unified runtime. Figure 2 illustrates Trinity's high-level architecture.

Architecture and Techniques. We briefly explain the architecture and usage of Trinity (details in Section 3). Trinity offers a *clean separation of concerns* between 3 user/developer roles in our setting; data scientists (who write LA scripts), PL/GraalVM developers (who support more PLs/LA systems), and DB+ML researchers (who devise Morpheus-style rewrite rule optimizations). To this end, Trinity has 4 components: *MatrixLib*, a new matrix interoperability API; *MorpheusDSL*, a PL-agnostic Morpheus rewrite rule engine; a collection of host PL-specific matrix datatype *Adapters*; and *Normalized Matrix* datatypes in the host PLs with adapters.

MatrixLib (Section 4) allows PL/GraalVM developers to *specify* generic LA operations that are *reusable* across PLs. It enables Trinity to meet the desideratum of genericity. MorpheusDSL (Section 5) is an *embeddable* domain-specific language (DSL) for GraalVM whose abstract syntax tree (AST) node semantics correspond to the algebraic rewrite rules of Morpheus. It enables Trinity to meet the desiderata of genericity and efficiency. Finally, our rewrite optimization stack can be *transparently* invoked in concrete PL-specific LA scripts via matrix Adapters (Section 4). These serve as a flexible, fast, and easy-to-implement approach to support new PLs/LA systems. Thus, it enables Trinity to meet the desideratum of extensibility.

Novelty and Industrial Track Relevance. Trinity marries cutting-edge PL/compilers techniques with DB+ML systems techniques. But we believe its key novelty is less in its individual techniques and more in its first-of-a-kind holistic architecture that synthesizes the "right" techniques from disparate fields to meet our goals. To

the best of our knowledge, Trinity is the first data science system to offer *3 axes of generality in a unified manner*: support for many ML/LA scripts, support for many PLs/LA systems, and support for polyglot workflows. The industrial data science landscape is increasingly polyglot, with end-to-end workflows for data preparation, model building, and inference often straddling PLs such as Python, R, Java, and Javascript. Thus, Trinity's unified approach can help reduce implementation costs and maintenance headaches for industrial deployments. We believe our work is a timely case study of how emerging industrial data science products can benefit from the latest research in the DB+ML systems space.

Overall, this paper makes the following contributions:

- To the best of our knowledge, this is the first paper to study the problem of generalizing factorized ML/LA optimizations to multiple PLs/LA systems in one go.
- We architect a new system, Trinity, leveraging GraalVM to offer factorized LA as a generic reusable service: implement once in a DSL, reuse efficiently in a polyglot runtime.
- We devise new interoperability abstractions in GraalVM to let developers and researchers make such optimizations easily available to new PLs/LA systems.
- We demonstrate prototypes with 3 host PLs of GraalVM, including a cross-PL workflow to demonstrate the high generality of Trinity.
- We perform an extensive empirical analysis of Trinity's efficiency using synthetic and real-world multi-table datasets. Overall, Trinity offers substantial speedups over materialized execution, even 8x on some real datasets and ML algorithms. Trinity also has competitive runtimes relative to a prior single PL-specific stack, MorpheusR, with the max overhead being < 2x.

2 BACKGROUND AND PRELIMINARIES

2.1 Linear Algebra Systems

Linear Algebra (LA) is an elegant formal language in which many statistical and ML algorithms are expressed. Data are represented as matrices. *LA operators* transform a matrix/matrices to another matrix/matrices. Common LA operators include include scalar-matrix addition and multiplication, matrix-matrix multiplication, and matrix aggregation. An *LA system* is a tool that supports matrices as first-class datatypes and has many basic and derived LA operators. Popular examples include R, Python's NumPy and SciPy, Math.js in JavaScript, Matlab, and SAS IML [9, 12, 13, 15, 43, 56]. Many ML algorithms can be seen as LA scripts in which data and model parameters are all represented as matrices and manipulated with LA operators. For example, Algorithm 1 shows how a popular classifier, Logistic Regression trained using gradient descent, as an LA script.[1] Table 1 lists some common LA operators (ops) that arise in statistical ML scripts; this is the minimal set Trinity expects of an LA system. All popular LA systems offer such ops, and we use R, NumPy, and Math.js for our prototype but note that our approach is generic enough to support other LA systems too.

[1]For simplicity of exposition, we show batch gradient descent. Many sophisticated gradient methods such as conjugate gradient and L-BFGS can also be expressed as just LA scripts over the whole dataset [23]. A notable exception is stochastic gradient descent, which needs external mini-batch sampling operations over the data [55].

Algorithm 1: Logistic Regression (training loop)

Input: Matrix T, vector Y, vector w, scalar α
for i *in* 1 : max_iter **do**
  $w = w + \alpha * (T^T(Y/1 + exp(Tw)))$
end

Table 1: Set of LA ops for an LA system to work with Trinity. T is a dataset matrix; X is a parameter matrix; x is a constant.

Operator Type	Name	Expression
Element-wise Scalar Op	Aritmetic Op ($\oslash, =, +, -.*, /, \hat{}$, etc)	$T \oslash x$ or $x \oslash T$
	Transpose	T^T
	Scalar Function f	$f(T)$
Aggregation	Row Summation	$rowSums(T)$
	Column Summation	$colSums(T)$
	Summation	$sum(T)$
Multiplication	Left Multiplication	TX
	Right Multiplication	XT
	Cross-Product	$crossprod(T)$
Matrix Addition	Matrix Addition	$T + X$ or $X + T$

2.2 GraalVM and Truffle

Oracle's GraalVM project aims to accelerate the development of PLs by amortizing the cost of building PL-specific VMs [60] GraalVM PLs are implemented using Truffle [58, 61], an interpreter-writing framework, and execute on a modified version of the HotSpot VM named GraalVM [60]. This approach has led to the development of high-performance PL re-implementations that include R, Python, and JavaScript–referred to as FastR, GraalPython, and GraalJS, respectively [4–6]. Figure 2 gives and an architechtural overview of GraalVM and how Trinity fits into it.

A Truffle language is specified by writing an AST interpreter for it in Java and making heavy use of an annotation pre-processor to minimize boilerplate. Using the interpreter as input, GraalVM uses a technique called "partial evaluation" to generate compiled code for that language. In addition to a default implementation for an AST node, language designers are encouraged to provide alternative, specialized, variants that provide high-performance when operating over a subset of inputs. Then at runtime, GraalVM will speculate about future inputs and compile AST nodes into their optimized variants. If a speculative assumption is ever invalidated, the code is *de-optimized* and the node is replaced with a more general implementation [59].

Example. Consider the implementation of a binary addition operator (+) for a simple language supporting numeric addition and addition between `Strings` as concatenation. This node could then be implemented in Truffle as shown in Listing 1. We use the `@Specialization` annotation to provide alternative (optimized) implementations for different classes of inputs while `@Fallback` is used for providing a default "catch-all" behaviour: throwing a type exception [3, 17].

2.3 Polyglot Programs and Interoperability

Since all Truffle languages share the same implementation framework, GraalVM can seamlessly combine nodes from different languages within the same AST [29, 30]. In practice, this means that

```
@NodeInfo(shortName = "+")
public abstract class AddNode extends BinaryNode {
@Specialization // Specialization for numeric types
protected Number add(Number left, Number right) {
  return new Number(
    left.getValue().add(right.getValue()));
}
// Specialization for strings
@Specialization(guards = "isString(left, right)")
protected String add(Object left, Object right) {
  return left.toString() + right.toString();
}
@Fallback // catch-all specialization
protected Object typeError(Object left, Object right) {
  throw Exception.typeError(this, left, right);
}
}
```

Listing 1: Implementation of a binary addition operator for a simple Truffle language, adapted from [16].

```
import polyglot as poly
# Example 1
arr = poly.eval(language="R",string="c(42,39)")
print(arr[1]) # prints 39
# Example 2
mathInJS = """
const mathjs = require("mathjs")
class UseMath {
    function add(x, y) { return mathjs.add(x,y); };
}
"""
useMath = poly.eval(language="nodejs",string="mathInJS");
useMath.add(1, 2); # returns 3
```

Listing 2: Embedding an R fragment in Python using *polyglot-eval*. We generate a list in R and then access its elements.

end-users can seamlessly combine and execute fragments of different languages within the same script, and that the VM will partially evaluate and optimize the resulting multi-language programs.

Interoperability. When discussing GraalVM's interoperability features, we are describing the utilities that Truffle language implementers have in order to enable some data structure to be shared among other Truffle languages. For our purposes, this mostly refers to Truffle's INTEROP protocol, which maps messages to the language-specific operations that are used to interact with objects in a language-agnostic manner [31]. Listing 3 from a later discussion on interoperability exemplifies the usage of this technology.

Polyglot Programs. A polyglot program is one that composes functionality from multiple PLs within the same application [25]. Many Truffle languages facilitate this process by providing a function to enable end-users to syntactically embed fragments of other Truffle languages within another *host* language [2]. For instance, in Listing 2 we see an instance of a Python script (host) borrowing functionality from R, by creating a foreign vector of two numbers that can be accessed by the host, and JavaScript, where we encapsulate the functionality of the `math.js` library in a class and export it to Python so we can call on its methods. End-users of GraalVM's polyglot programs can safely assume that many primitive types such as arrays, strings, and numeric types will map their functionality to INTEROP messages such that they can be re-used in foreign languages using that language's native operators. For other types, while their interface is also mapped to INTEROP messages, their layout may be expected by other languages so interacting with them may require domain knowledge of the object's interface in its language of origin. This is one of the key issues this work addresses for matrix datatypes.

2.4 Notation: Normalized Data

For the sake of tractability, we focus on *star schema* primary key-foreign key (PK-FK) joins, which are common in practice. Snowflake joins can be easily reduced to star joins with relatively low overhead in our setting. We use the same notation as the Morpheus paper [23] for uniformity. For simplicity of exposition, we discuss notation only for a two-table join. We are given tables $\mathbf{R}(\underline{RID}, X_R)$ and $\mathbf{S}(Y, X_S, K)$. 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 $\mathbf{R}$. We refer to $\mathbf{R}$ as the *attribute* table (akin to dimension table in OLAP) and $\mathbf{S}$ as the *entity* table (akin to fact table in OLAP). The materialized join output is $\mathbf{T}(Y, [X_S, X_R]) \leftarrow \pi(\mathbf{S} \bowtie_{K=RID} \mathbf{R})$, where $[X_S, X_R]$ is the concatenation of the feature vectors. We use standard notation for the corresponding matrix representation of the feature vectors in a table: R for $\mathbf{R}.X_R$ and similarly, S and T.

