SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra

Yisu Remy Wang
University of Washington
remywang@uw.edu

Shana Hutchison
University of Washington
shutchis@uw.edu

Jonathan Leang
University of Washington
jleang@uw.edu

Bill Howe
University of Washington
billhowe@uw.edu

Dan Suciu
University of Washington
suciu@uw.edu

ABSTRACT

Machine learning algorithms are commonly specified in linear algebra (LA). LA expressions can be rewritten into more efficient forms, by taking advantage of input properties such as *sparsity*, as well as program properties such as *common subexpressions* and *fusible operators*. The complex interaction among these properties' impact on the execution cost poses a challenge to optimizing compilers. Existing compilers resort to intricate heuristics that complicate the codebase and add maintenance cost, but fail to search through the large space of equivalent LA expressions to find the cheapest one. We introduce a general optimization technique for LA expressions, by converting the LA expressions into Relational Algebra (RA) expressions, optimizing the latter, then converting the result back to (optimized) LA expressions. The rewrite rules we design in this approach are complete, meaning that any equivalent LA expression is covered in the search space. The challenge is the major size of the search space, and we address this by adopting and extending a technique used in compilers, called *equality saturation*. Our optimizer, SPORES, uses rule sampling to quickly cover vast portions of the search space; it then uses a constraint solver to extract the optimal plan from the covered space, or alternatively uses a greedy algorithm to shorten compile time. We integrate SPORES into SystemML and validate it empirically across a spectrum of machine learning tasks; SPORES can derive all existing hand-coded optimizations in SystemML, and perform new optimizations that lead to up to 10X speedup.

PVLDB Reference Format:
Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, Dan Suciu. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *PVLDB*, 13(11): 1919-1932, 2020.
DOI: https://doi.org/10.14778/3407790.3407799

1. INTRODUCTION

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407799

Consider the Linear Algebra (LA) expression $sum((X - UV^T)^2)$ which defines a typical loss function for approximating a matrix X with a low-rank matrix UV^T. Here, $sum()$ computes the sum of all matrix entries in its argument, and A^2 squares the matrix A element-wise. Suppose X is a sparse, 1M x 500k matrix, and suppose U and V are dense vectors of dimensions 1M and 500k respectively. Thus, UV^T is a rank 1 matrix of size 1M x 500k, and computing it naively requires 0.5 trillion multiplications, plus memory allocation. Fortunately, the expression is equivalent to $sum(X^2) - 2U^TXV + U^TU * V^TV$. Here U^TXV is a scalar that can be computed efficiently by taking advantage of the sparsity of X, and, similarly, U^TU and V^TV are scalar values requiring only 1M and 500k multiplications respectively.

Optimization opportunities like this are ubiquitous in machine learning programs. State-of-the-art optimizing compilers such as SystemML [3], OptiML[38], and Cumulon[16] commonly implement syntactic rewrite rules that exploit the algebraic properties of the LA expressions. For example, SystemML includes a rule that rewrites the preceding example to a specialized operator[1] to compute the result in a streaming fashion. However, such syntactic rules fail on the simplest variations, for example SystemML fails to optimize $sum((X + UV^T)^2)$, where we just replaced $-$ with $+$. Moreover, rules may interact with each other in complex ways. In addition, complex ML programs often have many common subexpressions (CSE), that further interact with syntactic rules, for example the same expression UV^T may occur in multiple contexts, each requiring different optimization rules.

In this paper we describe SPORES, a novel optimization approach for complex linear algebra programs that leverages relational algebra as an intermediate representation to completely represent the search space. SPORES first transforms LA expressions into traditional Relational Algebra (RA) expressions consisting of joins, unions and aggregates. It then performs a cost-based optimizations on the resulting Relational Algebra expressions, using only standard identities in RA. Finally, the resulting RA expression is converted back to LA, and executed.

A major advantage of SPORES is that the rewrite rules in RA are complete. Linear Algebra seems to require an

[1]See the SystemML Engine Developer Guide for details on the weighted-square loss operator `wsloss`.

endless supply of clever rewrite rules, but, in contrast, by converting to RA, we can prove that our set of just 13 rules are complete. The RA expressions in this paper are over K-relations [13]; a tuple $X(i,j)$ is no longer true or false, but has a numerical value, e.g. 5, which could be interpreted, e.g., as the multiplicity of that tuple. In other words, the RA expressions that result from LA expressions are interpreted over bags instead of sets. The problem of checking equivalence of queries under bag semantics has a unique history. Chaudhuri and Vardi first studied the containment and equivalence problem under bag semantics, and claimed that two conjunctive queries are equivalent iff they are isomorphic: Theorem 5.2 in [6]. However, a proof was never produced. A rather long proof for this claim was given for the bag-set semantics in [8]. Green provided a comprehensive proof, showing that two unions of conjunctive queries (UCQ) are equivalent under bag semantics iff they are isomorphic, by using sophisticated techniques involving multiple semirings [12]. The completeness result of our 13 rules relies on a similar result, but stated for tensors rather than bags; we present here a simple and self-contained proof in Sec. 2.3. We note that, in contrast, *containment* of two UCQs with bag semantics is undecidable [18]; we do not consider containment in this paper. Finally, we prove that our optimizer rules are sufficient to convert any RA expression into its *canonical form*, i.e. to an UCQ, and thus can, in principle, discover all equivalent rewritings.

