Towards A Polyglot Framework for Factorized ML (Information System Architectures)

David Justo
University of California, San Diego
djusto@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" help 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 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 novel information 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 polyglot compiler/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 ($>$ 10x speedups in some cases), while being largely competitive to a prior single PL-focused Morpheus stack.

PVLDB Reference Format:
David Justo, Lukas Stadler, Nadia Polikarpova, and Arun Kumar. Towards A Polyglot Framework for Factorized ML. *PVLDB*, 13(xxx): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Proceedings of the VLDB Endowment, Vol. 13, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

Optimizing machine learning (ML) workflows on structured data on various data platforms is a major focus for the database community and industry. 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 [36, 25, 46, 45]. This is a form of cross-algebraic query optimization bridging relational algebra and linear algebra (LA) [37].

Example. Consider a data scientist at Yelp building a recommender system to predict ratings of businesses by users. She has at least 3 data tables: **Ratings**, **Users**, and **Businesses**. **Ratings** has *Stars* as the prediction target and labels from the past. She performs this star schema join to gather features from all base tables for her ML analyses, e.g., feature extraction, training ML models, etc.

Factorized ML techniques *automatically* push ML computations down through the join. This can save substantial runtimes (even $>$ 10x speedups on real datasets [25]), memory/storage footprints, and ultimately, total resource costs, especially in public clouds. But there is a key practical bottleneck: *How to apply factorized ML ideas to the wide variety variety of ML algorithms used in practice?* This is a *developability* challenge. Rewriting ML implementations by hand is a highly tedious and error-prone process for data scientists to do manually. This limits the reach of factorized ML in practical data science use cases.

To meet the above challenge, the Morpheus project [25] initiated work on *generalized* factorized ML. It is rooted in one key observation: many statistical/ML programs are just bulk linear algebra (LA) scripts. That is, they represent models as matrices and manipulate datasets using matrix arithmetic. Of course, not all ML methods fit this template (e.g., decision trees or deep learning), but many popular ML methods do, including linear and logistic regression (the most popular models in practice [18]) and linear SVMs solved using batch gradient and second-order methods, k-means clustering, matrix factorization, and more [25, 23]. Thus, by "factorizing" a set of LA operations (e.g., matrix-matrix multiplication), Morpheus *automates factorized ML* for all algorithms expressible as LA scripts.

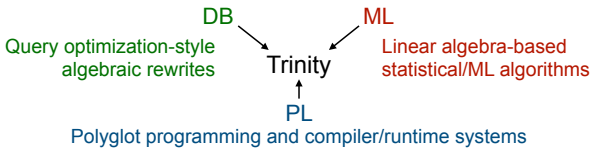


Figure 1: Trinity synthesizes and innovates upon concepts and techniques from the DB, ML, and PL communities.

Problem: Polyglot Data Science Landscape. However, the 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 [47]. All this PL diversity leads to a new practical bottleneck: *How to bring the benefits of factorized ML/LA to a wider variety of PLs/LA systems?* 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 [38]), prototyping them across PLs leads to tedious development grunt work for researchers.

In this paper, we take a first step towards a first-of-its-kind polyglot framework for factorized ML to meet the above developability challenges. We unify ideas from 3 fields: DB, ML, and PL, as Figure 1 illustrates. Specifically, we exploit an emerging polyglot compiler/runtime stack to architect a novel information system architecture that makes DB-style factorized ML/LA query optimization more widely available to LA-based statistical/ML analytics.

System Desiderata and Challenges. We have 3 key desiderata for such a polyglot framework. (1) *Generic availability*: Our framework 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 implementation environment. 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 cleaner separation of concerns. (3) *Efficiency:* Ideally, the higher generality we seek will not sacrifice too much performance relative to prior single PL-specific implementations.

Our Approach: Trinity. We present Trinity, the *first information system architecture 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 excellent support for PL interoperability [54]. We use GraalVM for two reasons. First, GraalVM lets us re-use 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

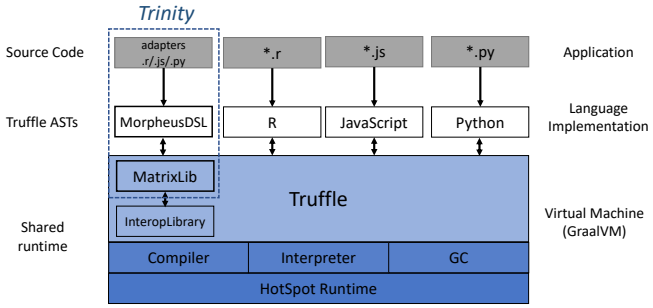


Figure 2: Trinity in and for GraalVM (Section 3 explains more). GraalVM supports many host PLs, they execute on a shared runtime Truffle, which provides services such as dynamic compilation and memory management. Truffle also provides interoperability services across PLs. We build two such services in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

span PLs. We believe such powerful capabilities are beneficial for data science workloads, since data scientists get more freedom to choose different PLs for writing LA scripts, representing and transforming data, and composing end-to-end workflows, all with a unified runtime. Figure 2 illustrates Trinity's high-level architecture.

Architecture and Summary of Techniques. Section 3 explains the usage and architecture in detail; we summarize them here. Trinity is designed to offer 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 rules). 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 generic availability. 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 generic availability 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.

Technical Novelty. Trinity applies cutting-edge PL/-compilers techniques to the DB+ML world. But we believe Trinity's key novelty is less in its individual techniques and more in its first-of-a-kind holistic system architecture that synthesizes the "right" techniques from disparate fields to meet our developability goals. Indeed, 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.

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 rewrite rules as a generic reusable service in a polyglot runtime efficiently.
- 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 real-world multi-table datasets. Overall, Trinity offers substantial speedups over materialized execution, even > 10x 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 2.1x .

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 [15, 43, 51, 12, 13, 9]. 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 that arise in ML algorithm scripts; this is the minimal set Trinity expects of an LA system. All popular LA systems offer such operators, 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.

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

[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 [25]. A notable exception is stochastic gradient descent, which needs external mini-batch sampling operations over the data [50].

Table 1: Minimal set of LA operators and functions for an LA system to work with Trinity. Notation: 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	$crossprod(T)$

2.2 GraalVM and Truffle

The GraalVM project aims to accelerate the development of programming languages by amortizing the cost of building language-specific VMs [54] GraalVM languages are implemented using Truffle [52, 55], an interpreter-writing framework, and execute on a modified version of the HotSpot VM named GraalVM [54]. This approach has led to the development of high-performance language re-implementations that include R, Python, JavaScript, and others. We refer to them as to as FastR, GraalPython, and GraalJS respectively [4, 6, 5]. For an architechtural overview of , and how Trinity fits into it, refer to Figure 2.

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 [53].

Example. Consider the implementation of a binary addition operator (`+`) for a simple language supporting number-to-number 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 [17, 3].

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 [31, 30]. In practice, this means that 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.

```
@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].

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 [32]. Listing 3 from a later discussion on interoperability exemplifies the ussage of this technology.

Polyglot Programs. As per the GraalVM literature, a polyglot program is one that composes functionality from multiple languages within the same application [27]. 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 [25] 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 $\mathbf{R}$.

```
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.

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. Table 2 summarizes our matrix notation.

Table 2: Matrix notation used in this paper.

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

2.5 Background on Morpheus

Normalized Matrix. The Normalized Matrix is a logical LA datatype introduced in Morpheus [25]. 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 serves to denormalize 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 the covers, Morpheus *automatically* rewrites ("factorizes") LA operators over T_N into LA operators over S, K, and R. In this sense, Morpheus brings the classical database 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 the Morpheus paper [25] for the entire set of rewrite rules.