2.5 Background on Morpheus

Normalized Matrix. This is a logical LA datatype introduced in Morpheus [23]. It represents the output of the join, T, in an equivalent "factorized" form using the inputs of the join as a 3-tuple: $T_N \equiv (S, K, R)$. Here, K is an ultra-sparse indicator matrix that encodes the PK-FK dependency as follows: $K[i, j] = 1$, if i^{th} row of $\mathbf{S}.K = j$, and 0, otherwise. Note the algebraic equivalence $T = [S, KR]$, wherein KR denormalizes the schema. Data scientists using Morpheus specify the base tables and foreign keys to create a Normalized Matrix. They then write LA scripts as usual using T_N as if it is a single data matrix. But under the hood, Morpheus *automatically* rewrites ("factorizes") LA operators over T_N into LA operators over S, K, and R. In this sense, Morpheus brings the classical DB notion of *logical data independence* to LA systems.

Morpheus Rewrite Rules. The core of Morpheus is an extensive framework of *algebraic rewrite rules* for LA operators over the Normalized Matrix. We present four key examples here and refer the interested reader to [23] for the entire set of rules.

$$\begin{aligned} T \oslash x &\rightarrow (S \oslash x, K, R \oslash x) \\ sum(T) &\rightarrow sum(S) + colSums(K) rowSums(R) \\ TX &\rightarrow SX[1 : d_S,] + K(RX[d_S + 1 : d,]) \\ XT &\rightarrow [XS, (XK)R] \end{aligned} \tag{1}$$

The first rewrite rule shows element-wise scalar addition/multiplication/etc. ($\oslash$ is from Table 1). The second shows full matrix

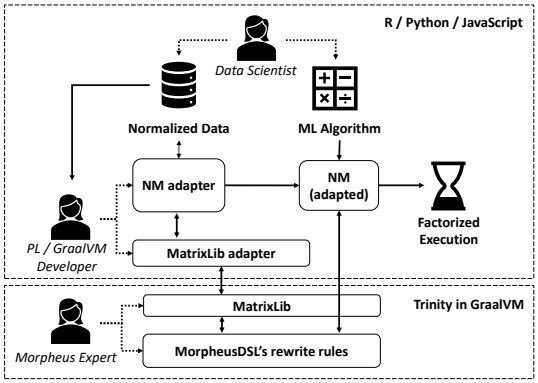

Figure 3: Usage of Trinity. Dotted arrows show what each user provides/maintains in the system. We use NM as short for Normalized Matrix. Solid arrows show the interaction between Trinity's components. The Morpheus expert and PL/-GraalVM Developer must work together to expand support to new host PLs in GraalVM by updating the adapters.

summation. The third one shows Left Matrix Multiplication (LMM). The last one shows Right Matrix Multiplication (RMM). Note that the first LA operator preserves the shape of the Normalized Matrix, while the others convert a Normalized Matrix to a regular matrix.

Redundancy Ratio. Morpheus reduces runtimes by reducing the raw number of FLOPS for LA ops on normalized data; recall that T has join-induced data redundancy. Thus, "factorized" execution has asymptotically lower runtime complexity as explained in [23]. We recap one example: LMM. Its number of FLOPS goes down from $d_X n_S(d_S + d_R)$ to $d_X(n_S d_S + n_R d_R)$. Note that star schemas typically have $n_S \gg n_R$. To theoretically estimate possible speedups, Morpheus defines two quantities: *tuple ratio* (TR): n_S/n_R and *feature ratio* ((FR): d_R/d_S. Since many LA ops are linear in the data size (CrossProduct is an exception), the *redundancy ratio* (RR) was defined to capture the FLOPS reduction: it is the ratio of the size of T to the sum of sizes of input matrices. As TR and/or FR go up, RR also goes up and Morpheus becomes faster than materialized. As our experiments show, many real-world datasets have non-trivial RR and thus benefit from Morpheus-style factorized execution.

3 SYSTEM OVERVIEW

3.1 Architechtural Overview

Design Goals and Intuition. Trinity is a first-of-its-kind polyglot framework for factorized ML. Our main design goal is to have a clean separation of concerns between 3 distinct groups of people, as Figure 3 shows, so that each can focus on their own expertise. We briefly highlight their roles (Section 3.2 goes into details). (1) *Data Scientist*, who can write LA scripts for statistical/ML analysis in any host PL without needing to know how factorized ML works. (2) *PL/GraalVM Developer*, who can add more host PLs or GraalVM optimizations without needing to know either ML or Morpheus rewrite rules. (3) *Morpheus Expert*, who can add/change rewrite rules while benefiting multiple host PLs *in one go*. GraalVM is a good fit for our goals because all its host PLs share a common implementation infrastructure Truffle, whose interoperability primitives enable us to *uniformly* specify rewrite rules just once.

Components. Trinity has 4 main components, as Figure 3 shows. (1) *MatrixLib*, a matrix *interoperability API* that provides a uniform means of interfacing with matrix datatypes regardless of their host PL/LA system of origin. (2) *MorpheusDSL*, an *embeddable DSL* to specify rewrite rules that has the benefit of being efficiently co-optimized with the host PL. (3) *Bi-directional adapters* to map between our generic matrix interface and a concrete interface of matrices in some PL/LA system. (4) *Normalized Matrix* constructor, which is what a Data Scientist will use to specify the base tables and foreign key matrices. It is basically host PL-specific syntactic sugar to shield Data Scientists from Trinity's system internals.

Component Interactions. MorpheusDSL's rewrite rules are implemented in terms of MatrixLib calls, abstracting the idiosyncrasies of each PL/LA system into a uniform interface. GraalVM's polyglot functionality enables us to embed MorpheusDSL's rewrite rules within any host PL. Internally, the Normalized Matrix constructor calls MorpheusDSL to obtain a Normalized Matrix; before returning, it *adapts* the object to conform to the appropriate matrix interface in the host PL/LA system. All LA op calls for this object would go through the adapter, which takes care of delegating them to MorpheusDSL for the rewrite rules to execute. The rewrite rules can execute because they interface with a *MatrixLib* adapter, which exposes a generic interface to manipulate the base matrices.

3.2 Using and Extending Trinity

Recall that Trinity offers a clean separation of concerns between Data Scientist, PL/GraalVM Developer, and Morpheus Expert. We now explain their usage/interaction with Trinity. This discussion shows how Trinity offers *high generality along 3 axes*: (1) Support for *multiple ML algorithms* expressed in LA, (2) Support for *multiple host PLs*, and (3) *Easy extensibility* to more algebraic and/or cost-based optimizations devised by DB+ML systems researchers.

3.2.1 Data Scientists. A Data Scientist benefits from Trinity because all LA-based ML algorithm implementations in the host PLs' LA systems now get *automatically* factorized over normalized data, potentially making their analytics run faster. New statistical/ML scripts in any of the host PLs–Python, R, Javascript, etc.–and *multilingual scripts* spanning these PLs also benefit. This high a level of generality for automating factorized ML *did not exist* before Trinity.

Data Scientists need only use our Normalized Matrix constructor to specify the base table and foreign key matrices (S, R, and K) instead of manually materializing the join. Recall that the output of this constructor is an object whose interface resembles that of a matrix. So, Trinity (like Morpheus) enables Data Scientists to perform their usual analyses on this "logically single but physically normalized" matrix. In the future, we can also consider overloading the join operator itself in a host PL/LA system to auto-generate this Normalized Matrix construction too.

3.2.2 PL/GraalVM Developers. A PL/GraalVM Developer benefits from Trinity because it makes it simpler for them to offer the benefits of factorized ML across host PLs, including future ones, e.g.,

Ruby or Scala, or even across different LA systems in existing host PLs. Without Trinity, they would have to deeply understand, reimplement, test, debug, and maintain the Morpheus rewrite rules *from scratch* separately for each host PL. With Trinity, they only need to provide two simple adapter classes to map their PL/LA System's matrix interface to that of Trinity and vice versa.

Bi-directional Adapters. Concretely, the first adapter is from a PL/LA System's matrix interface to that of MatrixLib; the second is from MorpheusDSL's Normalized Matrix interface to that of their LA System's. The latter should also export the Normalized Matrix constructor that calls MorpheusDSL. The PL/GraalVM Developer thus only needs to know how to use GraalVM's *polyglot-eval* utility, which has extensive documentation, community, and forums.

Beyond Factorized ML. Interestingly, MatrixLib's utility actually extends beyond Trinity and factorized ML. It is general enough to define other interoperable procedures or optimizations for LA. For instance, a GraalVM Developer can add Matlab-style optimizations for reordering matrix chain multiplications, a classic optimization in LA. By providing adapters to MatrixLib, the GraalVM Developer can also piggyback on future optimizations in GraalVM.

3.2.3 Morpheus Experts. A Morpheus Expert or similar DB+ML researchers working on cross-algebraic optimizations spanning linear and relational algebras also benefit from Trinity. This is because they need to implement their novel rewrite rules *only once* and Trinity makes their benefits automatically available across many host PLs/LA systems. This dramatically reduces software prototyping/engineering effort for DB+ML researchers and can potentially spur more research innovations in the LA+relational algebra query optimization space.

Adding Novel Rewrite Rules. To add a new rule, a Morpheus Expert only need to modify the MorpheusDSL implementation in Truffle in terms of MatrixLib invocations. They can then ask the PL/GraalVM Developer to update the Normalized Matrix adapter to support the new ops or to do so themselves.

Limitation: More LA Ops. We remark that one limitation in our setup arises when MatrixLib itself needs changes, e.g., adding novel rewrite rules in terms of *other* LA ops beyond Table 1. In this case, the Morpheus Expert must work closely with the PL/GraalVM Developer to ensure that the new LA ops are actually supported in the host PLs/LA systems and if so, expand the implementations of the MatrixLib interface and adapters accordingly.

4 MATRIX INTEROPERABILITY API

4.1 Design Considerations

Desiderata. We need an *interoperable* API to express and execute LA operations regardless of the host PL. This API will be used to represent Morpheus rewrite rules as generic procedures re-usable across host PLs/LA systems. We have 2 key desiderata for such an API. (1) *Generic Availability*: it should be usable by multiple host PLs/LA Systems and abstract over their specific matrix idioms. (2) *Extensiblility*: it should be easy for new host PLs/LA Systems to use the API without affecting prior procedures/stacks that used it.