However, we faced a major challenge in trying to exploit the completeness of the rules. The search space is very large, typically larger than that encountered in standard database optimizers, because of the prevalence of unions $+$, large number of aggregates $\sum$, and frequent common subexpressions. To tackle this, SPORES adopts and extends a technique from compilers called *equality saturation* [40]. It uses a data structure called the E-Graph [32] to compactly represent the space of equivalent expressions, and equality rules to populate the E-Graph, then leverages constraint solvers to extract the optimal expression from the E-Graph. We extend equality saturation with rule sampling and use a greedy extraction algorithm to quickly cover vast portions of the search space, and trade the guarantee of optimality for shorter compile time.

We have integrated SPORES into SystemML [3], and show that it can derive all hand-coded rules of SystemML. We evaluated SPORES on a spectrum of machine learning tasks, showing competitive performance improvement compared with more mature heuristic-based optimizers. Our optimizer rediscovers all optimizations by the latter, and also finds new optimizations that contribute to up to 10X speedup.

We make the following contributions in this paper:

1. We describe a novel approach for optimizing complex Linear Algebra expressions by converting them to Relational Algebra, and prove the completeness of our rewrite rules (Sec. 2).

2. We present a search algorithm based on Equality Saturation that can explore a large search space while reusing memory (Sec. 3).

3. We conduct an empirical evaluation of the optimizer using several real-world machine learning tasks, and demonstrate it's superiority over an heuristics-driven optimizer in SystemML (Sec. 4).

Table 1: Representation of expressions in LA and RA.

	A			x		$A * x^T$			Ax	
LA	$\begin{bmatrix}0 & 5\\ 7 & 0\end{bmatrix}$			$\begin{bmatrix}3\\ 2\end{bmatrix}$		$\begin{bmatrix}0 & 10\\ 21 & 0\end{bmatrix}$			$\begin{bmatrix}10\\ 21\end{bmatrix}$	
RA	i	j	$\#$	j	$\#$	i	j	$\#$	i	$\#$
	1	2	5	1	3	1	2	10	1	10
	2	1	7	2	2	2	1	21	2	21

2. REPRESENTING THE SEARCH SPACE

2.1 Rules R_{LR}: from LA to RA and Back

In this section we describe our approach of optimizing LA expressions by converting them to RA. The rules converting from LA to RA and back are denoted R_{LR}.

To justify our approach, let us revisit our example loss function written in LA and attempt to optimize it using standard LA identities. Here we focus on algebraic rewrites and put aside concerns about the cost model. Using the usual identities on linear algebra expressions, one may attempt to rewrite the original expression as follows:

$$\begin{aligned}&sum((X-UV^T)^2)\\ =&sum((X-UV^T)*(X-UV^T))\\ =&sum(X^2-2X*UV^T+(UV^T)^2)\\ =&sum(X^2)-2sum(X*UV^T)+sum((UV^T)^2)\end{aligned}$$

At this point we are stuck trying to rewrite $sum(X*UV^T)$ (recall that $*$ is element-wise multiplication); it turns out to be equal to $sum(U^TXV)$, for any matrices X,U,V (and it is equal to the scalar U^TXV when U,V are column vectors), but this does not seem to follow from standard LA identities like associativity, commutativity, and distributivity. Similarly, we are stuck trying to rewrite $sum((UV^T)^2)$ to $sum(U^TU*V^TV)$. Current systems manually add syntactic rewrite rules, whenever such a special case is deemed frequent enough to justify extending the optimizer.

Instead, our approach is to expand out the LA expression element-wise. For example, assuming for simplicity that U,V are column vectors, we obtain

$$\begin{aligned}&sum((UV^T)^2)=\textstyle\sum_{i,j}(U_i\times V_j)\times(U_i\times V_j)\\ &=\textstyle\sum_{i,j}(U_i\times U_i)\times(V_j\times V_j)\\ &=(\textstyle\sum_i U_i\times U_i)\times(\sum_j V_j\times V_j)\\ &=U^TU\times V^TV\end{aligned}$$

The expressions using indices represent Relational Algebra expressions. More precisely, we interpret every vector, or matrix, or tensor, as a K-relation [13] over the reals. In other words we view X_{ij} as a tuple $X(i,j)$ whose "multiplicity" is the real value of that matrix element. We interpret point-wise multiply as natural join; addition as union; sum as aggregate; and matrix multiply as aggregate over a join[2]. Table 1 illustrates the correspondence between LA and RA. We treat each matrix entry A_{ij} as the multiplicity of tuple (i,j) in relation A under bag semantics. For example $A_{2,1}=7$, therefore the tuple $(2,1)$ has multiplicity of 7 in the corresponding relation. The relation schema stores the

[2]In the implementation, we use outer join for point-wise multiply and addition, where we multiply and add the matrix entries accordingly. In this paper we use join and union to simplify presentation.