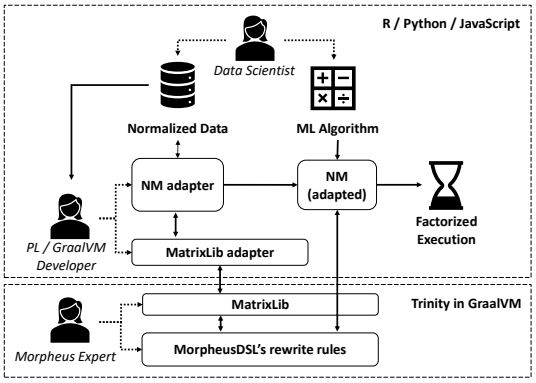


Figure 3: Usage of Trinity. Dotted arrows show what each user provides/maintains in the system. We use NM as short for Normalized Matrix. Solid arrows show the interaction between Trinity's components. The Morpheus expert and PL/GraalVM Developer must work together to expand support to new host PLs in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

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

The first rewrite rule shows element-wise scalar addition/-multiplication/etc. ($\oslash$ is from Table 1). The second shows full matrix 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 rewrite rules reduce runtimes by reducing the raw number of FLOPS for LA operators on normalized data; recall that T has join-induced data redundancy. Thus, "factorized" execution has asymptotically lower runtime complexity than the materialized single-matrix version. All complexity expressions are given in [25], but we discuss 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 quantify possible runtime speedups, Morpheus defines two quantities: *tuple ratio* (TR), defined as n_S/n_R and *feature ratio* ((FR), defined as d_R/d_S [25]. Since many LA operators are linear in the data size (Cross-Product is an exception), the *redundancy ratio* (RR) was defined to capture the FLOPS reduction. RR is the ratio of the size of T to the sum of sizes of input matrices. So, as TR and/or FR go up, RR also goes up and Morpheus becomes faster than materialized. But if TR is 1, RR is also 1, and there will be no speedups. As our experiments will show, many real-world datasets have non-trivial RR and thus benefit from Morpheus's 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, since MatrixLib abstracts the idiosyncrasies of each PL/LA system into a uniform interface for us. 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 would *adapt* the object to conform to the appropriate matrix interface in the host PL/LA system. All LA operators requested on this adapted object would go through the adapter, which takes care of adapting any further inputs and sends them to MorpheusDSL for the rewrite rules to execute. The rewrite rules can execute because they interface with the adapters, which expose a generic interface over matrices.

3.2 Using and Extending Trinity

User Roles. Recall that Trinity has a clean separation of concerns between a Data Scientist, a PL/GraalVM Developer, and a Morpheus Expert. We now explain their usage or interaction with Trinity a bit more. This discussion highlights 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 *multi-lingual scripts* spanning these PLs also benefit likewise. This high level of generality for automating factorized ML was *not possible before* Trinity.

Normalized Matrix 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 auta-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. We remark that MatrixLib's utility actually extends beyond Trinity and factorized ML. It is general enough to define other interoperable procedures or optimizations for LA systems. For instance, a GraalVM Developer can add Matlab-style optimization for deciding ordering of matrix chain multiplications, a classic LA optimization. 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 by using MatrixLib objects. They can then ask the PL/GraalVM Developer to update the Normalized Matrix adapter (or do so themselves) so that the new rewrites can be used in applicable situations.

Limitation: More LA Operators. We remark that one limitation in our setup arises when MatrixLib itself needs changes, e.g., adding novel rewrite rules for *other* LA operators beyond Table 1. In this case, the Morpheus Expert must work closely with the PL/GraalVM Developer to ensure that the new LA operators 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 [32, 29, 26]. 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 [53]. (2) It already exposes a foundation of interoperability APIs that works across all host PLs [32]; 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 [20].

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. The rationale for it 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, dataframes, 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 [19].

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

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 1-4 set up the node to use MatrixLib. The rest uses MatrixLib to uniformly invoke the scalar-multiplication operator for any input matrix.

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 *Adapters*, which guarantee 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 [19].

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 adapted inputs and returns adapted matrices. This let us compose generic sequences of LA operations.

5. EMBEDDABLE MORPHEUS DSL

```
1  class MatrixLibAdapter {
2    constructor(matrix) {
3        this.matrix = matrix;
4    }
5    scalarAddition(number) {
6        let res = math.add(this.matrix, number);
7        return new MatrixLibAdapter(res);
8    }
```

Listing 5: Preview of an example MatrixLib adapter for JavaScript.

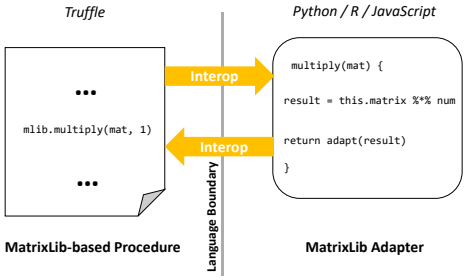


Figure 4: MatrixLib is abbreviated as mlib. Suppose the matrices' host uses $\% * \%$ for matrix multiplication (like R syntax). MatrixLib crosses the language boundary every time an LA operation is invoked. Built on top of Truffle's basic Interoperability API, it relies on adapters exporting a unified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

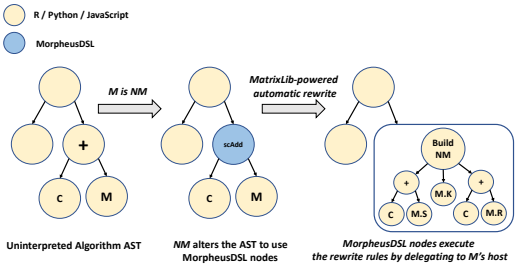


Figure 5: MorpheusDSL dynamically factorizes ML algorithms. C refers to some numeric constant, M refers to some matrix datatype and NM refers to the adapted Normalized Matrix. The rewrite occurs by using the host language's own LA operator implementations

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.

No Syntax. MorpheusDSL has no distinctive syntax; all expressions in this language evaluate to returning a constructor for a Normalized Matrix. So, this language effectively serves as a *guest* in a *host* PL. If the constructor is called with the *adapted* base table matrices, it returns a Normalized Matrix datatype. Again, its interface will likely not match the host PL's expected matrix interface; so it will be adapted again before it can be used in a pre-existing LA-based ML algorithm implementation.

AST Rewrite. 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 Implementation

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 provide it the appropriate inputs: entity table matrix S, an array of indicator matrices K's, and an array of attribute table matrices R's. Listing 6 illustrates this. Listing 7 previews how the Javascript Adapter re-exports the host PL's matrix interface from Trinity's generic MatrixLib interface.

```
1  // assume S, K, R are pre-existing math.js matrices
2  S = new MatrixLibAdapter(S);
3  K = [new MatrixLibAdapter(K)];
4  R = [new MatrixLibAdapter(R)];
5  // Obtain the Normalized Matrix constructor
6  let constructor = Polyglot.eval("morpheusDSL", "");
7  let normMatrix = constructor.build(S, K, R);
8  // now we can use the normalized matrix, ex:
9  let normMatrix.elementWiseSum(); // returns sum
```

Listing 6: Obtaining a Normalized Matrix in JavaScript

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.

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 show walk

```
  const multiply = math.typed('multiply', {
    'Matrix, NormMatrix': function (a, b) {
        let arg = new MatrixLibAdapter(a);
        let res = b.innerMatrix.
          rightMatrixMultiplication(arg);
        return res.unwrap();
    },
    //...
});
```