Technical Challenge: FFIs. Interoperability between PLs is a known hard problem and an active area of research in the PL community [24, 28, 31]. One approach is to target the Foreign Function Interface (FFI) of each PL we want to support, while abstracting the idiosyncracies of each with a unified API that translates high-level directives to the right FFI calls. But this complicates *generic availability* and *extensibility* because new host PLs/LA Systems may not expose FFIs for our implementation language of choice.

Technical challenge: IRs. Another approach constitutes targeting or creating a shared intermediate representation (IR) for relevant LA Systems and encode rewrite rules with the IR. This can work but it is nuanced to get such an IR at the right level of abstraction. For instance, if it is too low-level, it may be too cumbersome to express LA operators and rewrite rules in it. But a major concern with IRs is *extensiblility*: to enable a new host PL/LA System to work with the IR, one would effectively need a small compiler to support it, which is daunting.

Design Decisions and Tradeoffs with GraalVM. Observing the above challenges is what led us to choose GraalVM, shared runtime and a multi-lingual VM as the infrastructure to implement our interoperability API. GraalVM offers two main advantages. (1) All host PLs on it already share a high-level IR in the form of Truffle ASTs; so, identifying and modifying LA operator calls is much easier [59]. (2) It already exposes a foundation of interoperability APIs that works across all host PLs [31]; this reduces work for correctly interacting with foreign datatypes.

The above said, we have 2 limitations due to GraalVM. (1) Our reach is currently limited to GraalVM's host PLs. But they already have working implementations of multiple PLs, including Python, R, JavaScript, Java, Ruby, and more. (2) Support for some newer host PLs is still at the experimental stage, i.e., they are not fully stable yet. This means our rewrite rules may not yet offer runtime benefits in those PLs on par with older PLs.

Nevertheless, since GraalVM is a major active industrial project, we believe the above limitations will get mitigated over time. A tighter integration of GraalVM with the Oracle Database is also likely in the future; this could also bring the benefits of Trinity to Oracle's enterprise customers [19].

4.2 MatrixLib Features

Overview and Motivation. MatrixLib is a new Truffle Library for interoperability on matrices that supports multiple host PLs/LA Systems running on top of GraalVM. GraalVM Developers can use this unified API to interact with foreign matrix objects. Its rationale is as follows. When implementing AST nodes, Truffle developers have access to an interoperability protocol for manipulating foreign objects. However, Truffle does not provide abstractions to deal with many classes of common foreign datatypes such as trees and matrices. Thus, a Truffle-level procedure expecting a foreign matrix may need to know, in advance, the public interface of the PL/LA System to support and worse, handle each interface separately.

Example. Suppose we have to implement an AST node receiving a matrix input, with the output multiplying that matrix by 42. Suppose we expect NumPy or R matrices as input. Naively, we would write something similar to Listing 3. In it, we utilize InteropLibrary

[10], Truffle's programmatic means of sending interoperability messages to call methods from the input matrix. Even though this computation is trivial, 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 cannot support a wider set of matrix datatypes without extending this procedure. Thus, code duplication and grunt work gets amplified and quickly becomes unwieldy as we start supporting more LA operations and rewrite rules on them.

```
1  @Specialization
2  Object doDefault(Morpheus receiver, Object matrix,
3  @CachedLibrary("matrix") InteropLibrary interop)
4    throws UnsupportedMessageException {
5    // Handles each kind of matrix separately
6    Object output = null
7    boolean isPyMat =
8      interop.isMemberInvocable(matrix, "__mul__");
9    if(isPyMat){ // NumPy case
10     output = interop.invokeMember(matrix,
11     "__mul__", 42);
12   } else {    // R case
13     output = interop.invokeMember(matrix, "*", 42);
14   }
15   return output;
16 }
```

Listing 3: Implementation of a polyglot node for multiplying a NumPy or R matrix by 42. Note how each case must be handled separately. Lines 1-4 set up the node and enables interoperability calls. Line 7-8 naively checks if the input matrix came from Python. The rest use interoperability calls to invoke the right method names depending on the conditional check.

MatrixLib is a Truffle Library that simplifies the implementation of Truffle AST nodes operating over foreign matrices. Its key benefit is eliminating the need to know, in advance, the interface details of a foreign input matrix. Instead, MatrixLib users are given a unified interface supporting a variety of common matrix operations that foreign input matrices are *expected* to support. As an example, compare Listing 3 with Listing 4. We implemented MatrixLib as a Truffle Library, which a Truffle mechanism for exporting messages, a kind of interface, on Truffle-level objects [18].

Is MatrixLib a Java Interface? What about Efficiency? No, MatrixLib is implemented as a Truffle Library, which provide performance benefits we'll discuss shortly. However, MatrixLib does *specify* generic matrix interface for multiple host PLs. In particular, it expects its inputs to expose a variety of basic and self-descriptive LA method names such scalarAddition, rowWiseSum, rowWiseAppend, splice, transpose, getNumColumns, etc. Talking to host PLs is handled dynamically via an *Adapter*, which guarantees the expected matrix interface. Listing 5 shows an example of a MatrixLib Adapter for JavaScript. Overall, MatrixLib is one of the first external uses of the Truffle Library system, to the best of our knowledge. Finally, Truffle Libraries allow for context-specific inlining, caching, and other compiler optimizations. So, these let us meet our efficiency desiderata as well in spite of writing MatrixLib in Java [18].

```
1  @Specialization
2  Object doDefault(Morpheus receiver, Object matrix,
3   @CachedLibrary("receiver.adapter")
4   MatrixLibrary matlib)
5      throws UnsupportedMessageException {
6
7       Object output = matlib.scalarMultiplication(
8         receiver.adapter, matrix, 42);
9      return output;
10 }
```

Listing 4: Implementation of a polyglot node for multiplying an R or Numpy matrix by 42 via MatrixLib; note its succinctness vs the alternative. Lines 1-5 set up the node to use MatrixLib. The rest uses MatrixLib to uniformly invoke the scalar-multiplication operator for any input matrix.

```
1   class MatrixLibAdapter {
2     constructor() {}
3     scalarAddition(matrix, number) {
4         return math.add(matrix, number);
5     }
```

Listing 5: Preview of a MatrixLib adapter for JavaScript.

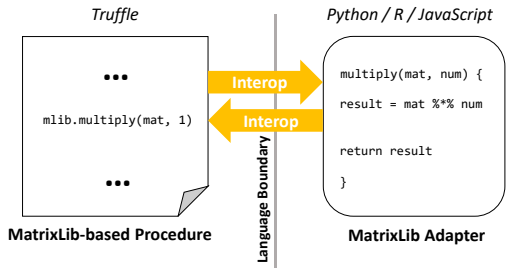

Figure 4: MatrixLib is abbreviated as mlib. Suppose the matrices' host uses $\%*\%$ for matrix multiplication (like R syntax). MatrixLib crosses the language boundary every time an LA operation is invoked. Built on top of Truffle's basic Interoperability API, it relies on adapters exporting a unified interface for matrices.

Crossing the PL Boundary. As Figure 4 shows, MatrixLib procedures work by requesting a sequence of computations in some Truffle language, the one where the operation's receiver resides. Implementation-wise, this is performed by making Truffle interoperability calls to a generic matrix interface, which is guaranteed by the adapter. Each interoperability call takes as input an adapter, providing the translation, and is followed by the operation's LA inputs. This let us compose generic sequences of LA operations.

5 EMBEDDABLE MORPHEUS DSL

5.1 Design and Overview

Overview. We now explain how we use MatrixLib to offer the Morpheus rewrite rules in a host PL-agnostic way. Recall two key desiderata: *generic availability* and *extensibility*. We achieve both by making a key observation: if we can represent our rewrite rules as a small Truffle "Language" itself, we can piggyback on GraalVM's pre-existing *polyglot-eval* functionality, i.e., its ability to talk *across* PLs. Thus, we use a level of *indirection* to represent our rewrite rules as a "language" itself: we call it *MorpheusDSL*. This meets both the above desiderata. But it raises a key question: will the cross-language overhead sacrifice our *efficiency* desideratum, i.e., kill the runtime benefits offered by Morpheus rewrite rules?

Addressing the Efficiency Challenge. We employ two mechanisms. (1) Since GraalVM handles multi-lingual programs using the same IR, it already optimizes across PL boundaries. So, its compiler should be able to "collapse" some of our indirection given enough warm-up. (2) The use of Truffle specializations machinery in MatrixLib enables context-speficic inlining, inline caches, and other compiler optimizations.

Obtaining a Normalized Matrix. MorpheusDSL has no distinctive syntax; all expressions in this language evaluate to returning a constructor for a Normalized Matrix. Once constructed, we effectively have access to a *guest* object in some *host* PL. The object's interface will not match the host PL's expected matrix APIs; so we utilize *adapters* to provide the translation.

AST Rewriting. The rewrite rules in MorpheusDSL orchestrate a sequence of LA operator calls *to the host* PL/LA System's physical LA operator implementation. Note that Trinity does *not* have its own physical LA operator implementations. It is the host PL that runs the actual LA computations using its own AST nodes. Figure 5 illustrates this. This is in line with prior PL-specific Morpheus implementations. In a sense, this is the *generalization of the classical DB idea of logical data independence to polyglot environments*. Recall that MatrixLib is only a means of making interoperability calls. To the best of our knowledge, this is the first Truffle "language" that works like this over data objects. Overall, even though the the rewrite rules may produce intermediate data, MorpheusDSL does *not* result in extra data copies but rather the data resides where it is: the host PL.

5.2 Usage and Requirements

We bundled MorpheusDSL with our custom GraalVM distribution; so, it is available for interoperability from any host PL. We now explain how it can be used by a PL/GraalVM Developer to support a new host PL.

