HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries

Rana Alotaibi
UC San Diego
ralotaib@eng.ucsd.edu

Bogdan Cautis
University of Paris-Saclay
bogdan.cautis@u-psud.fr

Alin Deutsch
UC San Diego
deutsch@cs.ucsd.edu

Ioana Manolescu
Inria & Institut Polytechnique de Paris
ioana.manolescu@inria.fr

ABSTRACT

Hybrid complex analytics workloads typically include (*i*) data management tasks (joins, selections, etc.), easily expressed using relational algebra (RA)-based languages, and (*ii*) complex analytics tasks (regressions, matrix decompositions, etc.), mostly expressed in linear algebra (LA) expressions. Such workloads are common in many application areas, including scientific computing, web analytics, and business recommendation. Existing solutions for evaluating hybrid analytical tasks – ranging from LA-oriented systems, to relational systems (extended to handle LA operations), to hybrid systems – either optimize data management and complex tasks separately, exploit RA properties only while leaving LA-specific optimization opportunities unexploited, or focus heavily on physical optimization, leaving semantic query optimization opportunities unexplored. Additionally, they are not able to exploit *precomputed (materialized) results* to avoid recomputing (part of) a given mixed (RA and/or LA) computation.

In this paper, we take a major step towards filling this gap by proposing HADAD, an extensible lightweight approach for optimizing hybrid complex analytics queries, based on a common abstraction that facilitates unified reasoning: *a relational model endowed with integrity constraints*. Our solution can be naturally and portably applied on top of pure LA and hybrid RA-LA platforms without modifying their internals. An extensive empirical evaluation shows that HADAD yields significant performance gains on diverse workloads, ranging from LA-centered to hybrid.

CCS CONCEPTS

• **Data management systems** → *Semantic query optimization.*

KEYWORDS

linear algebra, query rewriting, integrity constraints, chase

ACM Reference Format:
Rana Alotaibi, Bogdan Cautis, Alin Deutsch, and Ioana Manolescu. 2021. HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457311

1 INTRODUCTION

Modern analytical tasks typically include (*i*) data management tasks (e.g., joins, filters) to perform pre-processing steps, including feature selection, transformation, and engineering [21, 25, 35, 43, 45], tasks that are easily expressed using RA-based languages, as well as (*ii*) complex analytics tasks (e.g., regressions, matrices decompositions), which are mostly expressed using LA operations [34]. To perform such analytical tasks, data scientists can choose from a variety of systems, tools, and languages. Languages/libraries such as R [9] and NumPy [8], as well as LA systems such as SystemML [22], TensorFlow [14] and MLlib [11] treat matrices and LA operations as first-class citizens: they offer a rich set of built-in LA operations. However, it can be difficult to express pre-processing tasks in these systems. Further, expression rewrites, based on equivalences that hold due to well-known LA properties, are not exploited in some of these systems, leading to missed optimization opportunities.

Many works propose integrating RA and LA processing, where both algebraic styles can be used together [10, 24, 28, 33, 36, 38, 47]. [10] offers calling LA packages through user defined functions (UDFs), where libraries such as NumPy are embedded in the host language. Others suggest extending RDBMS to treat LA objects as first-class citizens by using built-in functions to express LA operations [33, 38]. However, LA operations' semantics remain hidden behind these functions, which the optimizers treat as black boxes. SPORES [47] and SPOOF [24] optimize LA expressions by converting them into RA, optimizing the latter, and then converting the result back to an (optimized) LA expression. They only focus on optimizing LA pipelines containing operations that can be expressed in RA. The restriction is that LA properties of complex operations such as inverse, matrix-decompositions are entirely unexploited. Morpheus [28] speeds up LA pipelines over large joins by pushing computation into each joined table, thereby avoiding expensive materialization. LARA [36] focuses on *low-level* optimization by exploiting data layouts (e.g., column-wise) to choose LA operators' physical implementations. A limitation of such approaches is that they lack *high-level reasoning about LA properties and rewrites*, which can drastically enhance the pipelines' performance [46].

Further, the aforementioned solutions do not support *semantic query optimization*, which includes exploiting integrity constraints and materialized views and can bring enormous performance advantages in hybrid RA-LA and even plain LA settings.

We propose HADAD, an extensible lightweight framework for providing semantic query optimization (including views-based, integrity constraint-based, and LA property-based rewriting) on top of both pure LA and hybrid RA-LA platforms, with no need to modify their internals. At the core of HADAD lies a common abstraction: *relational model with integrity constraints*, which enables reasoning in hybrid settings. Moreover, it makes it very easy to extend HADAD's semantic knowledge of LA operations by simply declaring appropriate constraints, with no need to change HADAD code. As we show, constraints are sufficiently expressive to declare (and thus allow HADAD to exploit) more properties of LA operations than previous work could consider.

Last but not least, our holistic, cost-based approach enables to judiciously apply for each query *the best available optimization.* For instance, given the computation $M(NP)$ for some matrices M, N and P, we may rewrite it into $(MN)P$ if its estimated cost is smaller than that of the original expression, *or* we may turn it into MV if a materialized view V stores exactly the result of (NP).

HADAD capitalizes on a framework previously introduced in [16] for rewriting queries across many data models, using materialized views, in a polystore setting that does not include the LA model. The novelty of HADAD is to extend the benefits of rewriting and views optimizations to pure LA and hybrid RA-LA computations, which are crucial for ML workloads.

Contributions. The paper makes the following contributions:

❶ We propose an *extensible lightweight approach to optimize hybrid complex analytics queries.* Our approach can be implemented on top of existing systems without modifying their internals; it is based on a powerful intermediate abstraction that supports reasoning in hybrid settings, namely *a relational model with integrity constraints.*

❷ We formalize the problem of *rewriting computations using previously materialized views in hybrid settings.* To the best of our knowledge, ours is the first work that brings views-based rewriting under integrity constraints in the context of LA-based pipelines and hybrid analytical queries.

❸ We provide *formal guarantees* for our solution in terms of soundness and completeness.

❹ We conduct an extensive set of empirical experiments on typical LA- and hybrid-based expressions, which show the benefits of HADAD.

Outline. The rest of this paper is organized as follows: §2 highlights HADAD's optimizations that go beyond the state of the art based on *real-world* scenarios, §3 formalizes the query optimization problem in the context of a hybrid setting. §4 provides an end-to-end overview of our approach. §5 presents our novel reduction of the rewriting problem into one that can be solved by existing techniques from the relational setting. §6 describes our extension to the query rewriting engine, integrating two different cost models, to help prune out inefficient rewritings as soon as they are enumerated. We formalize our solution's guarantees in §7 and present the experiments in §8. We discuss related work and conclude in §9.

2 HADAD OPTIMIZATIONS

We highlight below examples of performance-enhancing opportunities that are exploited by HADAD and not being addressed by LA-oriented and cross RA-LA existing solutions.

LA Pipeline Optimization. Consider the Ordinary Least Squares Regression (OLS) pipeline: $(X^TX)^{-1}(X^Ty)$, where X is a square matrix of size 10K×10K and y is a vector of size 10K×1. Suppose available a materialized view $V = X^{-1}$. HADAD rewrites the pipeline to $(V(V^T(X^Ty))$, by exploiting the LA properties $(CD)^{-1} = D^{-1}C^{-1}$, $(CD)E = C(DE)$ and $(D^T)^{-1} = (D^{-1})^T$ as well as the view V. The rewriting is more efficient than the original pipeline since it avoids computing the expensive inverse operation. Moreover, it optimizes the matrix chain multiplication order to minimize the intermediate result size. This leads to a 150× speed-up on MLlib [40]. Current popular LA-oriented systems [8, 9, 14, 22, 40] are not capable of exploiting such rewrites, due to the lack of systematic exploration of standard LA properties and views.

Hybrid RA-LA Optimization. Cross RA-LA platforms such as Morpheus [28], SparkSQL [18] and others [32, 36] can greatly benefit from HADAD's cross-model optimizations, which can find rewrites that they miss.

Factorization of LA Operations over Joins. For instance, Morpheus implements a powerful optimization that factorizes an LA operation on a matrix **M** obtained by joining tables **R** and **S** and casting the join result as a matrix. Factorization pushes the LA operation on **M** to operate on **R** and **S**, cast as matrices.

Consider a specific instantiation of factorization: `colSums`(**M**N), where matrix **M** has size 20M× 120 and N has size 120×100; both matrices are dense. The `colSums` operation sums up the elements in each column, returning the vector of these sums (the operation is common in ML algorithms such as K-means clustering [39]). On this pipeline, Morpheus applies a multiplication factorization rule to push the multiplication by N down to **R** and **S**: it computes **R**N and **S**N, then concatenates the resulting matrices to obtain **M**N. The size of this intermediate result is 20M×100. Finally, `colSums` is applied to the intermediate result, reducing to a 1×100 vector.

HADAD can help Morpheus do much better, by pushing the `colSums` operator to **R** and **S** (instead of the multiplication with N), then concatenating the resulting vectors. This leads to much smaller intermediate results, since the combined size of vectors `colSums`(**R**) and `colSums`(**S**) is only 1×120.

To this end, HADAD rewrites the pipeline to `colSums`(**M**)N by exploiting the property `colSums`(AB) =`colSums`$(A)B$ and applying its cost estimator, which favors rewritings with a small intermediate result size. Evaluating this HADAD-produced rewriting, Morpheus's multiplication pushdown rule no longer applies, while the `colSums` pushdown rule is now enabled, leading to 125× speed-up.

Pushing Selection from LA Analysis to RA Preprocessing. Consider another hybrid example on a Twitter dataset [13]. The JSON dataset contains tweet ids, extended tweets, entities including hashtags, filter-level, media, URL, and tweet text, etc. We implemented it on SparkSQL (with SystemML [22]).

In the preprocessing stage, our SparkSQL query constructs a *tweet-hashtag* filter-level matrix **N** of size 2M×1000, for all tweets posted from "USA" mentioning "covid", where rows are tweets, columns are hashtags, and values are filter-levels[1].

N is then loaded into SystemML, where rows with filter-level less than 4 are selected. The result undergoes an Alternating Least Square (ALS) [42] computation. A core building block of the ALS computation is the LA pipeline $(uv^T - \mathbf{N})v$. In our example, u is a tweet feature vector (of size 2M×1) and v is a hashtag feature vector (of size 1000×1).

[1]Matrix N is represented in MatrixMarket Format (MTX) since it is sparse.

We have two materialized views available: V_1 stores the tweet id and text as a *text datasource in Solr*, and V_2 stores tweet id, hashtag id, and filter-level for all tweets posted from "USA", and is materialized on disk as CSV file. The rewriting *modifies the preprocessing* of $\mathbf{N}$ by introducing V_1 and V_2; it also *pushes the filter-level selection* from the LA pipeline into the preprocessing stage. To this end, it rewrites $(uv^T - \mathbf{N})v$ to $uv^T v - \mathbf{N}v$, which is more efficient for two reasons. First, $\mathbf{N}$ is ultra sparse (0.00018% non-zero), which renders the computation of $\mathbf{N}v$ extremely efficient. Second, SystemML evaluates the chain $uv^T v$ efficiently, computing $v^T v$ first, which results in a scalar, instead of computing uv^T, which results in a dense matrix of size 2M×1000 (HADAD's cost model realizes this). Without the rewriting help from HADAD, SystemML is unable to exploit its own efficient operations for lack of awareness of the distributivity property of vector multiplication over matrix addition, $Av + Bv = (A + B)v$. The rewriting achieves 14× speed-up.