Listing 7: A preview of the Normalized Matrix adapter in Math.JS. Note that inputs to MorpheusDSL are always adapted first, and that the output of the rewrite needs to remove the MatrixLib adapter.

```
@Specialization(limit="3", ... // shortened
Object doDefault(NormalizedMatrix receiver,
 Object num, @CachedLibrary("receiver.S")
 MatrixLibrary matrixlibS,@CachedLibrary(limit="3")
 MatrixLibrary matrixlibGen)
 throws UnsupportedMessageException {

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

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.

through how various Trinity components interact when performing factorized execution. Figure 6 illustrates this. For exposition sake, we assume FastR is the host PL. Assume we have already constructed a Normalized Matrix and 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. This leads us 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 first MatrixLib call from Figure 8 leads to a MatrixLib-conforming Adapter of S to perform addition on it as implemented in Listing 5. Note that Listing 5 is an Adapter in a PL different than FastR–this is possible because MatrixLib abstracts over the matrices' PL of origin and can manipulate *any* matrix as longs as its Adapter maps between MatrixLib messages and the wrapped matrix's host PL/LA system. After executing the rewrite rule's steps, Trinity returns a new Normalized Matrix reference to

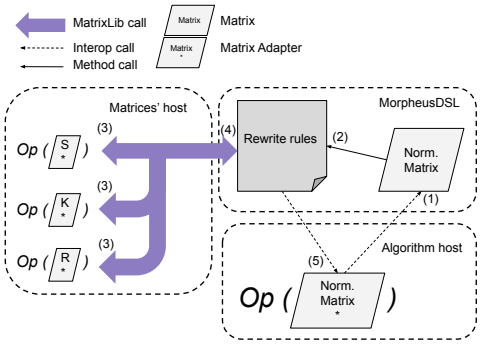


Figure 6: How Trinity's components interact for factorized execution. (1) Host PL invokes an LA operator from the Normalized Matrix Adapter, whose inner object is a foreign datatype originating from MorpheusDSL. (2) Rewrite rule begins executing in MorpheusDSL. (3) Since the rewrite rule is implemented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

the caller, FastR. Back at the caller, since we received a Normalized Matrix from MorpheusDSL, we wrap it in an Adapter and return the adapted Normalized Matrix, whose future LA operations will be factorized in the same way.

5.4 Factorizing Polyglot Scripts

As explained before, 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. In theory, Trinity's Normalized Matrix adapter could serve this role as well, because it can expose the host LA system's matrix interface from an object with our MatrixLib interface. In practice, and to facilitate debuggability, we did not re-use it for this purpose.

6. EXPERIMENTAL EVALUATION

We now present an empirical evaluation of Trinity's *efficiency* on both real-world and controllable synthetic datasets. We do this while also showcasing 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 the single-table matrix-based *Materialized* execution for various data 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?

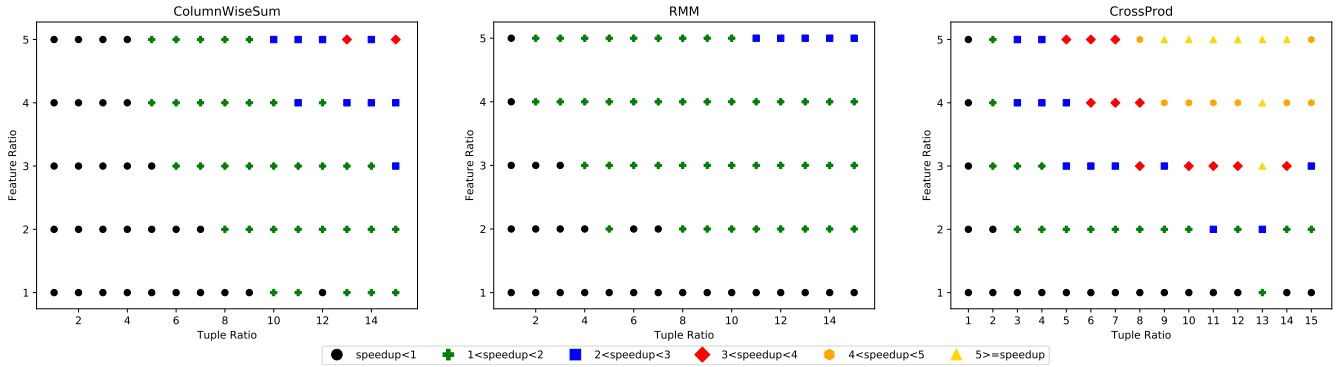


Figure 7: Discretized Speed-ups of *Trinity* over *Materialized* in FastR

As a heads-up summary, our results tell the following: (1) Trinity is significantly faster than *Materialized*; the speedups are in line with the expectations based on data redundancy ratio. (2) On many real-world datasets, Trinity's speedups are comparable to MorpheusR in many cases and only slightly lower in others ($\leq$ 2x slower). We see the latter as the price of Trinity's generality. This offers scope for future improvements. (3) Trinity seamlessly works with Javascript and a polyglot Python-R script. These newer host PLs see more modest speedups, 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.

Table 3: 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 also show results on all the 7 real-world normalized datasets from the original Morpheus paper. Table 3 lists the datasets and their statistics. For added context, we also report the actual redundancy ratio (RR) alongside. The RR here is the ratio of the memory footprint of the materialized single matrix (T) against the memory footprint of the corresponding Normalized Matrix. We use FastR's `object.size` utility to estimate the memory footprints. Note that most LA operations and LA-based ML algorithms have runtime complexity that is linear in these 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 machine with 47 Intel Xeon CPU E5-2690 v3 2.60GHz 12-cores CPUs, 512 GB RAM, and over 7 TB disk, running on Oracle Linux Server 7.3 as OS. Unless otherwise stated, we warm up for 15 runs and report the *median runtimes* of the following 10 runs. We chose median to reduce the potential noise induced by JIT compiler activity. We also report the *mean runtimes* for most of these experiments in the appendix and extended version[34] dueto space constraints; the trends/conclusions are the same though.

6.1 LA Operator-level Results in FastR