Example. Suppose we want to obtain a Normalized Matrix from within JavaScript. We request its constructor from MorpheusDSL and give the appropriate inputs: an entity table matrix S, an array of indicator matrices K's, an array of attribute table matrices R's, and a *matrixLib* adapter. Listing 6 illustrates this. Listing 7 previews how the Javascript Normalized Matrix adapter exports the host PL's matrix interface from Trinity's generic MatrixLib interface

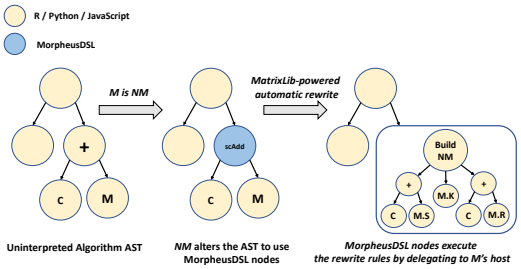

Figure 5: MorpheusDSL dynamically factorizes ML algorithms. C refers to some numeric constant, M refers to some matrix datatype and NM refers to the adapted Normalized Matrix. The rewrite occurs by using the host language's own LA operator implementations

```
// assume S, K, R are pre-existing math.js matrices
// Obtain the Normalized Matrix constructor
let adapter = new MatrixLibAdapter();
let constructor = Polyglot.eval("morpheusDSL", "");
this.normMatrix = constructor.build(S, K, R, adapter);
// now we can use the normalized matrix, ex:
let normMatrix.elementWiseSum(); // returns sum
```

Listing 6: Obtaining a Normalized Matrix in JavaScript

```
  // scalar addition
  const addScalar = math.typed('addScalar', {
    'NormMatrix, number': function (a, b) {
        let normMatrix = a.morph.scalarAddition(b);
        return new NormMatrix({"nmat": normMatrix});
    },
    //...
  } // matrix multiplication
  const multiply = math.typed('multiply', {
    'Matrix, NormMatrix': function (a, b) {
        return b.normMatrix.
          rightMatrixMultiplication(a);
    },
    // ...
});
```

Listing 7: A preview of the Normalized Matrix adapter in Math.JS. In most cases, adapting a factorized LA op like requires only a simple function invocation. In cases such as scalarAddition, where the result is also a Normalized Matrix, we need to adapt the output before returning

Rewrite Rules as AST Nodes. 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 a case study, we discuss the implementation of scalar addition. Recall its rewrite rule from Section 2. Listing 8 shows how that is implemented in MorpheusDSL.

```
1  Object doDefault(NormalizedMatrix receiver,
2   Object num, @CachedLibrary("receiver.adapter")
3   MatrixLibrary matrixlib) {
4
5    int size = receiver.Rs.length;
6    Object[] newRs = new Object[size];
7    for(int i = 0; i < size; i++) {
8     newRs[i] =
9      matrixlibGen.scalarAddition(receiver.adapter,
10       receiver.Rs[i], num);
11   }
12   Object newS =
13    matrixlibS.scalarAddition(receiver.adapter,
14      receiver.S, num);
15    // return a new normalized matrix
16    return createCopy(newS, receiver.Ks, newRs,
17     receiver.adapter);
18 }
```

Listing 8: Implementation of an interoperable scalarAddition using MatrixLib. Lines 1-6 set up the node and enable Truffle to cache specializations for the matrix arguments. Lines 7-16 perform the rewrite rule. The remaining lines return a new Normalized Matrix instance.

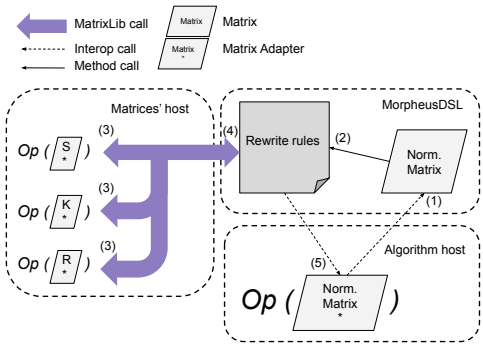

Figure 6: How Trinity's components interact for factorized execution. (1) Host PL invokes an LA op from the Normalized Matrix Adapter, whose inner object is a foreign datatype originating from MorpheusDSL. (2) Rewrite rule begins executing in MorpheusDSL. (3) Since the rewrite rule is implemented as MatrixLib calls, they execute ops on foreign matrices, which may originate from a PL different than the invoking host PL. (4) The result returns to MorpheusDSL. (5) MorpheusDSL returns result to invoking host PL.

5.3 End-to-End Working Example

Now that we have seen all the components of Trinity, we reuse our running example of scalar addition to walk through how various Trinity components interact when performing factorized execution. Figure 6 illustrates this. For exposition sake, we assume GraalJS is the host PL. Assume we have already constructed a Normalized Matrix; we add a constant to it.

The Data Scientist interacts with the adapted Normalized Matrix. So, the process begins by executing the addition method in the Adapter, which redirects that call to MorpheusDSL, as line 4 of Listing 7 shows. That leads to MorpheusDSL's scalarAddition node implementation, shown in Figure 8. The code implements the rewrite rule seen in Section 2.5 in terms of MatrixLib calls. Each MatrixLib call leads to the execution of FastR code corresponding to the MatrixLib LA operation requested. Here, the MatrixLib call from Listing 8 invokes the scalarAddition method of a GraalJS adapter, as implemented in Listing 5. After executing the rewrite rule's steps, Trinity returns a new Normalized Matrix reference to the caller, GraalJS. Back at the caller, since we received a Normalized Matrix from MorpheusDSL, we adapt the foreign Normalized Matrix before returning, as line 5 of Listing 7 shows. Since we have an adapted Normalized Matrix, future LA ops will be factorized the same way.

5.4 Factorizing Polyglot Scripts

GraalVM enables truly polyglot scripts wherein a user can, say, write LA scripts in one PL for ease of coding/maintainability but represent the matrices in a different foreign language's matrix implementation, say, for memory efficiency.

In Trinity, if matrices are represented in a host PL/LA system different from the one in which the LA script is written, then the rewrite rules that do not output a Normalized Matrix will output a foreign matrix. In such cases, and in order for the script to work, end-users must provide a mapping from their matrices' host PL to that of the host PL in which their script is written. This is reasonable because they need to write such a mapping anyway (i.e., even without Trinity) for their polyglot script to work. Conveniently, supporting Trinity already gives users the components they need to provide this mapping as we have developed *bi-directional* adapters to- and from- MatrixLib. In our experience running polyglot experiments with Trinity, only minor control-flow tweaks were needed in the existing adapters for the r and Python to work in tandom.

6 EXPERIMENTAL EVALUATION

We now empirically evaluate Trinity's *efficiency* on both real-world and controllable synthetic datasets. We also highlight Trinity's *generality* by using all of FastR, GraalPython (NumPy), and GraalJS (Math.js). Specifically, we answer the following 3 questions on efficiency. (1) How well does Trinity perform against *Materialized* execution for various redundancy ratios? (2) What is the overhead of Trinity relative to MorpheusR, a prior PL-specific implementation with much less generality? (3) How well does Trinity support new host PLs and polyglot scripts?

As a heads-up summary, our results show the following: (1) Trinity is significantly faster than *Materialized*; the speedups are in line with the expectations based on data redundancy ratio. (2) Trinity's speedups are just as competitive that of MorpheusR; their gap in real-world datasets is no more than 1.2× and often, much less. (3) Trinity works seamlessly with Javascript and a polyglot Python-R script. These newer host PLs see stranger behaviors, however, due to known GraalVM issues that Oracle is actively working on

Synthetic Datasets. We generate controlled synthetic 2-table join datasets with varying size parameters to control the redundancy ratio. We always fix n_S and d_S and vary TR and FR as specified in the experiments. Recall the definitions from Section 2.5.

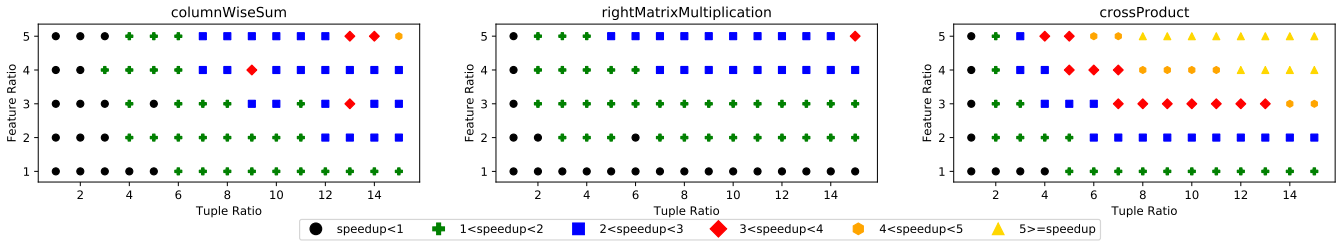

Figure 7: Discretized speedups of *Trinity* over *Materialized* in FastR.

Table 2: Real-world dataset statistics

Dataset	(n_S, d_S)	# *tbs*	(n_{R_i}, d_{R_i})	*RR*
Expedia	(942142, 27)	2	(11939, 12013) (37021,40242)	4.5
Movies	(1000209, 0)	2	(6040, 9509) (3706,3839)	12.7
Yelp	(215879, 0)	2	(11535, 11706) (43873,43900)	7.5
Walmart	(421570, 1)	2	(2340, 2387) (45,53)	5.9
LastFM	(343747, 0)	2	(4099, 5019) (50000,50233)	4.4
Books	(253120, 0)	2	(27876, 28022) (49972,53641)	2.3
Flights	(66548, 20)	3	(540, 718) (3167,6464) (3170,6467)	4.8

Real-world Datasets. We reuse all 7 real-world normalized datasets from the Morpheus paper; Table 2 lists the statistics. We also report the redundancy ratio (RR) alongside for more context; recall that RR is the ratio of the size of T against the size of the Normalized Matrix. We use FastR's `object.size` utility to estimate memory footprints for sizes. Most LA ops and LA-based ML algorithms have runtime complexity linear in the matrix sizes; Cross-Product and OLS linear regression are the only ones here that have runtimes quadratic in the number of features.

LA-based ML Algorithms and Parameters. We show results for all 4 LA-based ML algorithms from the original Morpheus paper: Linear Regression (*LinReg*), Logistic Regression (*LogReg*), K-Means Clustering (*KMeans*), and GNMF Clustering (*GNMF*). The number of iterations is set to 20 for all three iterative algorithms; number of centroids is 10 for *KMeans*; rank parameter is 5 for *GNMF*.

Experimental Setup. All experiments were run on a c8220 CloudLab node, equiped with Two Intel E5-2660 v2 10-core CPUs at 2.20 GHz, 256GB GB RAM, and over 1TB disk, running Ubuntu 16 as the OS. Unless otherwise stated, we warm up for 10 runs and report the *mean runtimes* of the following 15 runs.