HADAD detects and applies all the above-mentioned optimizations combined. It captures RA-, LA-, and cross-model optimizations precisely because it reduces all rewrites to a single setting in which they can synergize: relational rewrites under integrity constraints.

3 PROBLEM STATEMENT

We consider a set of *value domains* $\mathcal{D}_i$, e.g., $\mathcal{D}_1$ denotes integers, $\mathcal{D}_2$ denotes real numbers, $\mathcal{D}_3$ strings, etc, and two basic data types: *relations (sets of tuples)* and *matrices* (bi-dimensional arrays). Any attribute in a tuple or cell in a matrix is a value from some $\mathcal{D}_i$. We assume *a matrix can be implicitly converted into a relation* (the order among matrix rows is lost), and *the opposite conversion* (each tuple becomes a matrix line, in some order that is unknown, unless the relation was explicitly sorted before the conversion).

We consider a hybrid language $\mathcal{L}$, comprising a set R_{ops} of (unary or binary) *RA operators*; concretely, R_{ops} comprises the standard relational selection, projection, and join. We also consider a set L_{ops} of *LA operators*, comprising: unary (e.g., inversion and transposition) and binary (e.g., matrix product) operators. The full set L_{ops} of LA operations we support is detailed in §5.1. A *hybrid expression* in $\mathcal{L}$ is defined as follows:

- any value from a domain $\mathcal{D}_i$, any matrix, and any relation, is an expression;
- (RA operators): given some expressions E, E', $ro_1(E)$ is also an expression, where $ro_1 \in R_{ops}$ is a unary relational operator, and E's type matches ro_1's expected input type. The same holds for $ro_2(E, E')$, where $ro_2 \in R_{ops}$ is a binary relational operator (i.e., the join);
- (LA operators): given some expressions E, E' that are either numeric matrices or numbers (which can be seen as degenerate 1×1 matrices), and some real number r, the following are also expressions: $lo_1(E)$ where $lo_1 \in L_{ops}$ is a unary operator, and $lo_2(E, E')$ where $lo_2 \in L_{ops}$ is a binary operator (again, provided that E, E' match the expected input types of the operators).

Clearly, an important set of *equivalence rules* hold over our hybrid expressions, well-known respectively in the RA and the LA literature. These equivalences lead to *alternative evaluation strategies* for each expression. Further, we assume given a (possibly empty) set of *materialized views* $\mathcal{V} \in \mathcal{L}$, which have been previously computed over some inputs (matrices and/or relations), and whose results are directly available (e.g., as a file on disk). Detecting when a materialized view can be used instead of evaluating (part of) an expression is another important source of alternative evaluation strategies.

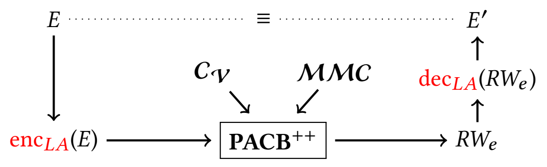

$$\begin{tikzcd}
E \arrow[rrrr, dotted, no head, "\equiv" description] \arrow[dd] & & & & E' \\
& \mathcal{C}_{\mathcal{V}} \arrow[dr] & & \mathcal{MMC} \arrow[dl] & \text{dec}_{LA}(RW_e) \arrow[u] \\
\text{enc}_{LA}(E) \arrow[rr] & & \boxed{\textbf{PACB}^{++}} \arrow[rr] & & RW_e \arrow[u]
\end{tikzcd}$$

Figure 1: Outline of Our Reduction

Given an expression E and a *cost model* that assigns a cost (a real number) to an expression, we consider the problem of **identifying the most efficient rewrite** derived from E by: (*i*) exploiting RA and LA equivalence rules, and/or (*ii*) replacing part of an expression with a scan of a materialized view equivalent to that expression.

Below, we detail our approach, the equivalence rules we capture, and two alternative cost models we devised for this hybrid setting. Importantly, our solution (based on a *relational encoding with integrity constraints*) capitalizes on the framework previously introduced in [16], where it was used to *rewrite queries using materialized views in a polystore setting*, where the data, views, and query cover a variety of data models (relational, JSON, XML, etc.). Those queries can be expressed in a combination of query languages, including SQL, JSON query languages, XQuery, etc. **The ability to rewrite such queries using heterogeneous views directly and fully transfers to HADAD**: thus, instead of a relation, we could have the (tuple-structured) results of an XML or JSON query; views materialized by joining an XML document with a JSON one and a relational database could also be reused. The novelty of our work is to **extend the benefits of rewriting and view-based optimization to LA computations, crucial for ML workloads**. In §5, we focus on capturing matrix data and LA computations in the relational framework, along with relational data naturally; this enables our novel, holistic optimization of hybrid expressions.

4 HADAD OVERVIEW

We outline here our approach as an extension to [16] for solving the rewriting problem introduced in §3.

Hybrid Expressions and Views. A hybrid expression (whether asked as a query, or describing a materialized view) can be purely relational (RA), in which case we assume it is specified as a conjunctive query [27]. Other expressions are purely LA ones; we assume that they are defined in a dedicated LA language such as R [9], DML [22], etc. , using LA operators from our set L_{ops} (see §5.1), commonly used in real-world ML workloads. Finally, a hybrid expression can combine RA and LA, e.g., an RA expression (resulting in a relation) is treated as a matrix input by an LA operator, whose output may be converted again to a table and joined further, etc.

Our approach is based on a reduction to a relational model. Below, we show how to bring our hybrid expressions - and, most specifically, their LA components - under a relational form (the RA part of each expression is already in the target formalism).

Encoding into a Relational Model. Let E be an LA expression (query) and $\mathcal{V}$ be a set of materialized views. We reduce the LA views-based rewriting problem to the relational rewriting problem under integrity constraints, as follows (see Figure 1). First, we *encode* *relationally* E, $\mathcal{V}$, and the set L_{ops} of LA operators. Note that the relations used in the encoding are *virtual* and *hidden*, i.e., invisible to both the application designers and users. They only serve to support query rewriting via relational techniques.

These virtual relations are accompanied by a set of relational *integrity constraints* $enc_{LA}(LA_{prop})$ that reflect a set LA_{prop} of LA properties of the supported L_{ops} operations. For instance, we model the *matrix addition* operation using a relation $\mathtt{add}_M(M, N, R)$, denoting that R is the result of $M+N$, together with a set of constraints stating that $\mathtt{add}_M$ is a *functional* relation that is *commutative*, *associative*, etc. These constraints are Tuple Generating Dependencies (**TGD**s) or Equality Generating Dependencies (**EGD**s) [15], which are a generalization of key and foreign key dependencies. We detail our relational encoding in §5.

Reduction from LA-based to Relational Rewriting. Our reduction translates the declaration of each view $V \in \mathcal{V}$ to constraints $enc_{LA}(V)$ that reflect the correspondence between V's input data and its output. Separately, E is also encoded as a relational query $enc_{LA}(E)$ over the relational encodings of L_{ops} and its matrices.

The reformulation problem is reduced to a purely relational setting as follows. We are given a (purely LA or hybrid) query expression $E \in \mathcal{L}$ and a set $\mathcal{V} \subseteq \mathcal{L}$ of hybrid views. We encode E as the relational query $enc_{LA}(E)$, the views as the relational integrity constraints $\mathcal{C}_{\mathcal{V}} = enc_{LA}(V_1) \cup \ldots \cup enc_{LA}(V_n)$. We add as further input the set of relational constraints $enc_{LA}(LA_{prop})$ mentioned above, which specify properties of the LA operators. We call them Matrix-Model encoding Constraints, or $\mathcal{MMC}$ in short. We must find a rewriting RW_e expressed over the relational views $\mathcal{C}_{\mathcal{V}}$, such that RW_e is equivalent to $enc_{LA}(E)$ under the constraints ($\mathcal{C}_{\mathcal{V}} \bigcup \mathcal{MMC}$) and is optimal according to our cost model. Note that RW_e is a *relationally encoded rewriting* of E; a final *decoding* step is needed to obtain $E' \in \mathcal{L}$, the (LA or hybrid) rewriting of E.

The challenge in coming up with the reduction consists in designing an encoding, i.e., one in which rewritings found by (*i*) encoding relationally, (*ii*) solving the resulting relational rewriting problem, and (*iii*) decoding a resulting rewriting over the views, is guaranteed to produce an equivalent expression E' (see §5).

Relational Rewriting Using Constraints. To solve the relational rewriting problem under constraints, the engine of choice is Provenance Aware Chase&Backchase (PACB) [30]. The PACB engine ($PACB^{++}$ hereafter) has been extended to utilize the $Pruned_{prov}$ algorithm discussed in [29, 30], which prunes inefficient rewritings during the search phase, based on a simple cost model (see §6).

Decoding of the Relational Rewriting. For the selected relational reformulation RW_e by $PACB^{++}$, a *decoding step* $dec(RW_e)$ is performed to translate RW_e into the native syntax of its respective underlying store/engine (e.g., R, DML, etc.).

5 REDUCTION TO THE RELATIONAL MODEL

Our internal model is relational, and it makes prominent use of expressive integrity constraints. This framework suffices to describe the features and properties of most data models used today, notably including relational, XML, JSON, graph, etc [16, 17].

Going beyond, in this section, we present a novel way to *reason relationally about LA operations* by treating them as un-interpreted functions with black-box semantics, and adding *constraints that capture their important properties*. First, we give an overview of a wide range of LA operations that we consider (§5.1). Then, in §5.2, we show how matrices and their operations can be *encoded* using a set of *virtual* relations, part of a schema we call $\mathcal{VREM}$ (for *Virtual Relational Encoding of Matrices*), together with the integrity constraints $\mathcal{MMC}$ that capture the properties of these operations. §5.3 exemplifies relational rewritings obtained via our reduction.

5.1 Matrix Algebra

We consider a wide range of matrix operations [19, 37], which are common in real-world ML algorithms [4]: element-wise multiplication ($\mathtt{multi}_E$), matrix-scalar multiplication ($\mathtt{multi}_{MS}$), matrix multiplication ($\mathtt{multi}_M$), addition ($\mathtt{add}_M$), division ($\mathtt{div}_M$), transposition (`tr`), inversion ($\mathtt{inv}_M$), determinant (`det`), trace (`trace`), diagonal (`diag`), exponential (`exp`), adjoints (`adj`), direct sum ($\mathtt{sum}_D$), direct product ($\mathtt{product}_D$), summation (`sum`), rows/columns summation (`rowSums`, `colSums`, respectively), QR (`QR`), Cholesky (`CHO`), LU (`LU`), and pivoted LU (`LUP`) decompositions.

5.2 VREM Schema and Relational Encoding

To model LA operations on the $\mathcal{VREM}$ relational schema (part of which appears in Table 1), we also rely on a set of integrity constraints $\mathcal{MMC}$, which are encoded using relations in $\mathcal{VREM}$. We detail the encoding below.