We first evaluate Trinity's efficiency at the LA operator 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* execution. We *exclude* the time to materialize the join, which might favor Materialized. Due to space constraints, we only show three LA operators here–ColumnWiseSum, RMM, and CrossProd–and present the rest in our appendix.Figure 7 shows the results. Other LA operators 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 5x in many cases.

6.2 ML Algorithm-level Results in FastR

Table 4: Mean runtimes (in seconds) for Materialized (M), median speed-ups of Trinity relative to M (S_T), median speed-ups 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 3.

		LogReg			**LinReg**		
	RR	M	S_T	S_P	M	S_T	S_P
M	12.7	74.3	11.2	23.4	73.3	18.6	40.0
Y	7.5	21.6	8.1	15.3	19.4	9.6	15.9
W	5.9	15.7	2.9	5.6	13.2	4.3	9.1
F	4.8	2.3	1.0	1.0	2.1	1.1	1.2
E	4.5	81.0	3.2	3.4	78.6	4.0	4.0
L	4.4	9.1	2.6	5.2	8.2	3.5	5.0
B	2.3	4.1	1.3	2.2	3.3	1.4	1.9

		KMeans			**GNMF**		
	RR	M	S_T	S_P	M	S_T	S_P
M	12.7	157.5	6.4	8.8	22.7	1.2	1.2
Y	7.5	47.6	6.1	7.9	5.8	1.1	1.2
W	5.9	30.6	1.8	3.8	7.0	0.5	0.9
F	4.8	4.7	0.9	0.8	2.0	0.6	0.6
E	4.5	186.0	2.9	2.6	24.8	0.6	0.4
L	4.4	22.2	2.1	2.8	8.0	1.1	1.1
B	2.3	10.3	1.0	0.9	5.0	0.7	0.6

We now compare Trinity's efficiency against Materialized on the 4 LA-based ML algorithms. We also compare Trinity's speedups with the speedups offered by MorpheusR, a PL-specific prior implementation of Morpheus, to evaluate the price of Trinity's generality. We use the 7 real-world datasets for these results. Table 4 shows the results.

We see that *LogReg* reports significant speedups with both Trinity and MorpheusR on all datasets except *Flights*. The speedups of Trinity are over 10x on *Movies* and 8x on *Yelp*. *LinReg* reports even higher speedups: Trinity is almost 19x and 9x faster than Materialized on those respective datasets. *KMeans* and *GNMF* report comparably lower speedups because a larger fraction of their LA scripts are spent on model computations, for which factorized execution is not relevant. However, *KMeans* reports over 6x speedup on *Movies* with Trintiy. That said, *KMeans* and *GNMF* do see slight slowdowns in some cases with Trinity (within 2.1x slowdown). This is because the raw number of FLOPS on those datasets are low due to their smaller sizes, causing the rewrite overheads to dominate relatively. This trend was also seen in the prior PL-specific Morpheus implementations in some cases [38, 25].

Trinity vs MorpheusR. In terms of relative comparisons of Trinity and MorpheusR, we see that Trinity is no more than roughly 2.1x slower than MorpheusR. In fact, it is roughly on par with MorpheusR in many cases and even slightly faster than MorpheusR in some cases, e.g., *KMeans* on *Expedia*. We believe this price may be reasonable for many practitioners given the other generality of Trinity. We also suspect as GraalVM matures, this gap will close. We also remark here that MorpheusR when run on regular GNU-R reported no slowdowns relative to Materialized on any of these datasets. However, the absolute runtimes were higher on GNU-R. This suggests that FastR's optimizations over GNU-R meant that the relative impact of the Morpheus rewrite rules translated to relatively lower speedups for these datasets with lower redundancy ratios.

Table 5: Runtimes (in seconds) on GraalVM for GraalJS and GraalPython+FastR.

	GraalJS				**GraalPython + FastR**			
FR	M	S_T	σ_M	σ_T	M	S_T	σ_M	σ_T
1	355.3	1.0	2.6	6.7	26.0	1.7	1.1	0.7
2	503.6	1.4	6.5	3.5	22.7	1.0	2.0	1.0
3	685.8	1.9	5.4	1.4	26.2	1.1	2.4	0.3
4	816.5	2.0	8.5	1.3	34.7	1.1	3.1	0.8
5	948.8	2.2	6.6	0.8	48.3	1.2	12.4	2.9

In the appendix, we compare Trinity against MorpheusR for some LA operators as well.The takeaways are largely similar to the discussion above in regard to the extra overheads and quirks of GraalVM. Thus, we skip discussing the LA operator-level overheads 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 *LogReg*. 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 GraalJS than in FastR. Table 5 presents the results.

We see that the speedups of *Trinity* go up with FR. 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.

6.3.2 Polyglot GraalPython + FastR Execution

To demonstrate that Trinity works with GraalVM's polyglot capabilities, we run the *LogReg* 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 , 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 the GraalJS experiment; Table 5 shows the results again.

Clearly, unlike GraalJS, the speedup behavior here is more erratic and less pronounced. In fact, the speedups seems to drop and then go up again with FR. This sort of non-monotonic behavior is anomalous and not expected from Morpheus rewrite rules. We believe this happens primarily because GraalPython implementation is still in alpha stage and unstable; it does not even offer repeatable runtimes yet!