6.1 LA Operator-level Results in FastR

We first evaluate Trinity's efficiency at the LA op level using synthetic data. This controlled experiment will shed light on interpreting the results with the ML algorithms and real-world datasets later. We fix $n_S = 10^5$ and $d_S = 20$, and vary TR and FR in $[1, 15]$ and $[1, 5]$, respectively. We plot the runtime speedup of Trinity against *Materialized*. We *exclude* the time to materialize the join, which might favor Materialized. Due to space constraints, we only show 3 LA ops here–ColumnWiseSum, RMM, and CrossProd–and present the rest in the Appendix. Figure 7 shows the results. Other LA ops show similar trends.

We see that Trinity is faster than Materialized for almost all TR-FR combinations that result in substantial data redundancy. As expected, the speedups grow as TR and/or FR grow, i.e., as the redundancy ratio grows. ColWiseSums sees slowdowns at lower TRs because the relative per-column overheads of columnar summation is higher with low numbers of rows. RMM has lower speedups that ColWiseSums at higher TRs due to the relatively higher overhead of its more complex rewrite rule. As expected CrossProd sees the highest speedups, even above 5× in many cases.

6.2 ML Algorithm-level Results in FastR

We now compare Trinity's efficiency against Materialized on the 4 LA-based ML algorithms. We also compare Trinity's speedups against that of MorpheusR, which is PL-specific, to assess the cost of Trinity's much higher generality. Table 3 shows the results on all the real-world datasets.

LogReg sees some of the highest speedups with both Trinity and MorpheusR on all datasets. Trinity is 8× faster than Materialized on *Movies*; 5× on *Yelp*. *LinReg* is a close second on speedups seen, 7× on *Movies* and 4× on *Yelp*. *Flights* sees the only example of *slowdowns* relative to Materialized, albeit only about 15% slower. In the original Morpheus paper [23] *Flights* did not see slowdowns but it did see among the lowest speedups. Since it has 3 joins and not much absolute total FLOPS, the overheads of rewrite rules dominate. Besides, FastR is also a highly optimized R runtimes; so, Materialized fares better in relative terms on this dataset.

KMeans sees 5× speedup on *Movies*, 3× on *Yelp*, and the lowest of 1.3× on *Flights*. *GNMF* sees slight slowdowns across the board. There are two reasons for this behavior. First, these datasets have relative low absolute total FLOPS due to their smaller sizes, which again cause rewrite overheads to matter. Even PL-specific MorpheusR had such issues in some cases [23, 37]. Second, and more pressing for GraalVM developers, is that standard GNU-R optimization wisdom does not directly apply to FastR because they are different implementations with different cost models. More concretely, addition of matrices in the *Matrix* library seemed to dominate the runtimes of *GNMF* on both Trinity and MorpheusR. But that was not the case with GNU-R. This overhead overshadowed the runtime gains from the rewrite rules in both MorpheusR

Table 3: Mean runtimes (in seconds) for Materialized (M), median speedups of Trinity relative to M (S_T), median speedups of MorpheusR relative to M (S_P). M, Y, W, F, E, L, and B refer to Movies, Yelp, Walmart, Flights, Expedia, LastFM, and Books, respectively. RR is their redundancy ratio as per Table 2.

		logReg			linReg		
	RR	M	S_T	S_P	M	S_T	S_P
M	12.7	165.07	8.13	8.56	132.66	7.49	7.55
Y	7.5	46.59	5.43	6.2	40.76	4.22	4.87
W	5.9	47.22	4.44	4.53	32.66	3.97	4.08
F	4.8	8.2	1.93	1.98	4.19	0.87	0.95
E	4.5	168.54	4.74	4.96	121.39	3.16	3.41
L	4.4	35.05	3.33	3.71	23.63	2.04	2.23
B	2.3	23.37	2.24	2.48	15.72	1.23	1.41

		KMeans			GNMF		
	RR	M	S_T	S_P	M	S_T	S_P
M	12.7	358.13	4.98	5.05	256.78	0.88	0.79
Y	7.5	109.8	3.22	3.29	68.8	0.82	0.85
W	5.9	95.03	3.1	3.15	97.18	0.8	0.8
F	4.8	20.46	1.37	1.44	21.64	0.81	0.8
E	4.5	384.51	1.38	1.39	265.8	0.66	0.68
L	4.4	85.56	2.11	2.07	101.06	0.84	0.84
B	2.3	66.31	1.47	1.49	82.85	0.84	0.82

and Trinity. In future work, we might to automatically capture the performance of different datatype-op configurations and translate those in MatrixLib adapters to cast to the right representation at the right time; currently our implementation depends on knowing these *a priori* from a *GraalVM PL developer* implementing the MatrixLib adpater.

Trinity vs MorpheusR. Across the board, Trinity mostly matches MorpheusR on speedups and is only about 20% slower in the worst case. In fact, Trinity is slightly faster than MorpheusR for *KMeans* on *LastFM*. We take all this as evidence that *Trinity*'s higher generality does not exact too high a price on efficiency. We compare the runtimes of Trinity and MorpheusR for some LA as well in the technical report [33]. The takeaways are largely similar to the above discussion; we skip the details here due to space constraints.

6.3 Other Host PLs and Polyglot Execution

Finally, we demonstrate Trinity's generality with two additional experiments: a factorized execution in GraalJS, as well as a polyglot factorized execution straddling GraalPython NumPy and FastR.

6.3.1 Factorized Execution in GraalJS. Due to space constraints: we present results only for *LinReg*. We fix $n_S = 10^4$ and $d_S = 20$, $TR = 10$, and vary FR from 1 to 5. We reduced n_S compared to Section 6.1 because model training took much longer in these experiments than in pure FastR. Table 4 presents the results.

Even though we see a single slowdown with *Trinity* when the feature ratio is 1, we see that the speedups of *Trinity* go up as FR increases. This validates that Trinity is able to successfully factorize the LA script execution in Javascript as well automatically using the same *same* underlying optimization and runtime infrastructure that FastR used. The speedups are lower than the corresponding FastR speedups, however, because GraalVM's support for Javascript is in early stages infancy. We expect this issue to get resolved as GraalVM matures.

Table 4: Runtimes (in seconds) on GraalVM for GraalJS and GraalPython+FastR.

	GraalJS		GraalPython + FastR	
FR	M	S_T	M	S_T
1	283.72	0.89	894.42	9.75
2	515.32	1.47	1353.53	14.85
3	584.89	1.58	1788.76	19.24
4	723.11	1.75	2041.92	21.85
5	873.49	2.02	2688.39	28.66

6.3.2 Polyglot GraalPython + FastR Execution. To demonstrate that Trinity works with GraalVM's polyglot capabilities, we run the *LinReg* LA script written in NumPy operating over matrices loaded in FastR. Before being exported to GraalPython, FastR matrices are wrapped in a MatrixLib adapter and then when received in GraalPython, they are wrapped in an adapter that exports the NumPy interface from the MatrixLib adapter. One can also do other pairs of host PLs but we chose this combination because GraalPython is still in active developmental stage and is not as good as FastR at supporting memory-intensive operations.

We use the same synthetic data setup as for GraalJS but instead have only 2 warm-up rounds and average over the next 5 training loops. We did this because GraalPython was unstable in our experiments and would *sometimes* fail with a segmentation fault when running for too long. Table 4 shows the results.

The speedup behavior here is strangely dramatic, growing from 9× for $FR = 1$ to 28× for $FR = 5$. While monotonic growth is to be expected, these numbers are far above the commensurate RR. For instance, $FR = 5$ leads to no more than $RR = 6$. That means 28× speedup clearly did *not* come from just the FLOPS savings of our rewrite rules. We believe this anomaly is due to GraalPython's current instability for memory-intensive workloads, which sometimes manifests in unexpected segmentation faults and may lead to overzealous garbage collection pauses or other background interference. Under these circumstances, the Materialized approach, which allocates much more memory than Trinity's Normalized Matrix, would obviously suffer from greater memory interference. Although GraalPython was unable to allocate larger matrices, these results still demonstrate that using Trinity and *MatrixLib* can enable a host PL to reap the benefits of factorized ML benefits by utilizing another PL's matrix representation.

6.4 Current Limitations

We have built a first-of-its-kind working prototype for a polyglot framework for factorized ML. We made a few assumptions for tractability sake. We recap our current major limitations.

(1) We chose GraalVM as our polyglot runtime; so, Trinity is tied to its host PLs and issues. But to the best of our knowledge, GraalVM is unique in its generality.

(2) We focused on star schemas (and snowflakes reduced to stars). But nothing in Trinity prevents generalizing it to support M:N joins or other join schemas like in MorpheusR [23].

(3) We target LA-based statistical/ML algorithms like Morpheus, not tree-based methods or deep learning. As such, deep learning requires GPU runtimes. We leave it to future work to extend Trinity to support tree-based ML too. Likewise, we leave it to future work to expand MorpheusDSL to add rewrite rules for non-linear operations such as feature interactions from MorpheusFI [37].

(4) Both Morpheus and MorpheusFI found that factorized execution is slower at very low TR and/or FR and proposed simple cost models to threshold on these to decide when to use Materialized instead. Trinity currently does not support such cost models; the user is expected to handle them out of band, e.g., before deciding to construct the Normalized Matrix. Additionally, and as we explained for the *GNMF* results, the runtimes of a *Matrix* adapter implementation is constrained by how well the author understands the PL's datatype-to-op runtime cost. It may be possible to automate such cost models in MorpheusDSL using GraalVM's more advanced capabilities; we leave such extensions to future work.

7 RELATED WORK

Factorized ML and LA. We extend a recent line of work in the DB world on factorized ML, which we split into 3 groups based on setting and target workloads: specific ML algorithms [34, 46, 49], in-RDBMS execution [20, 35, 48], and LA systems [23, 32, 36, 37]. Our work is *complementary* to all these prior works and builds on their ideas. The novelty of our Trinity is in its *generality*: it is the first to support factorized ML style ideas in a *polyglot* setting. Our work enables such novel DB+ML query optimization ideas to be *implemented once* but made available to multiple PL/LA systems *in one go*. While this paper focused on rewrite rules from Morpheus [23], our approach is generic enough to allow future work to easily augment MorpheusDSL with more rewrite rules from these other works above. Overall, we believe Trinity's generality can empower more such ideas from the DB+ML systems world to be adopted into industrial data science products such as GraalVM and ultimately, help benefit real-world data science users.