5.2.1 *Base Matrices and Dimensionality Modeling.*

We denote by $\mathrm{M}_{k\times z}(\mathcal{D})$ a matrix of k rows and z columns, whose values come from a domain $\mathcal{D}$, e.g., the domain of real numbers $\mathbb{R}$. For brevity we just use $\mathrm{M}_{k\times z}$. We define a virtual relation $name(M, n) \in \mathcal{VREM}$ attaching a unique ID M to any matrix identified by a name denoted n (which may be e.g., of the form "/M.csv"). This relation (shown at the top left in Table 1) is accompanied by an EGD key constraint $\mathcal{I}_{name} \in \mathcal{MMC}_m$, where $\mathcal{MMC}_m \subset \mathcal{MMC}$, stating that *two matrices with the same name n have the same ID*:

$$\mathcal{I}_{name}\colon \forall M \forall N\ name(M, n) \land name(N, n) \rightarrow M = N$$

Note that the matrix ID in *name* (and all the other virtual relations used in our encoding) are not IDs of individual matrix objects: rather, each identifies an *equivalence class* (induced by value equality) of expressions. That is, two expressions are assigned the same ID iff they yield value-based-equal matrices. In Table 1, we use M and N to denote input matrices' IDs, R for the resulting matrix ID, and s for scalar input and output.

The dimensions of a matrix are captured by a $size(M, k, z)$ relation, where k and z are the number of rows, resp. columns and M is an ID. An EGD constraint $\mathcal{I}_{size} \in \mathcal{MMC}_m$ holds on the *size* relation, stating that the ID determines the dimensions:

$$\mathcal{I}_{size}\colon \forall M \forall k_1 \forall z_1 \forall k_2 \forall z_2$$
$$size(M, k_1, z_1) \land size(M, k_2, z_2) \rightarrow k_1 = k_2 \land z_1 = z_2$$

The identity and zero matrices are captured by $Zero(O)$ and $Identity(I)$ relations, where O and I denote their IDs, respectively. They are accompanied by EGD constraints $\mathcal{I}_{iden}, \mathcal{I}_{zero} \in \mathcal{MMC}_m$, stating that zero matrices with the same sizes have the same IDs, and this also applies for identity matrices with the same size:

$$\mathcal{I}_{zero}\colon \forall O_1 \forall O_2 \forall k \forall z$$
$$Zero(O_1) \land size(O_1, k, z) \land Zero(O_2) \land size(O_2, k, z) \rightarrow O_1 = O_2$$
$$\mathcal{I}_{iden}\colon \forall I_1 \forall I_2$$
$$Identity(I_1) \land size(I_1, k, k) \land Identity(I_2) \land size(I_2, k, k) \rightarrow I_1 = I_2$$

Operation	Encoding	Operation	Encoding	Operation	Encoding
Matrix scan	$name(M, n)$	Inversion	$\texttt{inv}_M(M, R)$	Cells sum	$\texttt{sum}(M, s)$
Multiplication	$\texttt{multi}_M(M, N, R)$	Scalar Multiplication	$\texttt{multi}_{MS}(s, M, R)$	Row sum	$\texttt{rowSums}(M, R)$
Addition	$\texttt{add}_M(M, N, R)$	Determinant	$\texttt{det}(M, R)$	Colsums	$\texttt{colSums}(M, R)$
Division	$\texttt{div}_M(M, N, R)$	Trace	$\texttt{trace}(M, s)$	Direct sum	$\texttt{sum}_D(M, N, R)$
Hadamard product	$\texttt{multi}_E(M, N, R)$	Exponential	$\texttt{exp}(M, R)$	Direct product	$\texttt{product}_D(M, N, R)$
Transposition	$\texttt{tr}(M, R)$	Adjoints	$\texttt{adj}(M, R)$	Diagonal	$\texttt{diag}(M, R)$

Table 1: Snippet of the $\mathcal{VREM}$ Schema

5.2.2 *Encoding Matrix Algebra Expressions*.

LA operations are encoded into dedicated relations, as shown in Table 1. We now illustrate the encoding of an LA expression on the $\mathcal{VREM}$ schema.

Example 5.1. *Consider the LA expression E: $((MN)^T)$, where the two matrices $M_{100\times1}$ and $N_{1\times10}$ are stored as "M.csv" and "N.csv", respectively. The encoding function $enc_{LA}(E)$ takes as argument the LA expression E and returns a conjunctive query whose: (i) body is the relational encoding of E using $\mathcal{VREM}$ (see below), and (ii) head has one distinguished variable, denoting the equivalence class of the result. For instance:*

```
$enc(((MN)^T) =$
    Let $enc(MN) =$
        Let $enc(M) = \boldsymbol{Q}_0(M)$:- $name(M, \text{"M.csv"})$;
            $enc(N) = \boldsymbol{Q}_1(N)$:- $name(N, \text{"N.csv"})$;
            $R_1 = freshId()$
        in
            $\boldsymbol{Q}_2(R_1)$:- $\texttt{multi}_M(M, N, R_1) \wedge \boldsymbol{Q}_0(M) \wedge \boldsymbol{Q}_1(N)$;
        $R_2 = freshId()$
    in
      $\boldsymbol{Q}(R_2)$:- $\texttt{tr}(R_1, R_2) \wedge \boldsymbol{Q}_2(R_1)$;
```

In the above, nesting is dictated by the syntax of E. From the inner (most indented) to the outer, we first encode M and N as small queries using the name relation, then their product (to whom we assign the newly created identifier R_1), using the $multi_M$ relation and encoding the relationship between this product and its inputs in the definition of $\boldsymbol{Q}_2(R_1)$. Next, we create a fresh ID R_2 used to encode the full E (the transposed of $\boldsymbol{Q}_2$) via relation tr, *in $\boldsymbol{Q}(R_2)$. For brevity, we omit the matrices' size relations in this example and hereafter. Unfolding $\boldsymbol{Q}_2(R_1)$ in the body of $\boldsymbol{Q}$ yields:*

$$\boldsymbol{Q}(R_2) \text{ :- } \texttt{tr}(R_1, R_2) \wedge \texttt{multi}_M(M, N, R_1) \wedge \boldsymbol{Q}_0(M) \wedge \boldsymbol{Q}_1(N);$$

Now, by unfolding $\boldsymbol{Q}_0$ and $\boldsymbol{Q}_1$ in $\boldsymbol{Q}$, we obtain the final encoding of $((MN)^T)$ as a conjunctive query $\boldsymbol{Q}$:

$$\boldsymbol{Q}(R_2) \text{ :- } \texttt{tr}(R_1, R_2) \wedge \texttt{multi}_M(M, N, R_1) \wedge name(M, \text{"M.csv"}) \wedge name(N, \text{"N.csv"});$$

5.2.3 *Encoding LA Properties as Integrity Constraints*.

Figure 2 shows some of the constraints $\mathcal{MMC}_{LA_{prop}} \subset \mathcal{MMC}$, which capture textbook LA properties [19, 37] of our LA operations (§5.1). The TGDs (1), (2) and (3) state that matrix addition is commutative, matrix transposition is distributive with respect to addition, and the transposition of the inverse of matrix M is equivalent to the inverse of the transposition of M, respectively. We also express that the *virtual relations are functional* by using EGD key constraints.

$$\forall M \forall N \forall R\ \texttt{add}_M(M, N, R) \to \texttt{add}_M(N, M, R) \quad (1)$$

$$\forall M \forall N \forall R_1 \forall R_2\ \texttt{add}_M(M, N, R_1) \wedge \texttt{tr}(R_1, R_2) \to$$
$$\exists R_3 \exists R_4\ \texttt{tr}(M, R_3) \wedge \texttt{tr}(N, R_4) \wedge \texttt{add}_M(R_3, R_4, R_2) \quad (2)$$

$$\forall M \forall R_1 \forall R_2\ \texttt{inv}_M(M, R_1) \wedge \texttt{tr}(R_1, R_2) \to$$
$$\exists R_3\ \texttt{tr}(M, R_3) \wedge \texttt{inv}_M(R_3, R_2) \quad (3)$$

Figure 2: Snippet of $\mathcal{MMC}_{LA_{prop}}$ Constraints

$$\forall M \forall N \forall R_1 \forall R_2 \forall R_3 \forall R_4$$
$$name(M, \text{"M.csv"}) \wedge name(N, \text{"N.csv"}) \wedge \texttt{tr}(N, R_1) \wedge$$
$$\texttt{tr}(M, R_2) \wedge \texttt{inv}_M(R_2, R_3) \wedge$$
$$\texttt{add}_M(R_1, R_3, R_4) \to name(R_4, \text{"V.csv"})$$

Figure 3: Relational Encoding of View V

For example, the following $\mathcal{I}_{multi_M} \in \mathcal{MMC}_{LA_{prop}}$ constraint states that $multi_M$ is functional, that is the products of pairwise equal matrices are equal.

$$\mathcal{I}_{multi_M} : \forall M \forall N \forall R_1 \forall R_2$$
$$\texttt{multi}_M(M, N, R_1) \wedge \texttt{multi}_M(M, N, R_2) \to R_1 = R_2$$

Other properties [19, 37] of the LA operations we consider are similarly encoded; due to space constraints, we delegate them to [12].

5.2.4 *Encoding LA Views as Constraints.*

We translate each view definition V (defined in LA language) into relational constraints $enc_{LA}(V) \in \mathcal{C}_{\mathcal{V}}$, where $\mathcal{C}_{\mathcal{V}}$ is the set of relational constraints used to capture the views $\mathcal{V}$. These constraints show how the view's inputs are related to its output over the $\mathcal{VREM}$ schema. Figure 3 illustrates the encoding as a TGD constraint of the view $V : (N)^T + (M^T)^{-1}$ stored in a file "V.csv" and computed using matrices N and M (e.g., stored as "N.csv" and "M.csv", respectively).

5.2.5 *Encoding Matrix Decompositions.*

Matrix decompositions play a crucial role in many LA computations. We model well-known decompositions (CD, LU, QR, and Pivoted LU or PLU) as a set of virtual relations $\mathcal{VREM}_{dec}$, which we add to $\mathcal{VREM}$. The functional aspect of these relations is captured by EGDs, conceptually similar to the constraint $\mathcal{I}_{multi_M}$ (§5.2.3). Due to space constraints, we discuss the encoding of various matrix decompositions in detail in the technical report [12].

5.2.6 *Encoding LA-Oriented System Rewrite Rules.*

Most LA-oriented systems [8, 9] execute an incoming LA expression *as-is*, that is: run operations in a sequence, whose order is dictated by the expression syntax. Such systems do not exploit basic LA properties, e.g., reordering a chain of multiplied matrices in order to reduce the intermediate size. SystemML [22] is the only system that models *some* LA properties as static rewrite rules. It also comprises a set of *rewrite rules* which modify the given expressions to avoid large intermediates for aggregation and statistical operations such as $\texttt{rowSums}(M)$, $\texttt{sum}(M)$, etc. For example, SystemML uses rule:

$$\texttt{sum}(MN) = \texttt{sum}(\texttt{colSums}(M)^T \odot \texttt{rowSums}(N)) \quad (i)$$

to rewrite $\texttt{sum}(MN)$ (summing all cells in the matrix product) where $\odot$ is a matrix element-wise multiplication, to avoid actually computing MN and materializing it. However, the performance benefits of rewriting depend on the rewriting power (or, in other words, on *how much the system understands the semantics of the incoming expression*), as the following example shows.

Example 5.2. *Consider an LA expression* E=$((M^T)^k(M + N)^T)$, *where* M *and* N *are square matrixes, and expression* E'=$\texttt{sum}(E)$, *which computes the sum of all cells in* E. E' *can be rewritten to*

$$RW_1 : \texttt{sum}(\texttt{colSums}(M + N)^T \odot \texttt{rowSums}(M^k))$$

Failure to exploit the LA properties $M^T N^T = (NM)^T$ *and* $(M^n)^T = (M^T)^n$ *prevents from finding rewriting* RW_1.
E' *admits the alternative rewriting*

$$RW_2\text{:}\ \texttt{sum}((\texttt{colSums}((M^T)^k))^T \odot (\texttt{colSums}(M + N)^T)$$

which can be obtained by directly applying the rewrite rule (i) *given previously and the LA property* $\texttt{rowSums}(M^T)$=$\texttt{colSums}(M)^T$. *However,* RW_2 *creates more intermediate results than* RW_1.