But apart from that known issue, speedups are anyway expected to be lower here than a single host PL execution because of the extra overhead of the addition layer of indirection inherent in such polyglot execution. While using larger data sizes could have mitigated the relative effects of overheads, GraalPython is unfortunately unable to support such memory footprints yet, which often led to segmentation faults and other exceptions in our trails. Of course, Oracle is continuing to improve GraalPython, and thus we expect these issues to get mitigated in the future.

6.4 Current Limitations

We showed a working proof-of-concept for a first-of-a-kind polyglot framework for factorized ML. However, we made a few assumptions for tractability sake. We recap the major current 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 as the original Morpheus paper showed. (3) We target LA-based statistical/ML algorithms like the prior Morpheus work, not tree-based methods or deep nets. 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 MorphesuDSL to add rewrite rules for non-linear operations like feature interactions from MorpheusFI [38]. (4) Finally, both Morpheus and MorpheusFI showed that factorized execution may be slower at very low TR and/or FR and proposed a simple cost model to threshold on these to decide when to use regular materialized execution. 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. It may be possible to automate the cost models in MorpheusDSL using GraalVM's more advanced capabilities, but we leave such complex extensions to future work.

7. RELATED WORK

Factorized ML and LA. Our work extends a recent line of work in the DB community on factorized ML, which we split into three groups based on target workloads and setting: specific ML algorithms [44, 35, 46], in-RDBMS execution [36, 21, 45], and LA frameworks [25, 38, 33, 37]. Our work is *complementary* to all these prior works and builds on them. The novelty of our information system architecture (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 [25], 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 rather powerful capability can empower more such ideas from the DB+ML systems world to be adopted into GraalVM and ultimately, help benefit practical data science applications.

DSLs for ML and LA. The landscape of DSLs for ML, LA, and scientific computing in general, is large and diverse. We discuss a few closely related work from the PL and compilers literatures. OptiML and the larger Delite framework [48, 24, 49] optimize high-level ML application code to achieve high performance and parallelism on specialized hardware. Diesel [28] 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 Scientist to invoke GPU kernels with ease [8, 7]. 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 [22]. 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.

Interoperability in Multi-PL VMs. We build on top of GraalVM's many years of interoperability tooling research [41]. That being said, other language VMs offer varying degrees of language interoperability such as .NET, via its Common Language Runtime, and the Parrot VM [40, 1, 14]. JVM-based languages, such as Scala, Kotlin, Clojure, often expose Java interoperability primitives as a selling point [39]. Other JVM-based language re-implementaitons such as JRuby or Jython have access to similar benefits [11, 42].

8. CONCLUSION AND FUTURE WORK

Factorized ML techniques help reduce runtimes of ML algorithms over normalized datasets. 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 LA systems and PLs in the fast-growing 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 ML optimizations as an emebddable DSL in a polyglot virtual machine. Our framework, Trinity, is a first-of-its-kind PL-agnostic and LA system-agnostic implementation of Morpheus, a general factorized LA framework. In doing so, Trinity supports 3 axes of generality–multiple statistical/ML algorithms, PLs/LA systems, and several rewrite optimizations–all in one unified framework. Experiments with multiple normalized datasets validates Trinity's significant speedups over materialized execution and rather competitive performance relative to a single PL-specific tool, MorpheusR.

As research at the intersection of DB, PL, and ML systems grows, we believe Trinity offers a platform for more cross-algebraic query optimizations spanning linear and relational algebras to more easily make its way to data science applications across many PLs without extensive manual cross-stack development efforts. For future work, apart from relaxing the limitations in Section 6.4, we want to explore code-generation in Trinity to bypass polyglot-induced overheads. Another avenue is to optimize other data science computations beyond ML algorithms, e.g., data preparation and model debugging.

9. REFERENCES

[1] Common language runtime (clr) overview - .net framework. https://docs.microsoft.com/en-us/dotnet/standard/clr. Accessed: 2020-03-01.

[2] Embed languages with the graalvm polyglot api. https://www.graalvm.org/docs/reference-manual/embed/. Accessed: 2020-03-01.