DSLs for ML and LA. There is much prior work on these. We discuss a few closely related exemplars from the PL/compilers worlds. OptiML and the larger Delite framework [22, 52, 53] optimize high-level ML code to achieve high performance and parallelism on specialized hardware. Diesel [26] exposes a high-level language for LA and neural network construction that is efficiently compiled for GPUs using polyhedral compilation techniques. In GraalVM, the grCuda DSL exposes GPUs to a *host* PL, allowing data scientists to invoke GPU kernels with ease [7, 8]. All these DSLs and systems are *complementary* to our work, since they focus on *physical data independence*, while Trinity focuses on *logical data independence* for multi-table datasets. ParallelJulia is a numerical computation embedded DSL (eDSL) providing a compiler-based approach for optimizing array-style Julia code [21]. Conceptually, we build on their principle of *non-invasive* DSLs by designing Trinity to require only as few visible changes to the host PL's programming models as possible. IFAQ [50] develops an end-to-end optimizing DSL for ML over relational data. But their approach depends on programming with the new IFAQ DSL, while we chose to augment *pre-existing* LA systems and PLs to reduce developability overhead.

Interoperability in Multi-PL VMs. We build on many years of interoperability tooling research on GraalVM [40, 47]. That said, other language VMs offer varying degrees of language interoperability such as .NET, via its Common Language Runtime, and the Parrot VM [1, 14, 39]. JVM-based languages, such as Scala, Kotlin, Clojure, often expose Java interoperability primitives as a selling point [38]. Other JVM-based language re-implementations such as JRuby or Jython have access to similar benefits [11, 41].

Extensible runtimes with domain-aware optimizations There is much work on extending runtimes and compiler frameworks with domain-aware optimizations [27, 42, 54, 57]. In the case of GraalVM, Trinity is, to the best of our knowledge, the first contribution of this kind to this community. We develop new abstractions on top of their interoperability service to pave the way towards more polyglot, but domain-aware, optimization techniques. It is also due to GraalVM that we did not need to develop a custom IR for data analytics such as in Weld [44, 45]. Still, Weld is also complementary to our work because its IR can also be encoded as a GraalVM language and combined with Trinity's optimizations that focus specifically on factorized LA/ML.

8 CONCLUSION AND FUTURE WORK

Factorized ML techniques help reduce ML runtimes over normalized data. But all implementations of such techniques so far are tied to one specific ML/LA system in one particular PL. This makes it highly tedious to reap the benefits of factorized ML across different LA systems and PLs in the fast-growing polyglot data science arena, since each PL/LA system may require its own extensive, cumbersome, and costly manual development effort. We take a first step towards mitigating this developability challenge by representing key factorized LA/ML rewrite rules as an emebddable DSL in a cutting-edge industrial polyglot virtual machine, GraalVM. Our system, Trinity, is a first-of-its-kind PL-agnostic and LA system-agnostic implementation of Morpheus, a prior factorized LA framework. In doing so, Trinity supports 3 axes of generality–multiple statistical/ML algorithms, multiple PLs/LA systems, and several rewrite optimizations–all in one unified framework. Experiments with many normalized datasets show Trinity is often significantly faster than materialized execution and largely matches the efficiency of a prior single PL-specific tool, MorpheusR.

As research at the intersection of DB, PL, and ML systems grows, we believe Trinity offers a helpful platform for researchers devising new cross-algebraic optimization techniques to more easily transfer their ideas to industrial data science users across PLs with much less manual development effort. In future work, apart from relaxing the limitations in Section 6.4, we aim to explore code-generation in Trinity to bypass polyglot-induced overheads. Another avenue is to optimize other end-to-end data science workloads beyond ML training, e.g., data preparation and ML debugging.

REFERENCES

[1] [n.d.]. Common Language Runtime (CLR) overview - .NET Framework. https://docs.microsoft.com/en-us/dotnet/standard/clr. Accessed: 2020-03-01.

[2] [n.d.]. Embed Languages with the GraalVM Polyglot API. https://www.graalvm.org/docs/reference-manual/embed/. Accessed: 2020-03-01.

[3] [n.d.]. Fallback (GraalVM Truffle Java API Reference). https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/Fallback.html. Accessed: 2020-03-01.

[4] [n.d.]. FastR GitHub Repository. https://github.com/oracle/fastr/. Accessed: 2020-03-01.

[5] [n.d.]. GraalJS GitHub Repository. https://github.com/graalvm/graaljs. Accessed: 2020-03-01.

[6] [n.d.]. GraalVM Python Implementation GitHub Repository. https://github.com/graalvm/graalpython. Accessed: 2020-03-01.

[7] [n.d.]. grCuda Documentation. https://github.com/NVIDIA/grcuda/blob/master/docs/language.md. Accessed: 2020-03-01.

[8] [n.d.]. grCuda GitHub Repository. https://github.com/NVIDIA/grcuda. Accessed: 2020-03-01.

[9] [n.d.]. Interactive Matrix Programming With SAS IML Software. https://www.sas.com/en_us/software/iml.html. Accessed: 2020-03-01.

[10] [n.d.]. InteropLibrary (GraalVM Truffle Reference). https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/interop/InteropLibrary.html. Accessed: 2020-03-01.

[11] [n.d.]. Jython Project Homepage. https://www.jython.org/. Accessed: 2020-03-01.

[12] [n.d.]. Math.js Project Homepage. https://mathjs.org/. Accessed: 2020-03-01.

[13] [n.d.]. MATLAB Homepage. https://www.mathworks.com/products/matlab.html. Accessed: 2020-03-01.

[14] [n.d.]. ParrotVM Documentation - HLLs and Interoperation. http://docs.parrot.org/parrot/latest/html/docs/book/draft/chXX_hlls.pod.html. Accessed: 2020-03-01.

[15] [n.d.]. The R Project for Statistical Computing. https://www.R-project.org/. Accessed: 2020-03-01.

[16] [n.d.]. SimpleLanguage GitHub Repository. https://github.com/graalvm/simplelanguage/blob/master/language/src/main/java/com/oracle/truffle/sl/nodes/expression/SLAddNode.java. Accessed: 2020-03-01.

[17] [n.d.]. Specialization (GraalVM Truffle Java API Reference). https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/Specialization.html. Accessed: 2020-03-01.

[18] [n.d.]. TruffleLibraries Documentation. https://github.com/oracle/graal/blob/master/truffle/docs/TruffleLibraries.md. Accessed: 2020-03-01.

[19] [n.d.]. Walnut Project Homepage on Oracle Labs. https://labs.oracle.com/pls/apex/f?p=LABS:project_details:0:15. Accessed: 2020-03-01.

[20] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. In-Database Learning with Sparse Tensors. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Houston, TX, USA) *(SIGMOD/PODS '18)*. Association for Computing Machinery, New York, NY, USA, 325–340. https://doi.org/10.1145/3196959.3196960

[21] Todd A. Anderson, Hai Liu, Lindsey Kuper, Ehsan Totoni, Jan Vitek, and Tatiana Shpeisman. 2017. Parallelizing Julia with a Non-Invasive DSL. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs)*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.4

[22] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. 2011. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, Calin Cascaval and Pen-Chung Yew (Eds.). ACM, 35–46. https://doi.org/10.1145/1941553.1941561

[23] Lingjiao Chen, Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. 2017. Towards Linear Algebra over Normalized Data. *PVLDB* 10, 11 (2017), 1214–1225. https://doi.org/10.14778/3137628.3137633

[24] Lin Clark. [n.d.]. WebAssembly Interface Types: Interoperate with All the Things! – Mozilla Hacks - the Web developer blog. https://hacks.mozilla.org/2019/08/webassembly-interface-types/. Accessed: 2020-03-01.

[25] Michael L. Van de Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. 2018. Fast, Flexible, Polyglot Instrumentation Support for Debuggers and other Tools. *CoRR* abs/1803.10201 (2018). arXiv:1803.10201 http://arxiv.org/abs/1803.10201

[26] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. 2018. Diesel: DSL for linear algebra and neural net computations on GPUs. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Justin Gottschlich and Alvin Cheung (Eds.). ACM, 42–51. https://doi.org/10.1145/3211346.3211354

[27] Grégory M. Essertel, Ruby Y. Tahboub, Fei Wang, James M. Decker, and Tiark Rompf. 2019. Flare & Lantern: Efficiently Swapping Horses Midstream. *Proc. VLDB Endow.* 12, 12 (2019), 1910–1913. https://doi.org/10.14778/3352063.3352097

[28] Michael Furr and Jeffrey Foster. 2008. Checking type safety of foreign function calls. *ACM Trans. Program. Lang. Syst.* 30 (07 2008). https://doi.org/10.1145/1377492.1377493

[29] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Mikel Luján. 2018. Cross-Language Interoperability in a Multi-Language Runtime. *ACM Trans. Program. Lang. Syst.* 40, 2 (2018), 8:1–8:43. https://doi.org/10.1145/3201898

[30] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance cross-language interoperability in a multi-language runtime. In *Proceedings of the 11th Symposium on Dynamic Languages, DLS 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Manuel Serrano (Ed.). ACM, 78–90. https://doi.org/10.1145/2816707.2816714

[31] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance Cross-language Interoperability in a Multi-language Runtime. *SIGPLAN Not.* 51, 2 (Oct. 2015), 78–90. https://doi.org/10.1145/2936313.2816714

[32] Dylan Hutchison, Bill Howe, and Dan Suciu. 2017. LaraDB: A Minimalist Kernel for Linear and Relational Algebra Computation. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond* (Chicago, IL, USA) *(BeyondMR'17)*. Association for Computing Machinery, New York, NY, USA, Article 2, 10 pages. https://doi.org/10.1145/3070607.3070608

[33] David Justo, Lukas Stadler, Nadia Polikarpova, and Arun Kumar. [n.d.]. *Towards A Polyglot Framework For Factorized ML*. Technical Report. Tech. rep. https://adalabucsd.github.io/papers/TR_2020_Trinity.pdf.