To fully exploit the potential of rewrite rules (for statistical or aggregation operations), they should be accompanied by sufficient knowledge of, and reasoning on, known properties of LA operations.

To bring such fruitful optimization to other LA-oriented systems lacking support of such rewrite rules, we have incorporated SystemML's rewrite rules into our framework, encoding them as a set of integrity constraints over the virtual relations in the schema $\mathcal{VREM}$, denoted $\mathcal{MMC}_{StatAgg} \subset \mathcal{MMC}$. Thus, these rewrite rules can be exploited together with other LA properties. For instance, the rewrite rule (i) is modeled by the following constraint:

$$\begin{aligned}&\mathcal{I}_{sum} : \forall M \forall N \forall R\ \texttt{multi}_M(M, N, R) \wedge \texttt{sum}(R, s) \rightarrow\\&\exists R_1 \exists R_2 \exists R_3 \exists R_4 \texttt{colSums}(M, R_1) \wedge \texttt{tr}(R_1, R_2)\\&\wedge \texttt{rowSums}(N, R_3) \wedge \texttt{multi}_E(R_2, R_3, R_4) \wedge \texttt{sum}(R_4, s)\end{aligned}$$

We refer the reader to the technical report [12] for a full list of SystemML's encoded rewrite rules.

5.3 Relational Rewriting Using Constraints

With the set of views constraints $C_{\mathcal{V}}$ and $\mathcal{MMC} = \mathcal{MMC}_m \cup \mathcal{MMC}_{LA_{prop}} \cup \mathcal{MMC}_{StatAgg}$, we rely on $PACB^{++}$ to rewrite a given expression under integrity constraints. We exemplify this below, and detail $PACB^{++}$'s inner workings in §6.

The view V shown in Figure 3 can be used to *fully* rewrite (return the answer for) the pipeline $\boldsymbol{Q} : (M^{-1} + N)^T$ by exploiting the TGDs (1), (2) and (3) listed in Figure 2, which describe the following three LA properties, denoted LA_{prop_1}: $M + N = M + N$; $((M + N))^T = (M)^T + (N)^T$ and $((M)^{-1})^T = ((M)^T)^{-1}$. The relational rewriting RW_0 of $\boldsymbol{Q}$ using the view V is $RW_0(R_4)$:- $name(R_4, \text{“}V.csv\text{”})$. In this example, RW_0 is the only *views-based* rewriting of $\boldsymbol{Q}$. However, *five* other rewritings exist (shown in Figure 4), which reorder its operations just by exploiting the set LA_{prop_1} of LA properties.

$$RW_1 : (M^{-1})^T + N^T \qquad RW_2 : (M^T)^{-1} + N^T$$
$$RW_3 : N^T + (M^{-1})^T \qquad RW_4 : N^T + (M^T)^{-1}$$
$$RW_5 : (N + M^{-1})^T$$

Figure 4: Equivalent Rewritings of $\boldsymbol{Q}$

Rewritings have different evaluation costs. We discuss next how we estimate which among these alternatives (including evaluating $\boldsymbol{Q}$ directly) is likely the most efficient.

6 CHOICE OF AN EFFICIENT REWRITING

We introduce our cost model in §6.1, which can take two different sparsity estimators (§6.2). Then, we detail our extension to the PACB rewriting engine based on the $Prune_{prov}$ algorithm (§6.3) to prune out inefficient rewritings.

6.1 Cost Model

We estimate the cost of an expression E, denoted $\gamma(E)$, as the sum of the intermediate result sizes if one evaluates E “as stated”, in the syntactic order dictated by the expression. Real-world matrices may be *dense* (most or all elements are non-zero) or *sparse* (a majority of zero elements). The latter admits more economical representations that do not store zero elements, which our intermediate result size measure excludes. To estimate the number of non-zeros (*nnz*, in short), we incorporated two different sparsity estimators from the literature (discussed in §6.2) into our framework.

Example 6.1. *Consider* $E_1 = (MN)M$ *and* $E_2 = M(NM)$, *where we assume the matrices* $M_{50K \times 100}$ *and* $N_{100 \times 50K}$ *are dense. The total cost of* E_1 *is* $\gamma(E_1) = 50K \times 50K$ *and* $\gamma(E_2) = 100 \times 100$.

6.2 LA-based Sparsity Estimators

We outline below two existing *sparsity estimators* [22, 47] that we have incorporated into our framework to estimate *nnz*[2].

6.2.1 ***Naïve Metadata Estimator.*** The naïve metadata estimator [23, 47] derives the sparsity of the output of LA expression solely from the base matrices' sparsity. This incurs no runtime overhead since metadata about the base matrices, including the *nnz*, columns and rows are available before runtime in a specific metadata file.

6.2.2 ***Matrix Non-zero Count (MNC) Estimator.*** The MNC estimator [44] exploits matrix structural properties such as single non-zero per row, or columns with varying sparsity, for efficient, accurate, and general sparsity estimation; it relies on count-based histograms that exploit these properties. We have also adopted this framework into our approach, and compute histograms about the *base* matrices offline. However, the MNC framework still needs to derive and construct histograms for *intermediate results* online (during rewriting cost estimation). We study this overhead in §8.

6.3 Pruning Rewritings: $PACB^{++}$

We extended the PACB rewriting engine with the $Prune_{prov}$ algorithm discussed in [29, 30], to eliminate inefficient rewritings during the rewriting search phase. The naïve PACB algorithm generates all minimal (by join count) rewritings before choosing a *minimum-cost* one. While this suffices on the scenarios considered in [16, 30], the settings we obtain from our LA encoding stress-test the naïve algorithm, as commutativity, associativity, etc. blow

[2]Solving the problem of sparsity estimation is beyond the scope of this paper.

up the space of alternate rewritings exponentially. Scalability considerations forced us to further optimize naïve PACB to find only *minimum-cost rewritings*, aggressively pruning the others during the generation phase. We detail below just enough of the *PACB*'s inner working to explain $Prune_{prov}$.

6.3.1 ***PACB Background***.

At the core of PACB is the idea to *model views as constraints*, in this way reducing the view-based rewriting problem to constraints-only rewriting. Specifically, for a given view V defined by a query, the constraint V_{IO} states that *for every match of the view body against the input data, there is a corresponding (head) tuple in the view output*, while the constraint V_{OI} states the converse inclusion, i.e., *each view output tuple is due to a view body match*. From a set $\mathcal{V}$ of view definitions, PACB therefore derives a set of view constraints $C_{\mathcal{V}} = \{V_{IO}, V_{OI} \mid V \in \mathcal{V}\}$.

Given a source schema σ with a set of integrity constraints $\mathcal{I}$, a set $\mathcal{V}$ of views defined over σ, and a conjunctive query Q over σ, the **rewriting problem** thus becomes: find every reformulation query ρ over the schema of view names $\mathcal{V}$ that is equivalent to Q under the constraints $\mathcal{I} \cup C_{\mathcal{V}}$.

Example 6.2. *For instance, if $\sigma = \{R, S\}$, $\mathcal{I} = \emptyset$, $\tau = \{V\}$ and we have a view V materializing the join of relations R and S, $V(x,y)$:- $R(x,z) \wedge S(z,y)$, the pair of constraints capturing V is the following:*

$$V_{IO}: \quad \forall x \forall z \forall y\ R(x,z) \wedge S(z,y) \rightarrow V(x,y)$$
$$V_{OI}: \quad \forall x \forall y V(x,y) \rightarrow \exists z\ R(x,z) \wedge S(z,y).$$

Given the query $Q(x,y)$:- $R(x,z) \wedge S(z,y)$, PACB finds the rewriting $RW(x,y)$:- $V(x,y)$. Algorithmically, this is achieved by:

(i) chasing Q with the constraints $\mathcal{I} \cup C_{\mathcal{V}}^{IO}$, where $C_{\mathcal{V}}^{IO} = \{V_{IO} \mid V \in \mathcal{V}\}$; intuitively, this enriches (extends) Q with all the consequences that follow from its atoms and the constraints $\mathcal{I} \cup C_{\mathcal{V}}^{IO}$.

(ii) restricting the chase result to only the $\mathcal{V}$-atoms; the result is called the universal plan *U.*

(iii) annotating each atom of the universal plan U with a unique ID called a provenance term.

(iv) chasing U with the constraints in $\mathcal{I} \cup C_{\mathcal{V}}^{OI}$, where $C_{\mathcal{V}}^{OI} = \{V_{OI} \mid V \in \mathcal{V}\}$, and annotating each relational atom a introduced by these chase steps with a provenance formula[3] *$\pi(a)$, which gives the set of U-subqueries whose chasing led to the creation of a; the result of this phase, called the* backchase*, is denoted B.*

(v) matching Q against B and outputting as rewritings the subsets of U that are responsible for the introduction (during the backchase) of the atoms in the image $h(Q)$ of Q; these rewritings are read off directly from the provenance formula $\pi(h(Q))$.

In our example, $\mathcal{I}$ is empty, $C_{\mathcal{V}}^{IO} = \{V_{IO}\}$, and the result of the chase in phase (i) is $Q_1(x,y)$:- $R(x,z) \wedge S(z,y) \wedge V(x,y)$. The universal plan obtained in (ii) by restricting Q_1 to the schema of view names is $U(x,y)$:- $V(x,y)^{p_0}$, where p_0 denotes the provenance term of atom $V(x,y)$. The result of backchasing U with $C_{\mathcal{V}}^{OI}$ in phase (iv) is $B(x,y)$:- $V(x,y)^{p_0} \wedge R(x,z)^{p_0} \wedge S(z,y)^{p_0}$. Note that the $\pi(R)$ and $\pi(S)$ of the R and S atoms (a simple term p_0, in this example) are introduced by chasing the view V. Finally, in phase (v) we find one match image given by h from Q's body into the R and S atoms from B's body. The provenance $\pi(h(Q))$ of the image of Q under h is p_0, which corresponds to an equivalent rewriting: $RW(x,y)$:- $V(x,y)$.

[3]Provenance formulas are constructed from provenance terms using logical conjunction and disjunction.

6.3.2 $Prune_{prov}$ ***Minimum-Cost Rewriting***.

Recall from §6.3.1 that the minimal rewritings of a query Q are obtained by first finding the set $\mathcal{H}$ of all matches (i.e., containment mappings) from Q to the result B of backchasing the universal plan U. Denoting with $\pi(S)$ the provenance formula of a set of atoms S, PACB computes the DNF form D of $\bigvee_{h \in \mathcal{H}} \pi(h(Q))$. Each conjunct c of D determines a subquery $sq(c)$ of U which is guaranteed to be a rewriting of Q.

The idea behind cost-based pruning is that, whenever the naïve PACB backchase would add a provenance conjunct c to an existing atom a's provenance formula $\pi(a)$, $Prune_{prov}$ does so more conservatively: if the cost $\gamma(sq(c))$ is larger than the minimum cost threshold T found so far, then c will never participate in a minimum-cost rewriting and need not be added to $\pi(a)$. Moreover, atom a itself need not be chased into B in the first place if all its provenance conjuncts have above-threshold cost.