[3] Fallback (graalvm truffle java api reference). https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/Fallback.html. Accessed: 2020-03-01.

[4] Fastr github repository. https://github.com/oracle/fastr/. Accessed: 2020-03-01.

[5] Graaljs github repository. https://github.com/graalvm/graaljs. Accessed: 2020-03-01.

[6] Graalvm python implementation github repository. https://github.com/graalvm/graalpython. Accessed: 2020-03-01.

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

[8] grcuda github repository. https://github.com/NVIDIA/grcuda. Accessed: 2020-03-01.

[9] Interactive matrix programming with sas iml software. https://www.sas.com/en_us/software/iml.html. Accessed: 2020-03-01.

[10] Interoplibrary (graalvm truffle reference). https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/interop/InteropLibrary.html. Accessed: 2020-03-01.

[11] Jython project homepage. https://www.jython.org/. Accessed: 2020-03-01.

[12] Math.js project homepage. https://mathjs.org/. Accessed: 2020-03-01.

[13] Matlab homepage. https://www.mathworks.com/products/matlab.html. Accessed: 2020-03-01.

[14] Parrotvm documentation - hlls and interoperation. http://docs.parrot.org/parrot/latest/html/docs/book/draft/chXX_hlls.pod.html. Accessed: 2020-03-01.

[15] The r project for statistical computing. https://www.R-project.org/. Accessed: 2020-03-01.

[16] 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] Specialization (graalvm truffle java api reference). https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/Specialization.html. Accessed: 2020-03-01.

[18] State of data science and machine learning 2019. https://www.kaggle.com/kaggle-survey-2019. Accessed: 2020-03-01.

[19] Trufflelibraries documentation. https://github.com/oracle/graal/blob/master/truffle/docs/TruffleLibraries.md. Accessed: 2020-03-01.

[20] Walnut project homepage on oracle labs. https://labs.oracle.com/pls/apex/f?p=LABS:project_details:0:15. Accessed: 2020-03-01.

[21] M. Abo Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. In-database learning with sparse tensors. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, SIGMOD/PODS 18, page 325340, New York, NY, USA, 2018. Association for Computing Machinery.

[22] T. A. Anderson, H. Liu, L. Kuper, E. Totoni, J. Vitek, and T. Shpeisman. Parallelizing julia with a non-invasive DSL. In P. Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 4:1–4:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[23] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, and et al. Systemml: Declarative machine learning on spark. *Proc. VLDB Endow.*, 9(13):14251436, Sept. 2016.

[24] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In C. Cascaval and P. Yew, editors, *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 35–46. ACM, 2011.

[25] L. Chen, A. Kumar, J. F. Naughton, and J. M. Patel. Towards linear algebra over normalized data. *PVLDB*, 10(11):1214–1225, 2017.

[26] L. Clark. 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.

[27] M. L. V. de Vanter, C. Seaton, M. Haupt, C. Humer, and T. Würthinger. Fast, flexible, polyglot instrumentation support for debuggers and other tools. *CoRR*, abs/1803.10201, 2018.

[28] V. Elango, N. Rubin, M. Ravishankar, H. Sandanagobalane, and V. Grover. Diesel: DSL for linear algebra and neural net computations on gpus. In J. Gottschlich and A. Cheung, editors, *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 42–51. ACM, 2018.

[29] M. Furr and J. Foster. Checking type safety of foreign function calls. *ACM Trans. Program. Lang. Syst.*, 30, 07 2008.

[30] M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, and M. Luján. Cross-language interoperability in a multi-language runtime. *ACM Trans. Program. Lang. Syst.*, 40(2):8:1–8:43, 2018.

[31] M. Grimmer, C. Seaton, R. Schatz, T. Würthinger,

and H. Mössenböck. High-performance cross-language interoperability in a multi-language runtime. In M. Serrano, editor, *Proceedings of the 11th Symposium on Dynamic Languages, DLS 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 78–90. ACM, 2015.

[32] M. Grimmer, C. Seaton, R. Schatz, T. Würthinger, and H. Mössenböck. High-performance cross-language interoperability in a multi-language runtime. *SIGPLAN Not.*, 51(2):78–90, Oct. 2015.

[33] D. Hutchison, B. Howe, and D. Suciu. 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*, BeyondMR17, New York, NY, USA, 2017. Association for Computing Machinery.

[34] D. A. Justo. *Write once, rewrite everywhere: A Unified Framework for Factorized Machine Learning*. PhD thesis, UC San Diego, 2019.

[35] A. Kumar, M. Jalal, B. Yan, J. Naughton, and J. M. Patel. Demonstration of santoku: Optimizing machine learning over normalized data. *Proc. VLDB Endow.*, 8(12):18641867, Aug. 2015.

[36] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD 15, page 19691984, New York, NY, USA, 2015. Association for Computing Machinery.