[34] Arun Kumar, Mona Jalal, Boqun Yan, Jeffrey Naughton, and Jignesh M. Patel. 2015. Demonstration of Santoku: Optimizing Machine Learning over Normalized Data. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1864–1867. https://doi.org/10.14778/2824032.2824087

[35] Arun Kumar, Jeffrey Naughton, and Jignesh M. Patel. 2015. Learning Generalized Linear Models Over Normalized Data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1969–1984. https://doi.org/10.1145/2723372.2723713

[36] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breundefined, Tilmann Rabl, and Volker Markl. 2019. An Intermediate Representation for Optimizing Machine Learning Pipelines. *Proc. VLDB Endow.* 12, 11 (July 2019), 1553–1567. https://doi.org/10.14778/3342263.3342633

[37] Side Li, Lingjiao Chen, and Arun Kumar. 2019. 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*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1571–1588. https://doi.org/10.1145/3299869.3319878

[38] Wing Hang Li, David Robert White, and Jeremy Singer. 2013. JVM-hosted languages: they talk the talk, but do they walk the walk?. In *PPPJ '13*.

[39] Todd M Malone. 2014. Interoperability in Programming Languages.

[40] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2019. Towards polyglot adapters for the GraalVM. In *Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming, Genova, Italy, April 1-4, 2019*. ACM, 1:1–1:3. https://doi.org/10.1145/3328433.3328458

[41] Charles O Nutter, Thomas Enebo, Nick Sieger, and Ian Dees. 2011. *Using JRuby: Bringing Ruby to Java*. Pragmatic Bookshelf.

[42] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. 2013. Spiral in scala: towards the systematic construction of generators for performance libraries. In *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, Jaakko Järvi and Christian Kästner (Eds.). ACM, 125–134. https://doi.org/10.1145/2517208.2517228

[43] Travis E Oliphant. 2006. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA.

[44] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proc. VLDB Endow.* 11, 9 (May 2018), 1002–1015. https://doi.org/10.14778/3213880.3213890

[45] Shoumik Palkar, J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, Saman P. Amarasinghe, M. Zaharia, and Stanford InfoLab. 2016. Weld : A Common Runtime for High Performance Data Analytics.

[46] Steffen Rendle. 2013. Scaling Factorization Machines to Relational Data. *Proc. VLDB Endow.* 6, 5 (March 2013), 337–348. https://doi.org/10.14778/2535573.2488340

[47] Alexander Riese, Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2020. User-Defined Interface Mappings for the GraalVM. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming* (Porto, Portugal) *('20)*. Association for Computing Machinery, New York, NY, USA, 19–22. https://doi.org/10.1145/3397537.3399577

[48] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. 2019. A Layered Aggregate Engine for Analytics Workloads. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1642–1659. https://doi.org/10.1145/3299869.3324961

[49] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning Linear Regression Models over Factorized Joins. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/2882903.2882939

[50] Amir Shaikhha, Maximilian Schleich, Alexandru Ghita, and Dan Olteanu. 2020. Multi-Layer Optimizations for End-to-End Data Analytics. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 145–157. https://doi.org/10.1145/3368826.3377923

[51] Daniel Smilkov, Nikhil Thorat, Yannick Assogba, Ann Yuan, Nick Kreeger, Ping Yu, Kangyi Zhang, Shanqing Cai, Eric Nielsen, David Soergel, Stan Bileschi, Michael Terry, Charles Nicholson, Sandeep N. Gupta, Sarah Sirajuddin, D. Sculley, Rajat Monga, Greg Corrado, Fernanda B. Viegas, and Martin Wattenberg. 2019. TensorFlow.js: Machine Learning for the Web and Beyond. Palo Alto, CA, USA. https://arxiv.org/abs/1901.05350

[52] Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embedded Comput. Syst.* 13, 4s (2014), 134:1–134:25. https://doi.org/10.1145/2584665

[53] Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2013. Forge: generating a high performance DSL implementation from a declarative specification. In *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, Jaakko Järvi and Christian Kästner (Eds.). ACM, 145–154. https://doi.org/10.1145/2517208.2517220

[54] Ruby Y. Tahboub and Tiark Rompf. 2020. Architecting a Query Compiler for Spatial Workloads. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2103–2118. https://doi.org/10.1145/3318464.3389701

[55] Anthony Thomas and Arun Kumar. 2018. A Comparative Evaluation of Systems for Scalable Linear Algebra-Based Analytics. *Proc. VLDB Endow.* 11, 13 (Sept. 2018), 2168–2182. https://doi.org/10.14778/3275366.3284963

[56] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* (2020). https://doi.org/10.1038/s41592-019-0686-2

[57] Fei Wang, Daniel Zheng, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. 2019. Demystifying differentiable programming: shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.* 3, ICFP (2019), 96:1–96:31. https://doi.org/10.1145/3341700

[58] Christian Wimmer and Thomas Würthinger. 2012. Truffle: a self-optimizing runtime system. In *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens (Ed.). ACM, 13–14. https://doi.org/10.1145/2384716.2384723

[59] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 662–676. https://doi.org/10.1145/3062341.3062381

[60] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. 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*, Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld (Eds.). ACM, 187–204. https://doi.org/10.1145/2509578.2509581

[61] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages, DLS '12, Tucson, AZ, USA, October 22, 2012*, Alessandro Warth (Ed.). ACM, 73–82. https://doi.org/10.1145/2384577.2384587

A FULL LA-OPERATOR RESULTS

Trinity's LA-Operator Results. We now discuss Trinity's operator-level speed-ups over the materialized approach for our full suite of operator benchmarks on synthetic data. The experimental settings are the same as in the evaluation section, and the results are visualized in Figure 8. We see that *scalarMultiplication*, *elementWiseSum*, and *crossProduct* all achieve substatial speed-ups going through the full-range from a slow-down towards about $4x$ or $5x$ speed-up as TR-FR grows. These three achieve the largest speed-ups.

Comparatively, *rightMatrixMultiplication* and *columnWiseSum* achieve modest speed-ups, with the majority of their performance bounded between a $1x$ to $3x$ speed-up. Finally, *rowWiseSum* and *LeftMatrixMultiplication* yield the least speed-ups, with *LeftMatrixMultiplication* being the least-performant. In our experiments, the performance of *LeftMatrixMultiplication* could be explained, in part, because the addition-step of the rewrite rule seemed too expensive, relative to the other intermediate steps of the rewrite rule, which differed from our experience in GNU-R. We believe that this is due to cost model differences between GNU-R and FastR. Finally, given that we do start seeing frequent speed-ups for FR=5, we believe that larger redudancy ratios for this operator would also yield more dramatic speed-ups.

MorpheusR's LA-Operator Results. Now we discuss MorpheusR's operator-level speed-ups over materialized, like we did for Trinity's results. The data is visualized in Figure 9. The trends for MorpheusR speed-ups closely follow the same pattern as Trinity; as we would expect. We see that *scalarMultiplication*, *elementWiseSum*, and *crossProduct* achieve the highest speed-ups, followed by *rightMatrixMultiplication* and *columnWiseSum*, and then by *rowWiseSum* and *leftMatrixMultiplication*.

Of notable difference is that *leftMatrixMultiplication* for MorpheusR does not achieve any speed-ups. This is not entirely surprising considering that Trinity also struggled to beat the materialized approach for this operator. Minor performance differences like this one can be explained by Trinity's modular design. Since each LA operator execution in Trinity is managed by the MatrixLib adapter, operator-level optimizations such as choosing the right matrix output representation given the input datatypes take place every time that operator is called. With MorpheusR we cannot do this as easily since each rewrite rule inlines its operator-level optimizations, meaning that there's room for missing optimization opportunities when coding these manually. Given more time, we could have found all these missed optimization opportunities to get MorpheusR to track Trinity's *leftMatrixMultiplication* more closely. Regardless, the fact that MorpheusR requires this level of extra care speaks to the developability wins of using Trinity.

Comparing MorpheusR directly with Trinity. Finally, we comment on the performance difference between MorpheusR and Trinity. Figure 10 shows the relative speed-ups of MorpheusR over Trinity; note that the legend has changed. At first glance, it's clear that MorpheusR is, in the vast mayority of cases, within $0.3x$ of Trinity's runtime both in terms of speed-ups and slow-downs. Additionally, we don't see this $0.3x$ difference manifest itself more often in speed-ups or in slowdowns: in most operators speed-ups and slow-downs interleave suggesting that both approaches perform quite similarly. The two exceptions to this observation are *scalarAddition* and *leftMatrixMultiplication*. In the former, MorpheusR appears to consistently outperform Trinity with a speed-up around $1.3x$ whereas in the latter MorpheusR was mainly within $0.5x$ and $0.7x$ of Trinity's runtime.

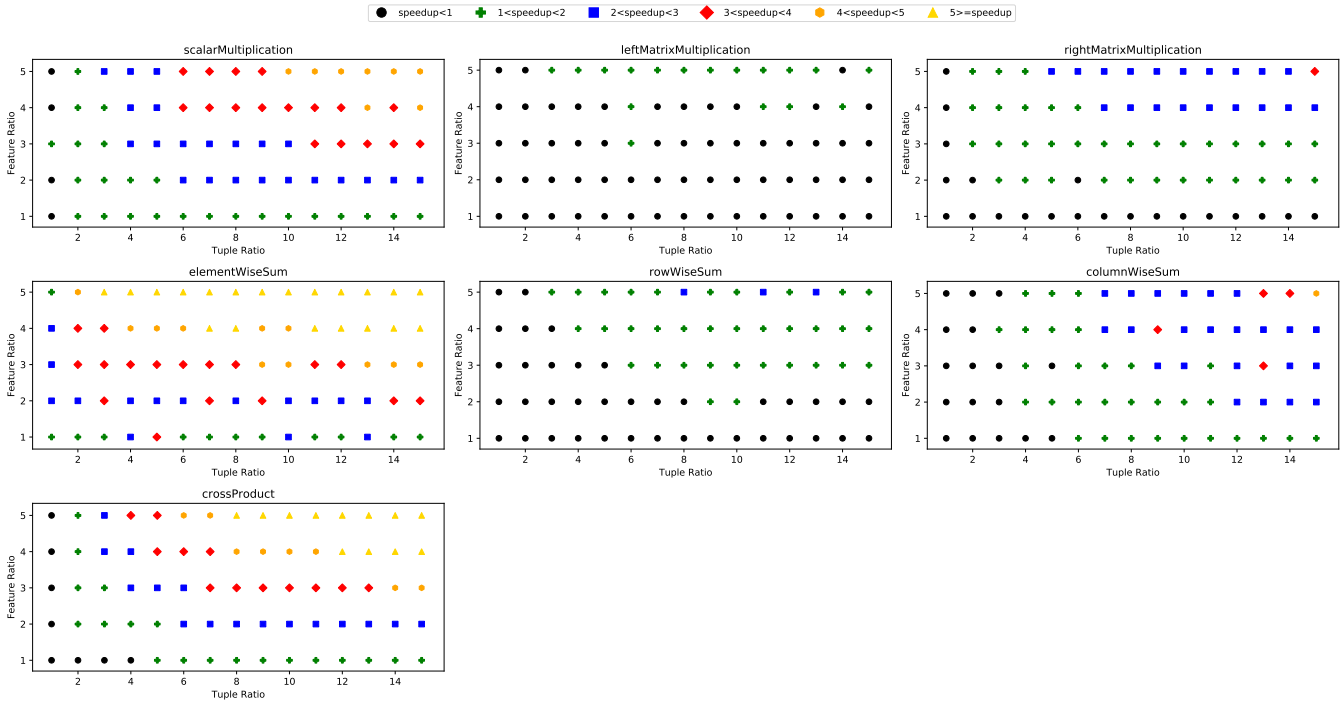

Figure 8: All discretized speedups of *Trinity* over *Materialized* in FastR.