Example 6.3. *Let $E = M(NM)$, where we assume for simplicity that $M_{50K \times 100}$ and $N_{100 \times 50K}$ are dense. Exploiting the associativity of matrix-multiplication $(MN)M = M(NM)$ during the chase leads to the following universal plan U annotated with provenance terms:*

$$U(R_2): name(M, \text{“}M.csv\text{”})^{p_0} \wedge size(M, \text{“50000”}, \text{“100”})^{p_1} \wedge$$
$$name(N, \text{“}N.csv\text{”})^{p_2} \wedge size(N, \text{“100”}, \text{“50000”})^{p_3} \wedge$$
$$\mathtt{multi}_M(M, N, R_1)^{p_4} \wedge \mathtt{multi}_M(R_1, M, R_2)^{p_5} \wedge$$
$$\mathtt{multi}_M(N, M, R_3)^{p_6} \wedge \mathtt{multi}_M(M, R_3, R_2)^{p_7}$$

Now, consider in the backchase the associativity constraint C:

$$\forall M \forall N \forall R_1 \forall R_2$$
$$\mathtt{multi}_M(M, N, R_1) \wedge \mathtt{multi}_M(R_1, M, R_2) \rightarrow$$
$$\exists R_4 \mathtt{multi}_M(N, M, R_4) \wedge \mathtt{multi}_M(M, R_4, R_2)$$

There exists a match h embedding the two atoms in the premise P of C into the U atoms whose provenance annotations are p_4 and p_5. The provenance conjunct collected from P's image is $\pi(h(P))$=$p_4 \wedge p_5$.

Without pruning, *the backchase would chase U with the constraint C, yielding U' which features additional $\pi(h(P))$-annotated atoms*

$$\mathtt{multi}_M(N, M, R_4)^{p_4 \wedge p_5} \wedge \mathtt{multi}_M(M, R_4, R_2)^{p_4 \wedge p_5}$$

E has precisely two matches h_1, h_2 into U'. $h_1(E)$ involves the newly added atoms as well as those annotated with p_0, p_1, p_2, p_3. Collecting all their provenance annotations yields the conjunct $c_1 = p_0 \wedge p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5$. c_1 determines the U-subquery $sq(c_1)$ corresponding to the rewriting $(MN)M$, of cost $(50K)^2$.

$h_2(E)$'s image yields the provenance conjunct $c_2 = p_0 \wedge p_1 \wedge p_2 \wedge p_3 \wedge p_6 \wedge p_7$, which determines the rewriting $M(NM)$ that happens to be the original expression E of cost 100^2.

The naïve PACB would find both rewritings, cost them, and drop the former ,which introduces a large intermediate result ($\gamma(sq(\pi(h(P))) = (50K)^2$), above the threshold, in favor of the latter.

With pruning, *the threshold T is the cost of the original expression 100^2. The chase step with C is never applied, as it would introduce the provenance conjunct $\pi(h(P))$ which determines U-subquery $sq(\pi(h(P)) = \mathtt{multi}_M(M, N, R_1)^{p_4} \wedge \mathtt{multi}_M(R_1, M, R_2)^{p_5}$ of cost $(50K)^2$ exceeding T. The atoms needed as image of E under h_1 are thus never produced while backchasing U, so the expensive rewriting is never discovered. This leaves only the match image $h_2(E)$, which corresponds to the efficient rewriting $M(NM)$.*

6.3.3 ***Our improvements on*** $Prune_{prov}$. Whenever the pruned chase step is applicable and applied for each constraint, the original algorithm searches for all *minimal-rewritings* $\mathcal{RW}$ that can be found "so far", then it costs each $rw \in \mathcal{RW}$ to find the "so far" *minimum-cost one* rw_e and adjusts the threshold T to the cost of rw_e. However, this strategy can cause *redundant* costing of $rw \in \mathcal{RW}$ whenever the pruned chase step is applied again for another constraint. Therefore, in our modified version of $Prune_{prov}$, we keep track of the rewriting costs already estimated, to prevent such redundant work. Additionally, the search for *minimal-rewritings* "so far" (matches of the query Q into the evolving universal plan instance U', see §6.3.1), whenever the pruned chase step is applied, is modeled as a query evaluation of Q against U' (viewed as a symbolic/canonical database [15]). This involves repeatedly evaluating the same query plan. However, the query is evaluated over evolutions of the same instance. Each pruned chase step *adds a few new tuples* to the evolving instance, corresponding to atoms introduced by that step, while most of the instance is unchanged. Therefore, instead of evaluating the query plan from scratch, we employ incremental evaluation as in [30]. The plan is kept in memory along with the populated hash tables and, whenever new tuples are added to the evolving instance, we push them to the plan.

7 GUARANTEES ON THE REDUCTION

We detail the conditions under which we guarantee that our approach is *sound* (i.e., generates only equivalent, cost-optimal rewritings), and *complete* (i.e., finds all equivalent cost-optimal rewritings).

Let $\mathcal{V} \subseteq \mathcal{L}$ be a set of materialized view definitions, where $\mathcal{L}$ is the language of hybrid expressions described in §3.

Let LA_{prop} be a set of properties of the LA operations in L_{ops} that admits relational encoding over $\mathcal{VREM}$. We say that LA_{prop} is *terminating* if it corresponds to a set of TGDs and EGDs with terminating chase (this holds for our choice of LA_{prop}).

Denote with γ a cost model for expressions from $\mathcal{L}$. We say that γ is *monotonic* if expressions are never assigned a lower cost than their subexpressions (this is true for both models we used).

We call $E \in \mathcal{L}$ $(\gamma, LA_{prop}, \mathcal{V})$*-optimal* if for every $E' \in \mathcal{L}$ that is $(LA_{prop},\mathcal{V})$-equivalent to E we have $\gamma(E') \geq \gamma(E)$.

Let $\text{Eq}^{\gamma}\langle LA_{prop}, \mathcal{V}\rangle(E)$ denote the set of all $(\gamma, LA_{prop}, \mathcal{V})$-optimal expressions that are $(LA_{prop},\mathcal{V})$-equivalent to E.

We denote with $HADAD\langle LA_{prop}, \mathcal{V}, \gamma\rangle$ our parameterized solution based on relational encoding followed by PACB++ rewriting and next by decoding all the relational rewritings generated by the cost-based pruning PACB++ (recall Figure 1). Given $E \in \mathcal{L}$, $HADAD\langle LA_{prop}, \mathcal{V}, \gamma\rangle(E)$ denotes all expressions returned by $HADAD\langle LA_{prop}, \mathcal{V}, \gamma\rangle$ on input E.

Theorem 7.1 (Soundness). *If the cost model γ is monotonic, then for every $E \in \mathcal{L}$ and every $rw \in HADAD\langle LA_{prop}, \mathcal{V}, \gamma\rangle(E)$, we have $rw \in Eq^{\gamma}\langle LA_{prop}, \mathcal{V}\rangle(E)$.*

Theorem 7.2 (Completeness). *If γ is monotonic and LA_{prop} is terminating, then for every $E \in \mathcal{L}$ and every $rw \in Eq^{\gamma}\langle LA_{prop}, \mathcal{V}\rangle(E)$, we have $rw \in HADAD\langle LA_{prop}, \mathcal{V}, \gamma\rangle(E)$.*

8 EXPERIMENTAL EVALUATION

We evaluate HADAD to answer the following questions:

- **§8.1.1, §8.1.2, §8.2.1** and **§8.2.2**: Can HADAD find rewrites with/without views that outperform the original pipelines without modifying the internals of the existing systems? Are the identified rewrites found by state-of-the-art platforms?
- **§8.1.3** : Is HADAD's optimization overhead compensated by the performance gains in execution?

We evaluate our approach, first on **pure LA pipelines** (§8.1), then on **hybrid expressions** (§8.2). Due to space constraints, we highlight some of our results here, relegating the full details to [12].

Experimental Environment. We used a single node with an Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz, and 123G RAM. We run on OpenJDK Java 8 VM. As for **LA systems/libraries**, we used **R 3.6.0**, **Numpy 1.16.6 (python 2.7)**, **TensorFlow r1.4.2**, **Spark 2.4.5 (MLlib)**, and **SystemML 1.2.0**. For hybrid experiments, we use **MorpheusR**[5] and **SparkSQL** [18].

Systems Configuration Tuning. We use a JVM-based linear algebra library for SystemML as recommended in [46], at the optimization level **4**. Additionally, we enable multi-threaded matrix operations in a single node. We run Spark in a single node setting using OpenBLAS (built from the sources as detailed in [6]) to take advantage of its accelerations [46]. SparkMLlib's datatypes do not support many basic LA operations, such as Hadamard-product, To support them, we use the `Breeze` Scala library [2], convert MLlib's datatypes to Breeze types and express the basic LA operations in Spark. The driver memory allocated for Spark and SystemML is 115GB. To maximize TensorFlow performance, we compile it from sources. For all systems/libraries, we set the number of **cores** to **24** and use **double-precision** numbers.

8.1 LA-based Experiments

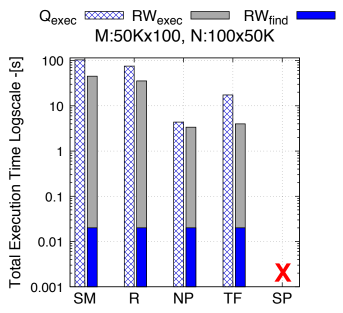
In these experiments, we study the performance benefits of our approach on LA-based pipelines as well as our optimization overhead.

Datasets. We use four **real-world, sparse matrices**. We present two of them here and describe the others in [12]. We use an Amazon books review [1](in JSON) and a Netflix movie rating datasets [7]. The two were converted into matrices where columns are items and rows are customers [47]. We extract subsets of them to ensure various computations applied on them fit in memory (e.g., NS denotes the small version of the Netflix dataset and AL denotes the large version for Amazon). We also use a set of **synthetic, dense matrices**; their dimensions appear at the top of experiment figures.

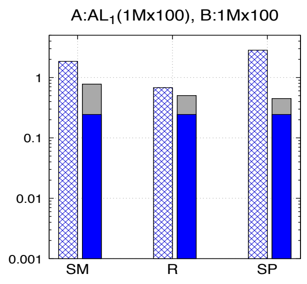
LA benchmark. We select a set $\mathcal{P}$ of **57 LA pipelines** used in prior studies and/or frequently occurring in real-world LA computations, as follows:

- **Real-world pipelines (10)** include expressions used in ALS (P1.7) [39], Ordinary Least Squares Regression (OLS) [46] (P1.6) and matrix chain multiplication (P1.3 and P1.5). Others are detailed in [12].
- **Synthetic pipelines (47)** were also generated, based on a set of basic matrix operations (inverse, multiplication, addition, etc.), and a set of combination templates, written as a Rule-Iterated Context-Free Grammar (RI-CFG) [41].

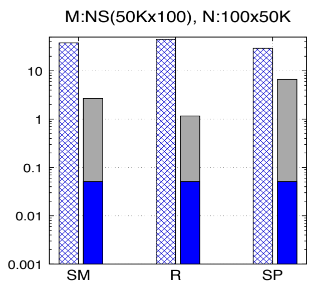
Table 2 lists pipelines that we discus here. All 57 benchmark pipelines appear in [12] (§9.1, Tables 2 and 3).