[37] A. Kunft, A. Katsifodimos, S. Schelter, S. Breundefined, T. Rabl, and V. Markl. An intermediate representation for optimizing machine learning pipelines. *Proc. VLDB Endow.*, 12(11):15531567, July 2019.

[38] S. Li, L. Chen, and A. Kumar. Enabling and optimizing non-linear feature interactions in factorized linear algebra. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1571–1588. ACM, 2019.

[39] W. H. Li, D. R. White, and J. Singer. Jvm-hosted languages: they talk the talk, but do they walk the walk? In *PPPJ '13*, 2013.

[40] T. M. Malone. Interoperability in programming languages. 2014.

[41] F. Niephaus, T. Felgentreff, and R. Hirschfeld. 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*, pages 1:1–1:3. ACM, 2019.

[42] C. O. Nutter, T. Enebo, N. Sieger, and I. Dees. *Using JRuby: Bringing Ruby to Java*. Pragmatic Bookshelf, 2011.

[43] T. E. Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[44] S. Rendle. Scaling factorization machines to relational data. *Proc. VLDB Endow.*, 6(5):337348, Mar. 2013.

[45] M. Schleich, D. Olteanu, M. Abo Khamis, H. Q. Ngo, and X. Nguyen. A layered aggregate engine for analytics workloads. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD 19, page 16421659, New York, NY, USA, 2019. Association for Computing Machinery.

[46] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD 16, page 318, New York, NY, USA, 2016. Association for Computing Machinery.

[47] D. Smilkov, N. Thorat, Y. Assogba, A. Yuan, N. Kreeger, P. Yu, K. Zhang, S. Cai, E. Nielsen, D. Soergel, S. Bileschi, M. Terry, C. Nicholson, S. N. Gupta, S. Sirajuddin, D. Sculley, R. Monga, G. Corrado, F. B. Viegas, and M. Wattenberg. Tensorflow.js: Machine learning for the web and beyond. Palo Alto, CA, USA, 2019.

[48] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embedded Comput. Syst.*, 13(4s):134:1–134:25, 2014.

[49] A. K. Sujeeth, A. Gibbons, K. J. Brown, H. Lee, T. Rompf, M. Odersky, and K. Olukotun. Forge: generating a high performance DSL implementation from a declarative specification. In J. Järvi and C. Kästner, editors, *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, pages 145–154. ACM, 2013.

[50] A. Thomas and A. Kumar. A comparative evaluation of systems for scalable linear algebra-based analytics. *Proc. VLDB Endow.*, 11(13):21682182, Sept. 2018.

[51] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, İ. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and S. . . Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 2020.

[52] C. Wimmer and T. Würthinger. Truffle: a self-optimizing runtime system. In G. T. Leavens, editor, *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, Tucson, AZ, USA, October 21-25, 2012*, pages 13–14. ACM, 2012.

[53] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In A. Cohen and M. T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 662–676. ACM, 2017.

[54] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and

M. Wolczko. One VM to rule them all. In A. L. Hosking, P. T. Eugster, and R. Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 187–204. ACM, 2013.

[55] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST interpreters. In A. Warth, editor, *Proceedings of the 8th Symposium on Dynamic Languages, DLS '12, Tucson, AZ, USA, October 22, 2012*, pages 73–82. ACM, 2012.

APPENDIX

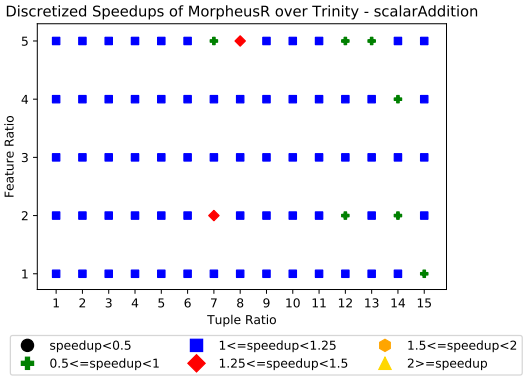
A. TRINITY VERSUS MORPHEUSR

Comparative Speed-ups. Since *Trinity* makes heavy use of indirection to remain host-agnostic, we expect it to be slower than *MorpheusR* but hope that GraalVM's JIT compiler would manage to collapse that indirection enough as to make that trade-off worthwhile. Figure 9 presents us the median speed-up of MorpheusR with respect to Trinity for *crossProd* while Figure 13 does so for *LMM*; the results for other operators may be found in the appendix. We chose these two operators because they deliniate Trinity's worst and best comparative performance respectively and so they help us establish that, compared to *MorpheusR* at this level of warm-up, Trinity is *roughly* no slower than double its runtime and no faster than half of it.

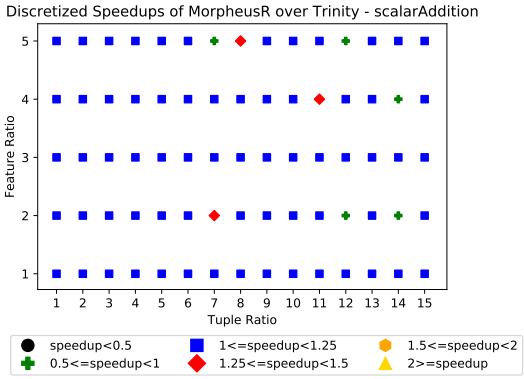
Why is Trinity faster than MorpheusR for LMM but slower in crossProd? Inspecting the comparative plots for all operators reveals that Trinity is more often *a bit slower* than MorpheusR, as we would have expected, so its performance for *LMM* may be an implementation quirk that just happens to satisfy FastR's operator cost model better than MorpheusR. After all, MorpheusR was desgined and tuned with performance tricks optimized for GNU-R, so some of them may not apply for GraalVM's R implementation.