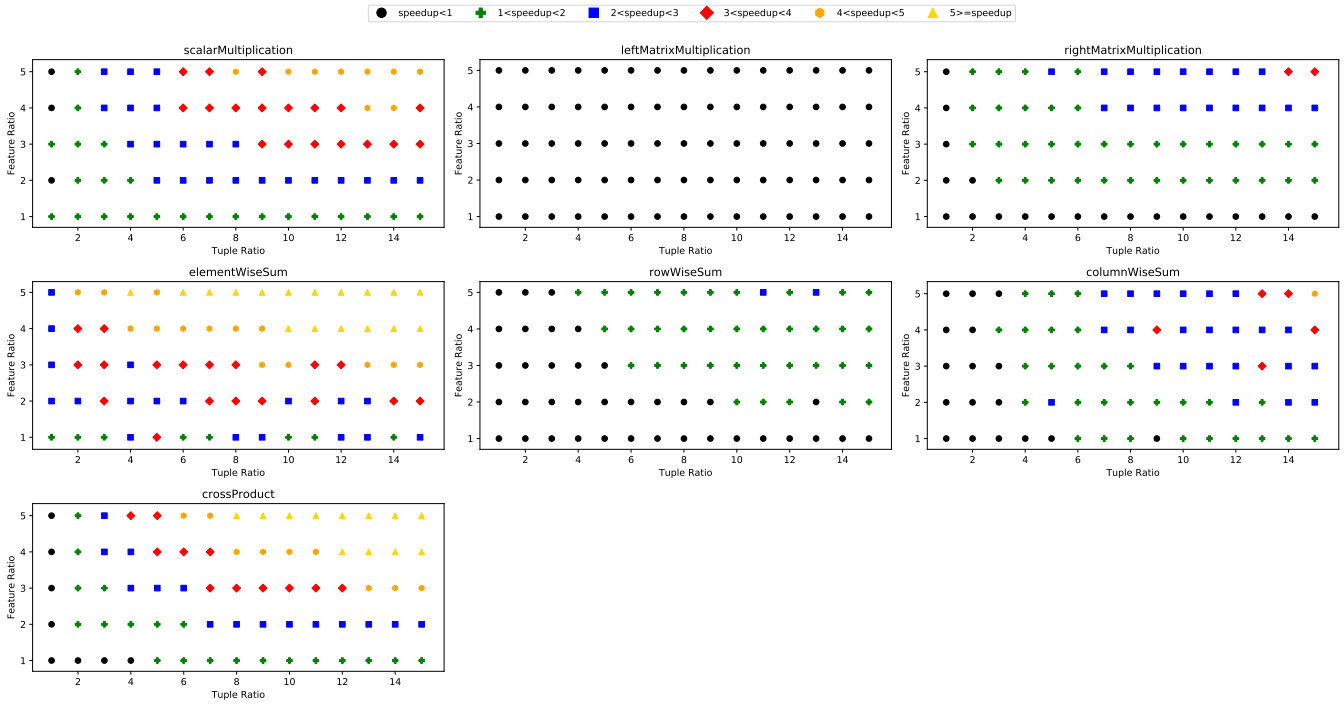

Figure 9: All discretized speedups of *MorpheusR* over *Materialized* in FastR.

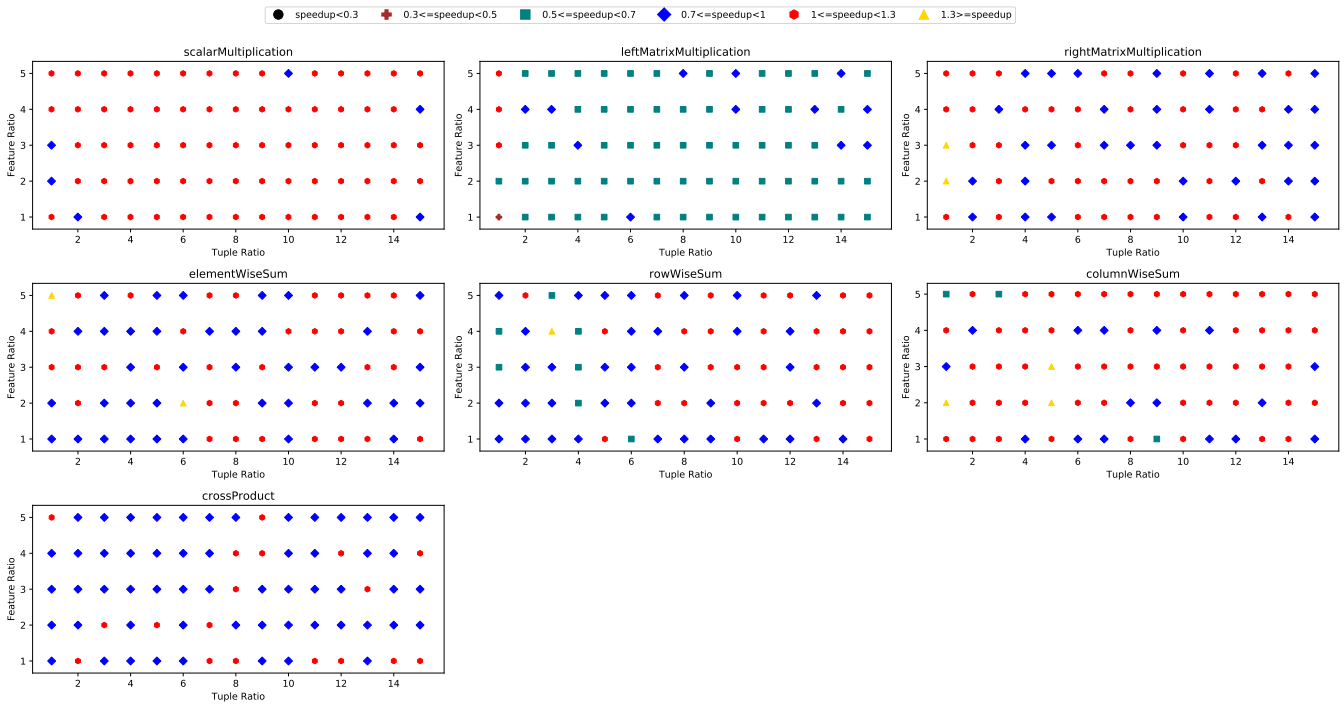

Figure 10: All discretized speedups of *MorpheusR* over *Trinity* in FastR.

B REVISITING GRAALPYTHON

As mentioned in the discussion of our GraalPython experiment, the Trinity speed-ups we observed could not be explained by the theory, so we suspected some additional factor had to contribute to those measurements. To find the culprit, we measured the duration of each individual LA operation in that experiment for both Trinity and Materialized in search for an abnormality.

The differentiating operation: the tranpose. We found that the dataset transpose operation in the *linReg* algorithm dominated the runtime of the Materialized approach. It took over 80% of the runtime in each training loop, which was unexpected. Recall that, for Trinity and MorpheusR, transposing the dataset matrix is simply changing a boolean flag that will trigger alternative rewrite rules, so it is virtually immediate, meaning that no Morpheus-based approach would suffer from this performance cost. As for why the transpose operation is so costly: it most likely requires allocating extra memory which, in our experience, is something GraalPython struggles with. As said before, we believe these issues will be mitigated as the platform continues to mature.

An alternative Linear Regression. To mitigate the performance cost of tranposing the dataset matrix, we slightly modified the *linReg* algorithm as shown in Algorithm 2. Do note that the algorithm for *linReg* is short enough to be expressed in one line, but we've split it into multiple to emphasize our change in the line highlighted in yellow and therefore introduced a new variable z to hold intermediate values. Our rewrite of *linReg* is equivalent to the canonical formulation, but avoids the cost of transposing the dataset matrix which we now know is too expensive.

Algorithm 2: LinReg Alternative

Input: X: the dataset
Output: w: the updated weights
$\gamma = 1x10^{-6}$
$nRow = getNumColsOf(X)$
$nCol = 1$
$w = genRandMatrix(nRow, nCol)$
for $k = 1$ *to* 20 **do**
  $z = (X * w) - y$
  `// Used to be` $z = X.T * z$
  $z = (z.T * X).T$
  $w = w - (\gamma * z)$
end
return w

Table 5: Runtimes (in seconds) on GraalVM for GraalPython+FastR with modified linReg.

	GraalPython + FastR	
FR	M	S_T
1	42.26	0.78
2	45.46	0.87
3	48.87	0.99
4	51.55	0.96
5	61.66	1.21

Using this variant algorithm, we ran our GraalPython experiment again, using the same configuration as before. Table 5 has the results. The results are much more modest this time around and somewhat

resemble the trend we observed in GraalJS. Trinity starts slower than Materialized at a $0.8x$ speed-up and slowly but monotonically climbs its way up to a $1.21x$ speed-up as FR increases. This follows our expectations, and suggests that at higher redudancy ratios and dataset sizes we would see larger speed-ups as well. This experiment confirms that our speed-ups in Table 4 were not primarily driven Morpheus rewrite rules, as we theorized. At the same time, now with the cost of the transpose operation mitigated, it also shows that Trinity's factorized rewrite continue to behave as expected, growing monotonically with higher redundancy ratios, regardless of host PL.