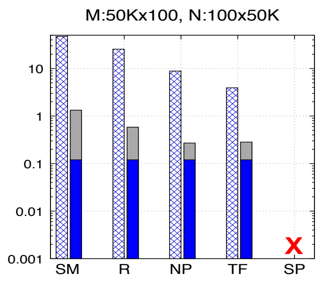
(a) **P1.1**



(b) **P1.2**



(c) **P1.3**



(d) **P1.4**

Figure 5: P1.1, P1.2, P1.3, and P1.4 Evaluation Before and After Rewriting

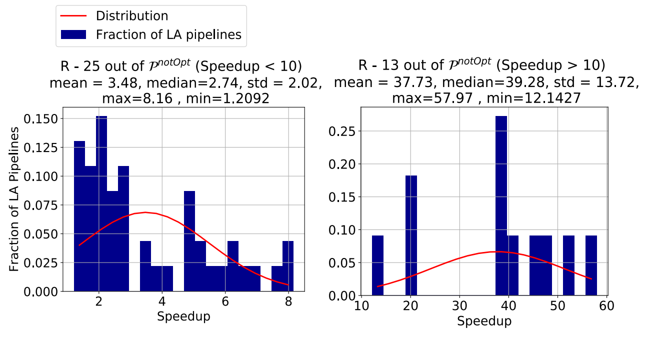


Figure 6: R speed-up on $\mathcal{P}^{\neg Opt}$

No.	Expression	No.	Expression
P1.1	$(MN)^T$	**P1.2**	$(A+B)v_1$
P1.3	$(MN)M$	**P1.4**	sum(rowSums($N^T M^T$))
P1.5	$((MN)M)N$	**P1.6**	$(D^T D)^{-1}(D^T v_1)$
P1.7	$(uv^T - X)v$	**P1.8**	$((((C+D)^T)^{-1})D)C$

Table 2: Subset of LA Benchmark Pipelines

Methodology. In §8.1.1, we show the performance benefits of HADAD to LA systems/tools mentioned above using a set $\mathcal{P}^{\neg Opt} \subset \mathcal{P}$ of **38** pipelines. The performance of $\mathcal{P}^{\neg Opt}$ can be improved *just by exploiting LA properties* (in the absence of views). For TensorFlow and NumPy, we present the results only for dense matrices, due to limited support for sparse matrices. In §8.1.2, we show how our approach improves the performance of **30** pipelines from $\mathcal{P}$, denoted $\mathcal{P}^{Views}$, using pre-materialized views. Finally, in §8.1.3, we study our rewriting performance and *optimization overhead* for the set $\mathcal{P}^{Opt} = \mathcal{P} \setminus \mathcal{P}^{\neg Opt}$ of **19** pipelines that are already optimized.

8.1.1 ***Effectiveness of LA Rewriting (No Views).*** For each system, we run the original pipeline and our rewriting 5 times; we report the average of the last 4 running times. We exclude the data loading time. For fairness, we ensured SparkMLib and SystemML compute the entire pipeline (despite their lazy evaluation mode).

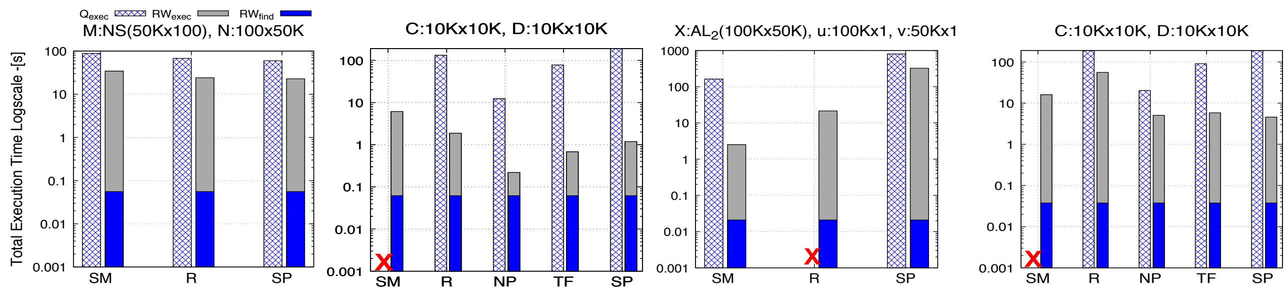
Figure 5 illustrates the original pipeline execution time Q_{exec} and the selected rewriting execution time RW_{exec} for **P1.1**, **P1.2**, **P1.3**, and **P1.4**, including the rewriting time RW_{find}, using the MNC cost model from §6.2.2. For each pipeline, the used datasets are on top of the figure. For brevity in the figures, we use **SM** for SystemML, **NP** for NumPy, **TF** for Tensorflow, and **SP** for MLlib.

For **P1.1** (see Figure 5(a)), both matrices are dense. The speed-up (1.3× to 4×) comes from rewriting $(MN)^T$ (intermediate result size is $(50K)^2$) into $N^T M^T$, much cheaper since both N^T and M^T are of size $50K \times 100$. We exclude MLlib from this experiment since it failed to allocate memory for the intermediate matrix (Spark/MLLib limits the maximum size of a dense matrix). As a variation (not plotted in the Figure), we ran the same pipeline with the ultra-sparse Amazon AS : $50K \times 100$ matrix (0.0075% nnz) used as M. The Q_{exec} and RW_{exec} time are very comparable using SystemML, because we avoid dense intermediates and multiplication becomes efficient. In R, this scenario leads to a runtime exception and to avoid it, we cast M during load time to a dense matrix. Thus, the speed-up achieved is the same as if M and N were both dense. If, instead, NS : $50K \times 100$ (1.3911% non-zeros) plays the role of M, our rewrite achieves $\approx 1.8\times$ speed-up for SystemML.

For **P1.2** (Figure 5(b)), we rewrite $(A+B)v_1$ to $Av_1 + Bv_1$. Adding Amazon sparse matrix AL_1 (0.0065% nnz) used as A to a dense matrix B results into materializing a dense intermediate of size $1M \times 100$. Instead, $Av_1 + Bv_1$ has fewer nnz in the intermediate results, and Av_1 can be computed efficiently since A is sparse. The MNC sparsity estimator has a noticeable overhead. We run the same pipeline, where the dense $5M \times 100$ matrix plays both A and B (not shown in the Figure). This leads to speed-up of up to 9× for MLlib, which does not natively support matrix addition, thus we convert its matrices to Breeze types in order to perform it [46].

The intermediate matrix size for evaluation plan $(MN)M$ of **P1.3** (Figure 5(c)) is $(50K)^2$, but only 100^2 for its rewriting $M(NM)$. To avoid MLLib memory failure, we use BlockMatrix type for both matrices. While M thus converted has the same sparsity, Spark views it as being of a dense type (multiplication on BlockMatrix produces dense matrices) [11]. SystemML does optimize the chain order if the user does not enforce it. Further (not shown in the Figure), we ran **P1.3** with AS in the role of M. This is 4× faster in SystemML since with an ultra sparse M, multiplication is more efficient. This is not the case for MLlib which views it as dense. For R, we again had to densify M during loading to prevent crashes.

Figure 5(d) shows speedups of up to 42× when rewriting **P1.4** into sum((colSums(M))$^T \odot$rowSums(N)). This exploits several properties:
(i) $(MN)^T = N^T M^T$,
(ii) sum(M^T) =sum(M),
(iii) sum(rowSums-(M)) = sum(M), and
(iv) sum(MN)=sum((colSums(M))$^T \odot$ rowSums-(N)).
SystemML captures (ii), (iii), and (iv) as rewrite rules, however, it is unable to exploit them here since it is unaware of (i). Other systems do not exploit these properties.



(a) **P1.5** (b) **P1.6** (c) **P1.7** (d) **P1.8**

Figure 7: P1.5, P1.6, P1.7 and P1.8 Evaluation Before and After Rewriting Using the Views V_{exp}

Speed-up Summary. Figure 6 shows the distribution of the significant rewriting speed-up on $\mathcal{P}^{\neg Opt}$ running on R (other systems' results are in [12]) , and using the MNC-based cost model. For clarity, we split the distribution into two figures: on the left, 25 $\mathcal{P}^{\neg Opt}$ pipelines with speed-up lower than 10×; on the right, the remaining 13 with greater speed-up. Among the former, 87% achieved at least 1.5× speed-up. The latter are sped up by 10× to 60×. The $((D)^{-1})^{-1}$ pipeline is an extreme case here (not plotted): it is sped up by about 1000×, simply by rewriting $((D)^{-1})^{-1}$ into D (which R fails to do).

8.1.2 *Effectiveness of view-based LA rewriting.*

We have defined a set V_{exp} of 12 views that pre-compute the result of some expensive operations (multiplication, inverse, determinant, etc.) which can be used to answer our $\mathcal{P}^{Views}$ pipelines, and materialized them on disk as CSV files. The experiments outlined below used the naïve cost model; all graphs have a log-scale y axis.

For **P1.5** (Figure 7(a)), using the view $V_4 = NM$ and the multiplication associativity leads to up to 2.8× speed-up. Figure 7(b) shows the performance gain for OLS pipeline **P1.6**, discussed in (§2). This rewrite leads to up to 150× speed-up on MLlib. On SystemML, the original pipeline timed out (> 1000 seconds).

P1.7 (Figure 7(c)) benefits from a view V_5, which pre-computes a dense intermediate uv^T; then, rewriting based on the property $(A+B)v = Av+Bv$ leads to a 65× speed-up in SystemML. For MLlib, as discussed before, to avoid memory failure, we used BlockMatrix types for all matrices and vectors, thus they were treated as dense. In R, the original pipeline triggers a memory allocation failure, which the rewriting avoids.

Figure 7(d) shows that for **P1.8** exploiting the views $V_2 = (D+C)^{-1}$ and $V_3 = DC$ leads to speed-ups of 4× to 41× on different systems. Properties enabling rewriting here are $C + D = D + C$, $(D^T)^{-1} = (D^{-1})^T$ and $(CD)E = C(DE)$.

8.1.3 *Rewriting Performance and Overhead.*

We now study the running time RW_{find} of our rewriting algorithm, and the *rewrite overhead* defined as $RW_{find}/(Q_{exec}+ RW_{find})$, where Q_{exec} is the time to run the pipeline "as stated". We ran each experiment 100 times and report the average of the last 99 times. The global trends are as follows: (*i*) For a fixed pipeline and set of data matrices, *the overhead is slightly higher using the MNC cost model*, since histograms are built during optimization. (*ii*) For a fixed pipeline and cost model, *sparse matrices lead to a higher overhead* simply because Q_{exec} tends to be smaller. (*iii*) *Some* (system, pipeline) pairs lead to a low Q_{exec} when *the system applies internally the same optimization* that HADAD finds "outside" of the system.

Concretely, for the $\mathcal{P}^{\neg Opt}$ pipelines, on the dense and sparse matrices, using the naïve cost model, 64% of the RW_{find} times are under 25ms (50% are under 20ms), and the longest is about 200m. Using the MNC estimator, 55% took less than 20ms, and the longest (outlier) took about 300ms. Among the 38 $\mathcal{P}^{\neg Opt}$ pipelines, SystemML finds efficient rewritings for a set of 9, denoted $\mathcal{P}^{\neg Opt}_{SM}$, while TensorFlow optimizes a set of 11, denoted $\mathcal{P}^{\neg Opt}_{TF}$. On these subsets, where HADAD's optimization is redundant, using *dense* matrices, the overhead is very low: with the *MNC* model, 0.48% to 1.12% on $\mathcal{P}^{\neg Opt}_{SM}$ (0.64% on average), and 0.0051% to 3.51% on $\mathcal{P}^{\neg Opt}_{TF}$ (1.38% on average). Using the naïve estimator slightly reduces this overhead, but across $\mathcal{P}^{\neg Opt}$, this model misses 4 efficient rewritings. On *sparse* matrices using SystemML, the overhead is at most 4.86% with the naïve estimator and up to 5.11% with the MNC one.