Why does the relative performance of MorpheusR compared to Trinity vary non-monotonically as TR-FR grow? Running these experiments on a managed runtime enviroment like GraalVM means that background VM activity like garbage collection pauses and compilation may introduce some noise in our measurements. These reduce once the sytem is warmed-up *enough* but warming-up a VM is not an exact science. In our experience, it often took hundreds if not thousands of iterations before smoothing-out our measurements, which did not align with the real-world uses that we envision for our platform and therefore we opted against it. By reviewing the appendix, it becomes clear that the noise is most evidenced in lower TR-FR configurations, which makes sense because those trials took less to run and so VM-induced noise becomes more visible.

B. MORE VISUALIZATIONS

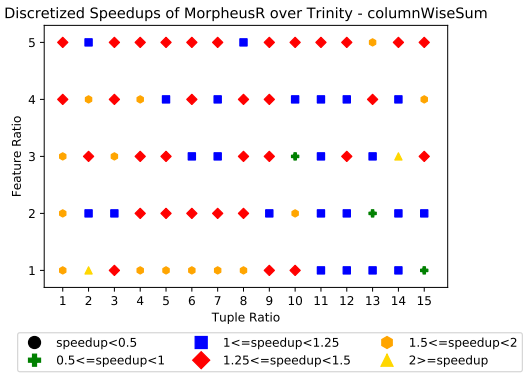


(a) *Mean*

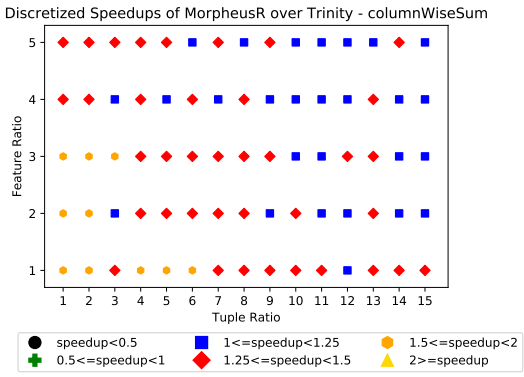


(b) *Median*

Figure 8: Discretized Speed-ups by *MorpheusR* over *Trinity* in FastR for scalarAddition

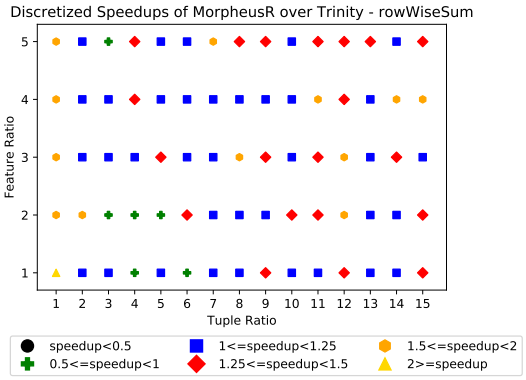


(a) *Mean*

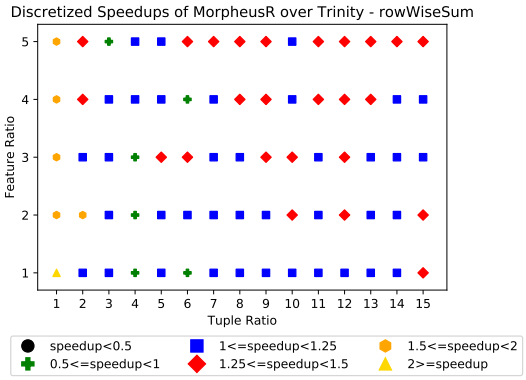


(b) *Median*

Figure 9: Discretized Speed-ups by *MorpheusR* over *Trinity* in FastR for crossProduct

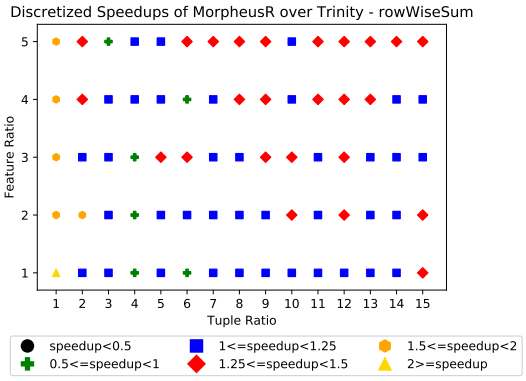


(a) *Mean*

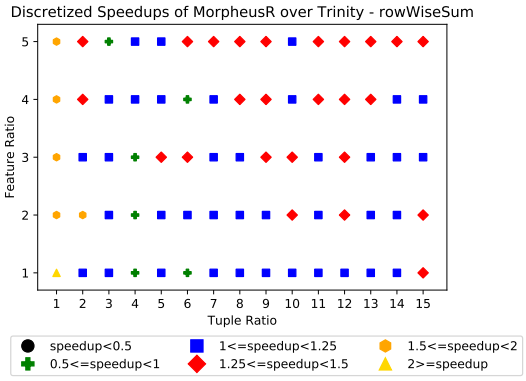


(b) *Mean*

Figure 10: Discretized Speed-ups by *MorpheusR* over *Trinity* in FastR for elementWiseSum

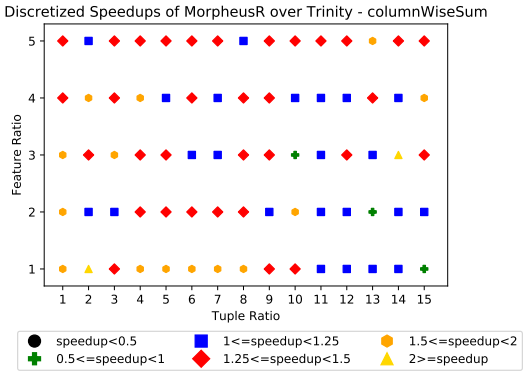


(a) *Mean*

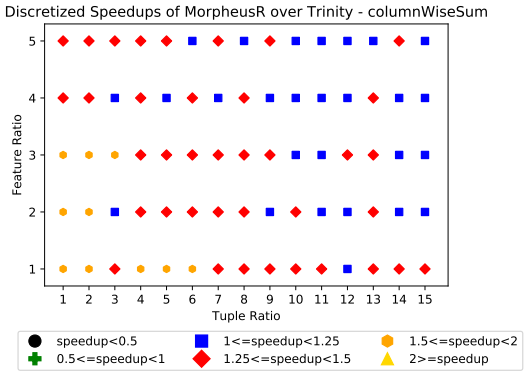


(b) *Median*

Figure 11: Discretized Speed-ups by *MorpheusR* over *Trinity* in FastR for rowWiseSum

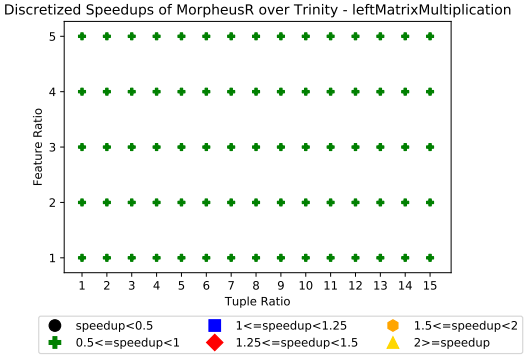


(a) *Mean*

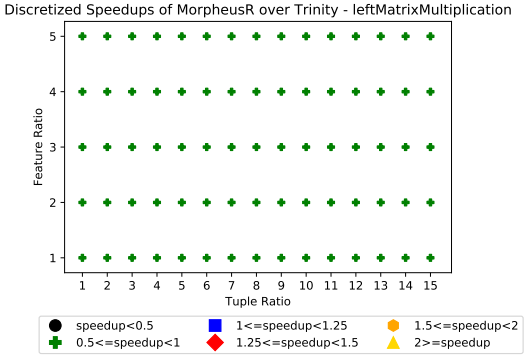


(b) *Median*

Figure 12: Discretized Speed-ups by *MorpheusR* over *Trinity* in FastR for columnWiseSum

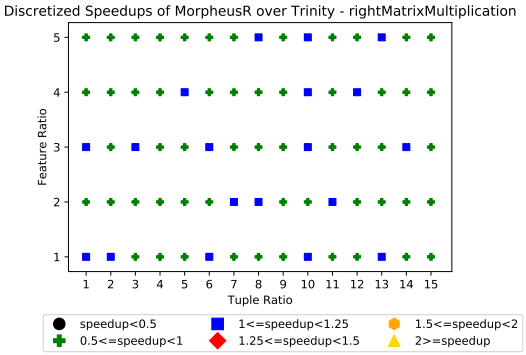


(a) *Mean*

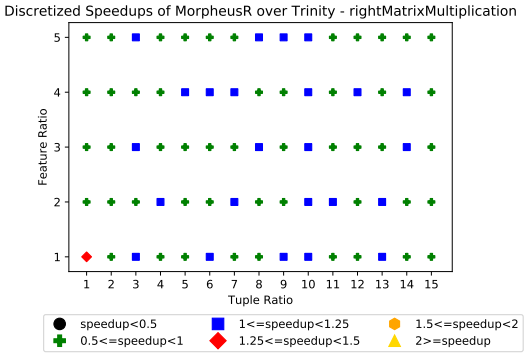


(b) *Median*

Figure 13: Discretized Speed-ups by *MorpheusR* over *Trinity* in FastR for leftMatrixMultiplication

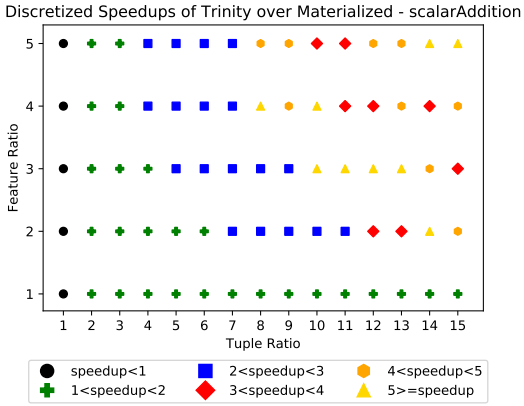


(a) *Mean*

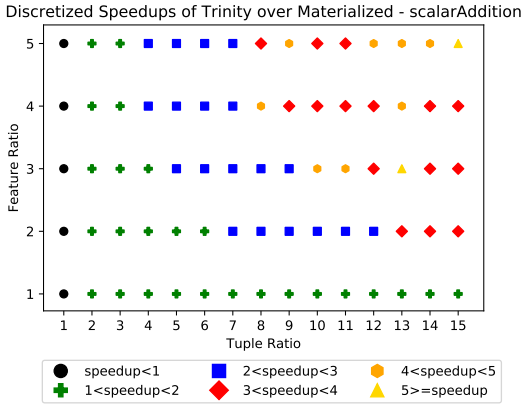


(b) *Median*

Figure 14: Discretized Speed-ups by *MorpheusR* over *Trinity* in FastR for rightMatrixMultiplication

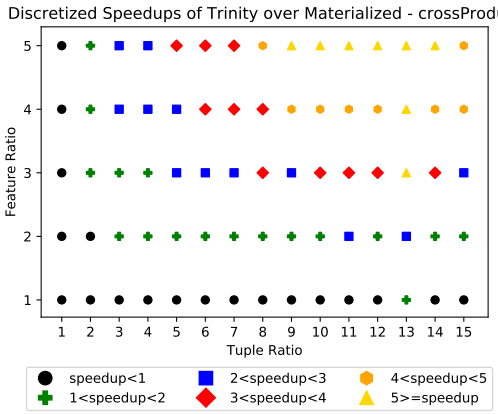


(a) *Mean*

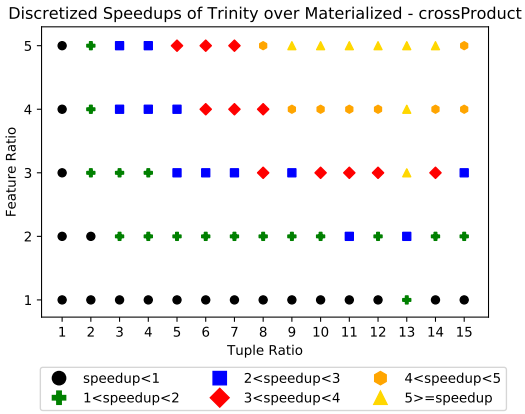


(b) *Median*

Figure 15: Discretized Speed-ups by *Trinity* over *Materialized* in FastR for scalarAddition

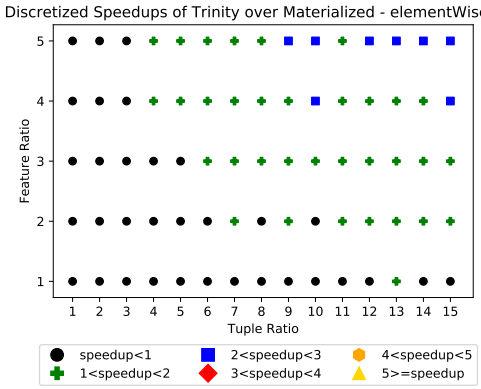


(a) *Mean*

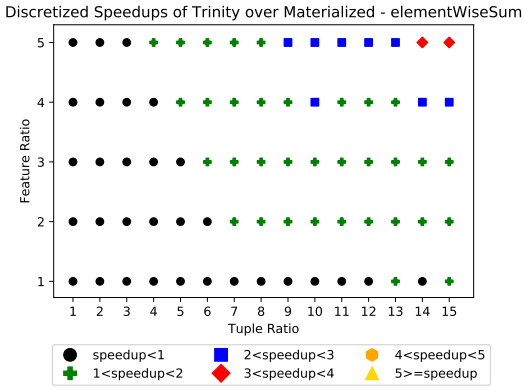


(b) *Median*

Figure 16: Discretized Speed-ups by *Trinity* over *Materialized* in FastR for crossProduct

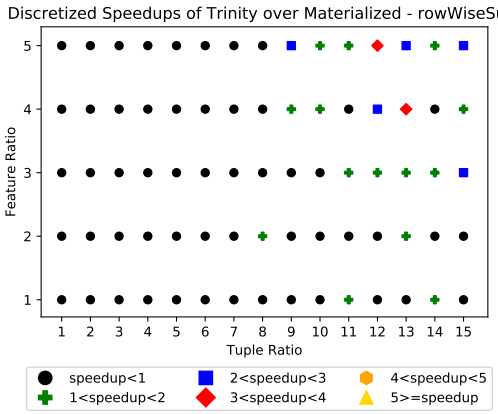


(a) *Mean*

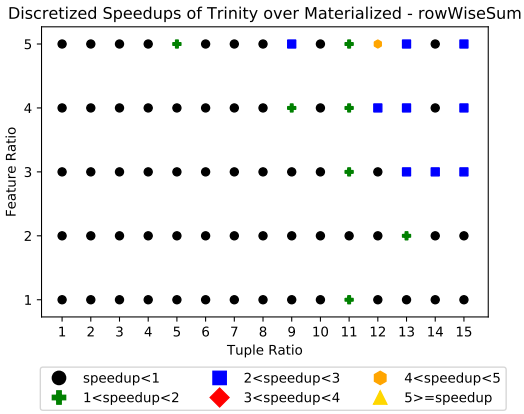


(b) *Mean*

Figure 17: Discretized Speed-ups by *Trinity* over *Materialized* in FastR for elementWiseSum

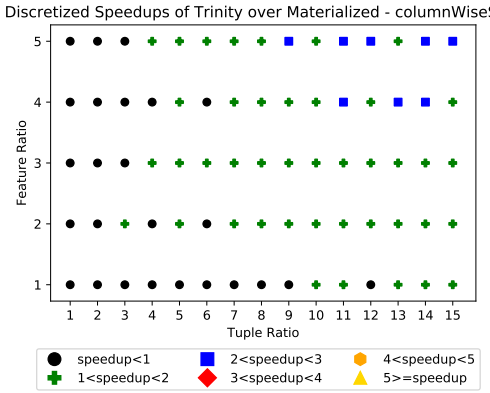


(a) *Mean*

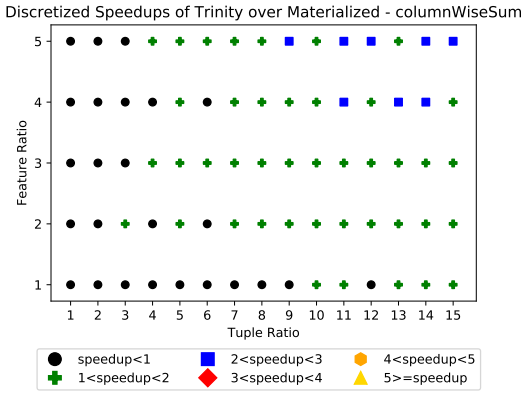


(b) *Median*

Figure 18: Discretized Speed-ups by *Trinity* over *Materialized* in FastR for rowWiseSum

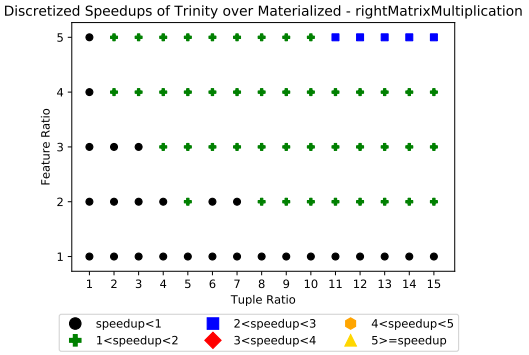


(a) *Mean*

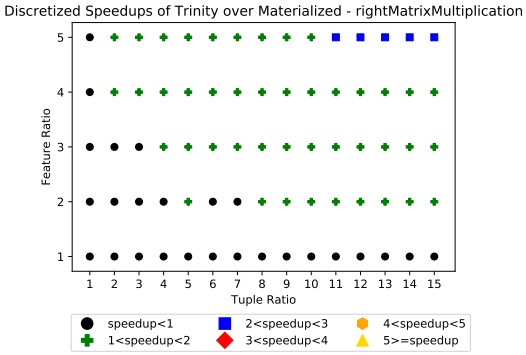


(b) *Median*

Figure 19: Discretized Speed-ups by *Trinity* over *Materialized* in FastR for columnWiseSum

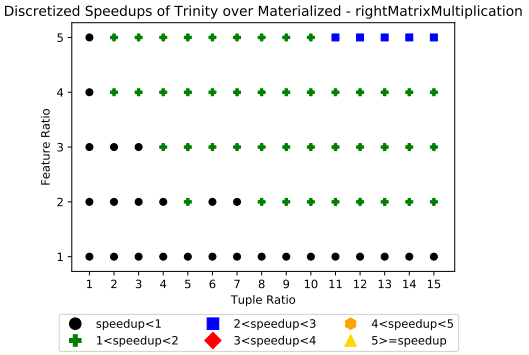


(a) *Mean*

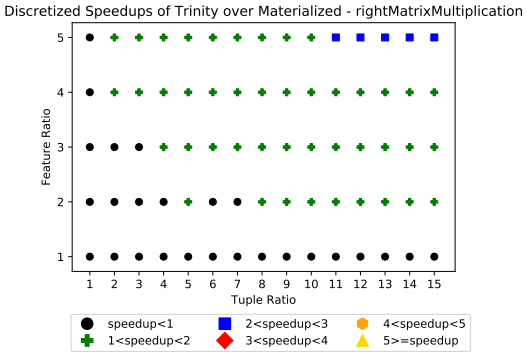


(b) *Median*

Figure 20: Discretized Speed-ups by *Trinity* over *Materialized* in FastR for leftMatrixMultiplication

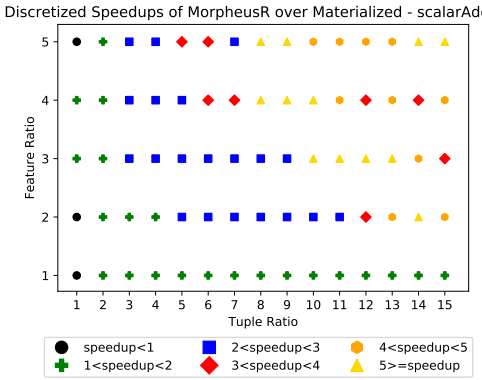


(a) *Mean*

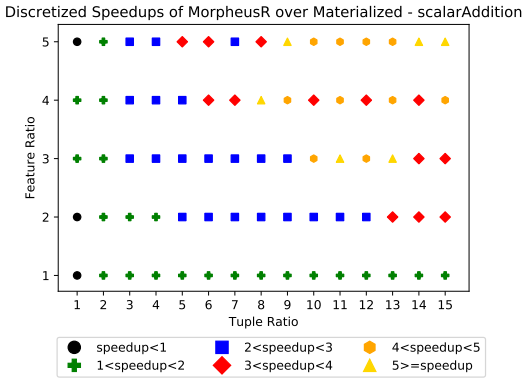


(b) *Median*

Figure 21: Discretized Speed-ups by *Trinity* over *Materialized* in FastR for rightMatrixMultiplication

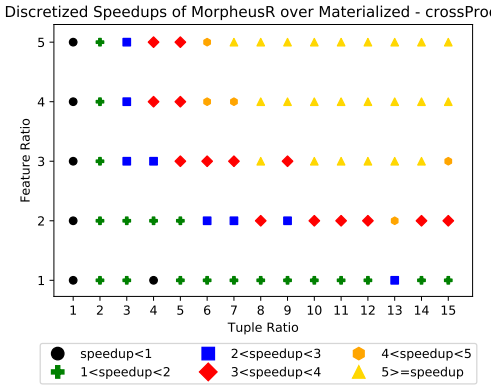


(a) *Mean*

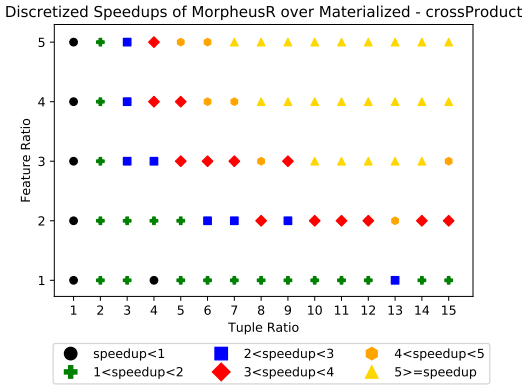


(b) *Median*

Figure 22: Discretized Speed-ups by *MorpheusR* over *Materialized* in FastR for scalarAddition

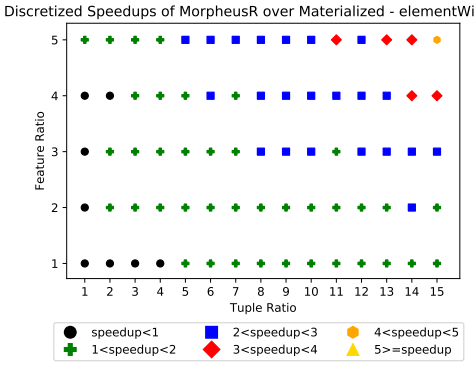


(a) *Mean*

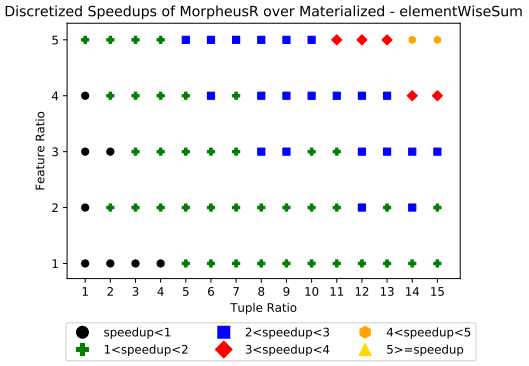


(b) *Median*

Figure 23: Discretized Speed-ups by *MorpheusR* over *Materialized* in FastR for crossProduct

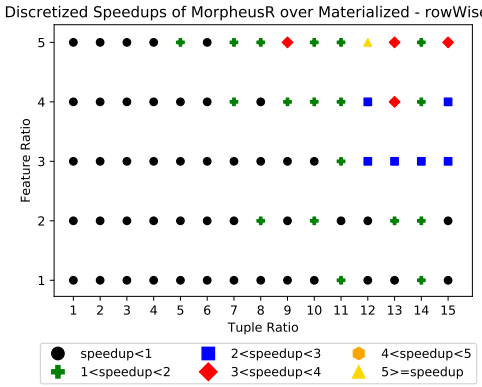


(a) *Mean*

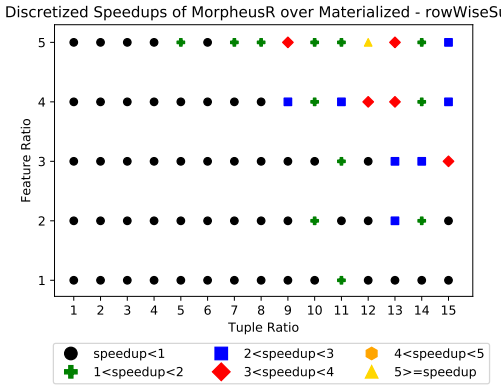


(b) *Mean*

Figure 24: Discretized Speed-ups by *MorpheusR* over *Materialized* in FastR for elementWiseSum

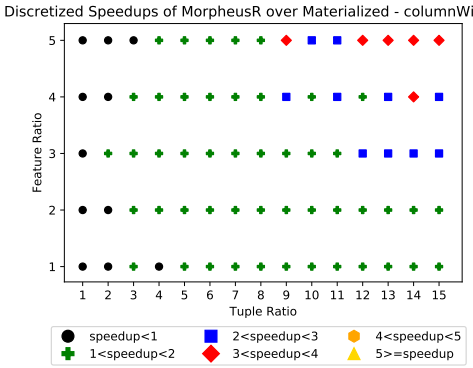


(a) *Mean*

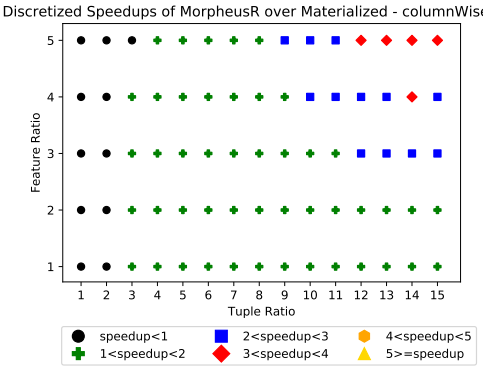


(b) *Median*

Figure 25: Discretized Speed-ups by *MorpheusR* over *Materialized* in FastR for rowWiseSum

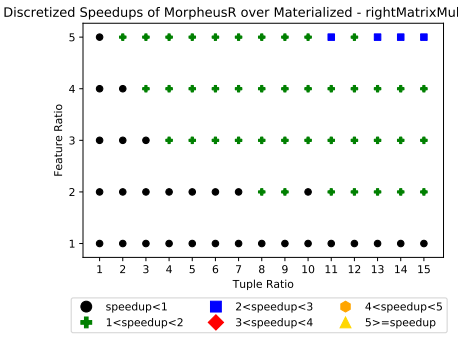


(a) *Mean*

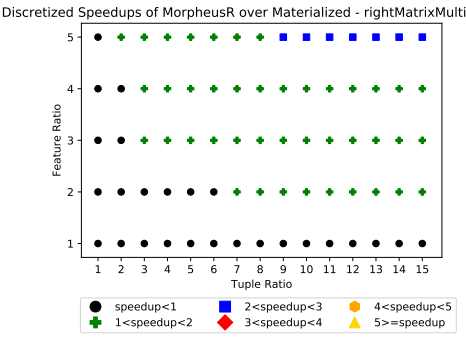


(b) *Median*

Figure 26: Discretized Speed-ups by *MorpheusR* over *Materialized* in FastR for columnWiseSum



(a) *Mean*



(b) *Median*

Figure 27: Discretized Speed-ups over *Materialized* in FastR for leftMatrixMultiplication

(a) *Mean* (b) *Median*

Figure 28: Discretized Speed-ups by *MorpheusR* over *Materialized* in FastR for rightMatrixMultiplication