Among the already optimal pipelines $\mathcal{P}^{Opt}$, 70% involve expensive operations such as inverse, determinant, etc. , leading to rather long Q_{exec} times. Thus, the rewriting overhead is less than 1% of the total time, on all systems, using sparse or dense matrices, and the naïve or MNC estimators. For the other $\mathcal{P}^{Opt}$ pipelines with short Q_{exec}, mostly multiplication chains are already optimized , on *dense* matrices, the overhead reaches 0.143% (SparkMlLib) to 9.8% (TensorFlow) using the naïve cost model, while the MNC cost model leads to an overhead of 0.45% (SparkMlib) up to 10.26% (TensorFlow). On *sparse* matrices, using the naïve and MNC cost models, the overhead is up to 0.18% (SparkMLlib) to 1.94% (SystemML), and 0.5% (SparkMLlib) to 2.61% (SystemML), respectively.

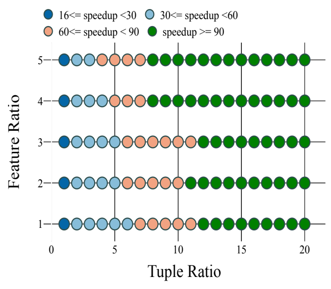
8.2 Hybrid Setting

We now study the benefits of rewriting on hybrid scenarios combining RA and LA operations. In §8.2.1, we show the performance benefits of HADAD to a cross RA-LA platform, MorpheusR [5]. We evaluate our hybrid benchmark on SparkSQL+SystemML in §8.2.2.

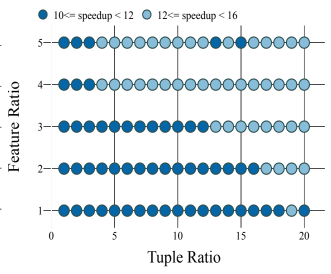
8.2.1 *MorpheusR Experiment.*

We use the same experimental setup introduced in [28] for generating synthetic datasets for the PK-FK join of tables **R** and **S**. The quantities varied are the *tuple ratio* (n_S/n_R) and feature ratio (d_R/d_S), where n_S and n_R are the number of rows and d_R and d_S are the number of columns (features) in **R** and **S**, respectively. We fix $n_R = 1M$ and $d_S = 20$. The join of **R** and **S** outputs $n_s \times (d_R + d_S)$ matrix **M**, which is always dense. We evaluate on Morpheus a set of 8 pipelines and their rewritings found by HADAD using the naïve cost model.

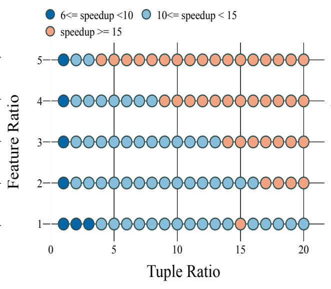
Discussion. **P1.9**: `colSums`(**M**N) is the example from §2, with **M** the output (viewed as matrix) of joining tables **R** and **S** generated



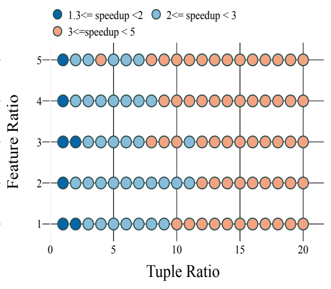
(a) **P1.9**



(b) **P1.10**



(c) **P1.11**



(d) **P1.12**

Figure 8: Speed-ups of Morpheus (with HADAD rewrites) over Morpheus (without HADAD rewrites)

as described above. N is a $ncol(\mathbf{M}) \times 100$ dense matrix. HADAD's rewriting yields up to 125× speed-up (see Figure 8(a)).

Figure 8(b) shows up to 15× speed-up for **P1.10**: rowSums($N\mathbf{M}$), where the size of N is $100 \times nrow(\mathbf{M})$. This is due to HADAD's rewriting: NrowSums($\mathbf{M}$), which enables Morpheus to push the rowSums operator to **R** and **S** instead of computing the large intermediate matrix multiplication.

P1.11: sum(N+$\mathbf{M}$) is run *as-is* by Morpheus since it does not factorize element-wise operations, e.g., addition. However, HADAD rewrites **P1.11** into sum(N)+sum($\mathbf{M}$), which avoids the (large and dense) intermediate result of the element-wise matrix addition. The HADAD rewriting enables Morpheus to execute sum($\mathbf{M}$) by pushing sum to **R** and **S**, for up to 20× speed-up (see Figure 8(c)).

Morpheus evaluates **P1.12**: sum(rowSums($\mathbf{M}$)) by pushing the rowSums operator to **R** and **S**. HADAD finds the rewriting sum($\mathbf{M}$), which enables Morpheus to push the sum operation instead, achieving up to 4.5× speed-up (see Figure 8(d)).

Since Morpheus does not exploit the associative property of matrix multiplication, it cannot reorder multiplication chains to avoid large intermediate results, which lead to runtime exception in R (MorpheusR's backend). For example, for the chain **P1.13**: $N^T N\mathbf{M}$, when the size of $\mathbf{M}$ is 1M×40 and the size of N is 40×1M, the size of the $N^T N$ intermediate result is 1M×1M, which R cannot handle (timed out >1000 seconds). HADAD exploits associativity and selects the rewriting $N^T(N\mathbf{M})$ of intermediate result size (40×40).

8.2.2 *Hybrid Micro-Benchmark Experiment.*

We create a *hybrid micro-benchmark* of ten queries on the real-life Twitter[13] and MIMIC [31] datasets to empirically study HADAD's rewriting benefits in a hybrid setting. We only include a subset of the Twitter dataset results here, detailing the others in [12].

Dataset Preparation. We obtain from Twitter API [13] 16GB of tweets (in JSON). We extract the structural parts of the dataset, which include user and tweet information, and store them in tables **User (U)** and **Tweet (T)**, linked via PK-FK relationships. The dataset is detailed in §2.

Queries and Views. Queries consist of two parts: (*i*) RA preprocessing (Q_{RA}) and (*ii*) LA analysis (Q_{LA}). In the Q_{RA} part, queries construct two matrices: **M** and **N**. The matrix **M** (2M×12;*dense*) is the output of joining **T** and **U**. The construction of matrix **N** is described in §2. We fix the Q_{RA} part across all queries and vary the Q_{LA} part using a set of LA pipelines detailed below.

In addition to the views defined in §2, we define three hybrid RA-LA materialized views: V_3, V_4 and V_5, which store the result of applying rowSums, colSums and matrix multiplication operations over base tables **T** and **U** (viewed as matrices).

Importantly, rewritings based on these views can only be found by *exploiting together LA properties* and *Morpheus' rewrites rules* (we incorporated them in our framework as a set of integrity constraints). The full list of queries and views is in [12].

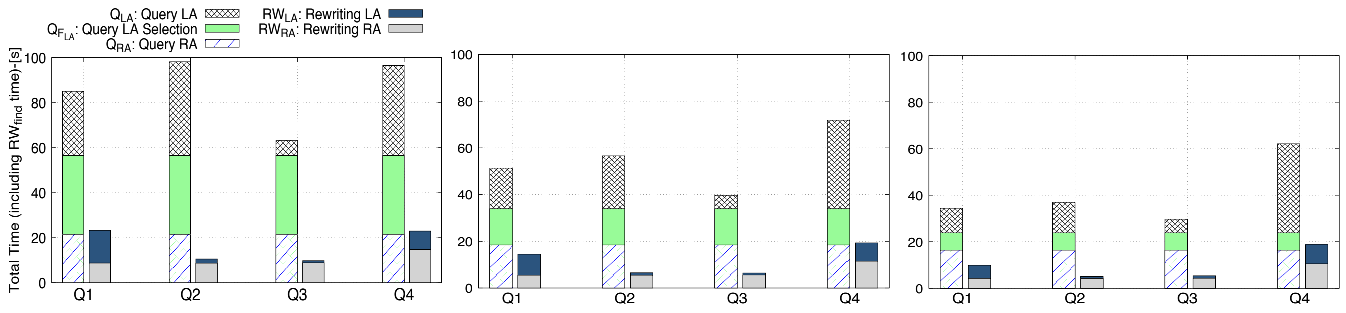
Discussion. After construction of **M** and **N** by the Q_{RA} part in SparkSQL, both matrices are loaded to SystemML to be used in the Q_{LA} part. Before evaluating an LA pipeline on **M** and **N**, all queries select **N** 's rows ($Q_{F_{LA}}$) with *filter-level* less than 4 (medium). For all of them, HADAD rewrites the Q_{RA} part of **N** as described in §2.

Q1: In the Q_{LA} part, **Q1** runs sum(C+**N**rowSums($X\mathbf{M}$)v) (inspired by the COX proportional hazard regression model used in SystemML's test suite [3]), where synthetic dense matrices C, X and v have size 2M×1000,1000×2M and 1×1000, respectively. HADAD (*i*) distributes the sum operation to avoid materializing the dense addition (SystemML includes this rewrite rule but fails to apply it); (*ii*) rewrites rowSums($X\mathbf{M}$) to XV_3, where V_3 =rowSums($\mathbf{T}$) +KrowSums($\mathbf{U}$), by exploiting rowSums($X\mathbf{M}$) = XrowSums($\mathbf{M}$) together with Morpheus's rewrite rule rowSums($\mathbf{M}$) $\rightarrow$ rowSums($\mathbf{T}$)+KrowSums($\mathbf{U}$)[4]. The multiplication chain in the rewriting is efficient since **N** is sparse. The rewriting of this query (including the rewriting of the Q_{RA} part of **N**) achieves 3.63× speed-up (Figure 9(a)-Q1).

Q2: The query runs ((**N**+X)v)colSums($\mathbf{M}$) in the Q_{LA} part, where dense matrices X and v are of size 2M×1000 and 1000×1, respectively. HADAD avoids the dense intermediate (**N**+X) by distributing the multiplication by v and realizing that the sparsity of **N** yields efficient multiplication. It also directly rewrites colSums($\mathbf{M}$) to V_4, where V_4 = [colSums($\mathbf{T}$),colSums(K)$\mathbf{U}$] by utilizing one of Morpheus's rewrite rules. The rewriting, including the rewriting of the Q_{RA} part of **N**, achieves 9.2× speed-up (Figure 9(a)-Q2).

Q3: The Q_{LA} part executes $\mathbf{N} \odot$trace(C+$v$colSums($\mathbf{M}X$)C), where the size of v, X and C are 20K×1, 12×20K and 20K×20K, respectively. First, HADAD distributes the trace operation (which SystemML does not apply) to avoid the dense intermediate addition. Second, HADAD enables the exploitation of view V_4 (see **Q2**) by utilizing colSums($\mathbf{M}X$) = colSums($\mathbf{M}$)X. The resulting multiplication chain is optimized by the order $v((V_4X)C)$. The final element-wise multiplication with **N** is efficient since **N** is ultra-sparse. The combined Q_{RA} and Q_{LA} rewriting speeds up **Q3** by 5.94× (Figure 9(a)-Q3).

[4]K is the unique sparse indicator matrix that captures the primary/foreign key dependencies between **T** and **U**, introduced by Morpheus's rewrite rules [28].



(a) **N matrix: 2M rows** (b) **N matrix: 1M rows** (c) **N matrix: 0.5M rows**

Figure 9: Hybrid Micro-Benchmark Twitter Dataset

Q4: HADAD's rewrite the Q_{LA}: $\mathbf{N}\odot\mathtt{sum}((X+C)\mathbf{M})$, where X and C are dense matrices of size 1000×1M, to $\mathbf{N}\odot\mathtt{sum}(X\mathbf{M})+\mathtt{sum}(V_5)$. The $V_5 = [C\mathbf{T}, CK\mathbf{U}]$ is utilized by exploiting $(X+C)\mathbf{M}= X\mathbf{M}+C\mathbf{M}$ together with Morpheus's rewrite rule: $C\mathbf{M}\rightarrow [C\mathbf{T}, CK\mathbf{U}]$. This optimization goes beyond SystemML's optimization since it does not consider distributing the multiplication of **M**, which then enables exploiting the view and distributing the sum operation to avoid a dense intermediate. The obtained rewriting (with the rewriting of the Q_{RA} part of **N**) achieves 3.91× (Figure 9(a)-Q4).

Varying Filter Selectivity. We repeat the benchmark for two different text-search selection conditions: "Trump" and "US election", obtaining 1M and 0.5M rows for **N**, respectively (we adjust the size of the synthetic matrices for dimensional compatibility). As shown in Figures 9(b) and 9(c), the benefit of the combined Q_{RA} and Q_{LA} stage rewriting increases with data size, remaining significant across the spectrum.

8.3 Experiment Takeaway

Our experiments with both real-life and synthetic datasets show performance gains across the board, for small rewriting overhead, in both pure LA and hybrid RA-LA settings. This is due to the fact that HADAD's rewriting power strictly subsumes that of optimizers of reference platforms like R, Numpy, TensorFlow, Spark (MLlib), SystemML and Morpheus. Moreover, HADAD enables optimization where it wasn't previously feasible, such as across a cascade of unintegrated tools, e.g. SparkSQL for preprocessing followed by SystemML for analytics.

9 RELATED WORK AND CONCLUSION

LA Systems/Libraries. SystemML [22] offers high-level R-like LA language and applies some logical LA pattern-based rewrites and physical execution optimizations, based on cost estimates for the latter. SparkMLlib [40] provides LA operations and built-in function implementations of popular ML algorithms on Spark RDDs. R [9] and NumPy [8] are two of the most popular computing environments for statistical data analysis, widely used in academia and industry. They provide a high-level abstraction that can simplify the programming of numerical and statistical computations, by treating matrices as first-class citizens and by providing a rich set of built-in LA operations. However, LA properties in most of these systems remain unexploited, which makes them *miss opportunities to use their own highly efficient operators* (recall the hybrid scenario in §2). Our experiments (§8.1) show that LA pipelines evaluation in these systems can be sped up, by more than 10×, by our rewriting using (*i*) LA properties and (*ii*) materialized views.

Bridging the Gap: RA and LA. There has been a recent increase in research for unifying the execution of RA and LA expressions [10, 28, 33, 38]. A key limitation of these approaches is that *the semantics of LA operations remains hidden* behind built-in functions or UDFs, preventing performance-enhancing rewrites as shown in §8.2.1.

SPORES [47], SPOOF [24], LARA [36] and RAVEN [32] are closer to our work. SPORES and SPOOF optimize LA expressions, by converting them into RA, optimizing the latter, and then converting the result back to an (optimized) LA expression. They are restricted to a small set of selected LA operations (the ones that can be expressed in RA), while we support significantly more (§5.1), and model properties allowing to optimize with them. LARA relies on a declarative domain-specific language for collections and matrices, which can enable optimization (i.e., *selection pushdown*) across the two algebraic abstractions. It heavily focuses on low-level optimization such as exploiting the choice of data layouts for physical LA operators' optimization. RAVEN takes a step forward by providing intermediate representation to enhance in-database model inferencing performance. It transforms classical ML models into equivalent neural networks to leverage highly optimized ML engines on CPU/GPU. [20, 26] study the expressive power of corss RA-LA [20] / LA [26] query languages.

In contrast to HADAD, as all aforementioned solutions do not reason with constraints, they provide no capabilities for *holistic* semantic query optimizations including RA/LA views-based and LA pure rewritings; such optimizations can bring large performance saving as shown in §8. We see HADAD as complementary to all of these platforms, on top of which it can be portably applied.

Conclusion. HADAD is an extensible lightweight framework for optimizing hybrid analytics queries, based on the powerful intermediate abstraction of *a relational model with integrity constraints*. HADAD extends the capability of [16] with a reduction from LA (or LA view)-based rewriting to relational rewriting under constraints. It enables a full exploration of rewrites using a large set of LA operations, with no modification to the execution platform. Our experiments show significant performance gains on various LA and hybrid workloads across popular LA and cross RA-LA platforms.

ACKNOWLEDGMENTS

This work is supported by a graduate fellowship from KACST.

REFERENCES

[1] Amazon Review Data. https://nijianmo.github.io/amazon/index.html, Accessed August, 2020.

[2] Breeze Wiki. https://github.com/scalanlp/breeze/wiki, Accessed June, 2020.

[3] Cox Proportional-Hazards Model. https://github.com/apache/systemds/blob/master/scripts/algorithms/Cox-predict.dml, Accessed Feb, 2021.

[4] Kaggle Survey. https://www.kaggle.com/kaggle-survey-2019, Accessed June, 2020.

[5] Morpheus. https://github.com/lchen001/Morpheus, Accessed December, 2020.

[6] Native Blas in SystemDS. https://apache.github.io/systemds/native-backend, Accessed June, 2020.

[7] Netflix Movie Rating. https://www.kaggle.com/netflix-inc/netflix-prize-data, Accessed August, 2020.

[8] NumPy. https://numpy.org/, Accessed June, 2020.

[9] Project R. https://www.r-project.org/other-docs.html, Accessed June, 2020.

[10] Python/NumPy in Monetdb. https://tinyurl.com/11ljy21v, Accessed June, 2020.

[11] SparkMLlib. https://spark.apache.org/mllib, Accessed June, 2020.

[12] Technical Report. https://arxiv.org/abs/2103.12317.

[13] Twitter API. https://developer.twitter.com/en/docs, Accessed January, 2021.

[14] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A System for Large-Scale Machine Learning. In *USENIX*, pages 265–283, 2016.

[15] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[16] R. Alotaibi, D. Bursztyn, A. Deutsch, I. Manolescu, and S. Zampetakis. Towards Scalable Hybrid Stores: Constraint-based Rewriting to the Rescue. In *SIGMOD*, pages 1660–1677, 2019.

[17] R. Alotaibi, B. Cautis, A. Deutsch, M. Latrache, I. Manolescu, and Y. Yang. ESTOCADA: Towards Scalable Polystore Systems. *PVLDB*, pages 2949–2952, 2020.

[18] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. SparkSQl: Relational Data Processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[19] S. Axler. *Linear Algebra Done Right.* Springer, 2015.

[20] P. Barceló, N. Higuera, J. Pérez, and B. Subercaseaux. On the Expressiveness of LARA: A Unified Language for Linear and Relational Algebra. In *ICDT*, pages 6:1–6:20, 2020.

[21] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, et al. Tfx: A Tnsorflow-based Production-Scale Machine Learning Platform. In *SIGKDD*, pages 1387–1395, 2017.

[22] 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, et al. SystemML: Declarative Machine Learning on Spark. *PVLDB*, pages 1425–1436, 2016.

[23] M. Boehm, A. V. Evfimievski, N. Pansare, and B. Reinwald. Declarative Machine Learning Classification of Basic Properties and Types. *arXiv:1605.05826*, 2016.

[24] M. Boehm, B. Reinwald, D. Hutchison, P. Sen, A. V. Evfimievski, and N. Pansare. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *PVLDB*, pages 1755–1768, 2018.

[25] J.-H. Böse, V. Flunkert, J. Gasthaus, T. Januschowski, D. Lange, D. Salinas, S. Schelter, M. Seeger, and Y. Wang. Probabilistic Demand Forecasting at Scale. *PVLDB*, pages 1694–1705, 2017.

[26] R. Brijder, F. Geerts, J. V. den Bussche, and T. Weerwag. On the Expressive Power of Query Languages for Matrices. *ACM Trans. Database Syst.*, pages 15:1–15:31, 2019.

[27] A. K. Chandra and P. M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Databases. In *ACM symposium on Theory of computing*, pages 77–90, 1977.

[28] L. Chen, A. Kumar, J. Naughton, and J. M. Patel. Towards Linear Algebra Over Normalized Data. *PVLDB*, pages 1214–1225, 2017.

[29] I. Ileana. *Query Rewriting Using Views : a Theoretical and Practical Perspective.* Theses, Télécom ParisTech, Oct. 2014.

[30] I. Ileana, B. Cautis, A. Deutsch, and Y. Katsis. Complete Yet Practical Search for Minimal Query Reformulations Under Constraints. In *SIGMOD*, pages 1015–1026, 2014.

[31] A. Johnson et al. MIMIC-III. *http://www.nature.com/articles/sdata201635*, 2016.

[32] K. Karanasos, M. Interlandi, D. Xin, F. Psallidas, R. Sen, K. Park, I. Popivanov, S. Nakandal, S. Krishnan, M. Weimer, et al. Extending Relational Query Processing with ML Inference. In *CIDR*, 2020.

[33] D. Kernert, F. Köhler, and W. Lehner. Bringing Linear Algebra Objects to Life in a Column-Criented in-Memory Database. In *In Memory Data Management and Analysis*, pages 44–55. Springer, 2013.

[34] A. Kumar, M. Boehm, and J. Yang. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *SIGMOD*, pages 1717–1722, 2017.

[35] A. Kumar, R. McCann, J. Naughton, and J. M. Patel. Model Selection Management Systems: The Next Frontier of Advanced Analytics. In *SIGMOD*, pages 17–22, 2016.

[36] A. Kunft, A. Katsifodimos, S. Schelter, S. Breß, T. Rabl, and V. Markl. An Intermediate Representation for Optimizing Machine Learning Pipelines. *PVLDB*, pages 1553–1567, 2019.

[37] K. Kuttler. *Linear Algebra: Theory and Applications.* The Saylor Foundation, 2012.

[38] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. Scalable Linear Algebra on a Relational Database System. In *ICDE*, pages 1224–1238, 2018.

[39] J. MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, pages 281–297, 1967.

[40] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. MLlib: Machine Learning in Apache Spark. *The Journal of Machine Learning Research*, pages 1235–1241, 2016.

[41] M. Milani, S. Hosseinpour, and H. Pehlivan. Rule-based Production of Mathematical Expressions. *Mathematics*, 6:254, 11 2018.

[42] T. M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[43] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison. Hidden Technical Debt in Machine Learning Systems. In *NeurIPS*, pages 2503–2511, 2015.

[44] J. Sommer, M. Boehm, A. V. Evfimievski, B. Reinwald, and P. J. Haas. MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions. In *SIGMOD*, pages 1607–1623, 2019.

[45] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating Model Search for Large Scale Machine Learning. In *ACM Symposium on Cloud Computing*, pages 368–380, 2015.

[46] A. Thomas and A. Kumar. A Comparative Evaluation of Systems for Scalable Linear Algebra-based Analytics. *PVLDB*, pages 2168–2182, 2018.

[47] Y. R. Wang, S. Hutchison, J. Leang, B. Howe, and D. Suciu. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *PVLDB*, pages 1919–1932, 2020.