Table 2: LA and RA operators. The type $M_{M,N}$ is a matrix of size $M \times N$; $[i,j]$ is a list of attribute names; $R_{i:M,j:N}$ is a relation with attributes i of size M and j of size N; S_1, S_2, S, and U are sets of attributes. `elemmult` and `elemplus` are broadcasting.

	name	type	syntax
LA	mmult	$M_{M,L} \times M_{L,N} \to M_{M,N}$	AB
	elemmult	$M_{M,N} \times M_{M,N} \to M_{M,N}$	$A * B$
	elemplus	$M_{M,N} \times M_{M,N} \to M_{M,N}$	$A + B$
	rowagg	$M_{M,N} \to M_{M,1}$	$sum_{row} A$
	colagg	$M_{M,N} \to M_{1,N}$	$sum_{col} A$
	agg	$M_{M,N} \to M_{1,1}$	$sum A$
	transpose	$M_{M,N} \to M_{N,M}$	A^T
conv.	bind	$M_{M,N} \times [i,j] \to R_{i:M,j:N}$	${}_{[i,j]}A$
	unbind	$R_{i:M,j:N} \times [i,j] \to M_{M,N}$	${}_{[-i,-j]}A$
RA	join	$R_{S_1} \times R_{S_2} \to R_{S_1 \cup S_2}$	$A \times B$
	union	$R_{S_1} \times R_{S_2} \to R_{S_1 \cup S_2}$	$A + B$
	agg	$R_S \times U \to R_{S \setminus U}$	$\sum_U A$

1. $A * B \to {}_{[-i,-j]}\left({}_{[i,j]}A \times {}_{[i,j]}B\right)$.
2. $A + B \to {}_{[-i,-j]}\left({}_{[i,j]}A + {}_{[i,j]}B\right)$.
3. $sum_{row} A \to [-i, \_] \sum_j {}_{[i,j]}A$. Similar for sum_{col}, sum.
4. $AB \to {}_{[-i,-k]} \sum_j \left({}_{[i,j]}A \times {}_{[j,k]}B\right)$.
5. $A^\mathsf{T} \to {}_{[-j,-i]}\,{}_{[i,j]}A$.
6. $A - B \to A + (-1) * B$

Figure 1: LA-to-RA Ruleset R_{LR}.

size of each dimension. $A * x^T$ denotes element-wise multiplication, where each element A_{ij} of the matrix is multiplied with the element x_j of the row-vector x^T. In RA it is naturally interpreted as the natural join $A(i,j) \bowtie x(j)$, which we write as $A(i,j) \times x(j)$. Similarly, Ax is the standard matrix-vector multiplication in LA, while in RA it becomes a query with a group by and aggregate, which we write as $\sum_j A(i,j) \times x(j)$. Our K-relations are more general than bags, since the entry of a matrix can be a real number, or a negative number; they correspond to K-relations over the semiring of reals $(\mathbb{R}, 0, 1, +, \times)$.

We now describe the general approach in SPORES. The definition of LA and RA are in Table 2. LA consists of seven operators, which are those supported in SystemML [3]. These operators all implement sum-product operations and take up the majority of run time in machine learning programs as we show in Section 4.2. We also support common operations like division and logarithm as we discuss in Section 3.3. RA consists of only three operators: $\times$ (natural join), $+$ (union), and $\sum$ (group-by aggregate). Difference is represented as $A - B = A + (-1)B$ (this is difference in $\mathbb{R}$; we do not support bag difference, i.e. difference in $\mathbb{N}$ like $3 - 5 = 0$, because there is no corresponding operation in LA), while selection can be encoded by multiplication with relations with 0/1 entries. We call an expression using these three RA operators an *RPlan*, for Relational Plan, and use the terms RPlan and RA/relational algebra interchangeably. Finally, there are two operators, *bind* and *unbind* for converting between matrices/vectors and K-relations.

The translation from LA to RA is achieved by a set of rules, denoted R_{LR}, and shown in Figure 1. The bind operator ${}_{[i,j]}$ converts a matrix to a relation by giving attributes i, j to its two dimensions; the unbind operator ${}_{[-i,-j]}$ converts a relation back to a matrix. For example, ${}_{[-j,-i]}\,{}_{[i,j]}A$ binds A's row indices to i and its column indices to j, then unbinds them in the opposite order, thereby transposing A.

1. $A \times (B + C) = A \times B + A \times C$
2. $\sum_i (A + B) = \sum_i A + \sum_i B$
3. If $i \notin A$, $A \times \sum_i B = \sum_i (A \times B)$ (else rename i)
4. $\sum_i \sum_j A = \sum_{i,j} A$
5. If $i \notin Attr(A)$, then $\sum_i A = A \times dim(i)$
6. $A + (B + C) = +(A, B, C)$ (assoc. & comm.)
7. $A \times (B \times C) = \times(A, B, C)$ (assoc. & comm.)

Figure 2: RA equality rules R_{EQ}.

SPORES translates a complex LA expression into RA by first applying the rules R_{LR} in Figure 1 to each LA operator, replacing it with an RA operator, preceded by *bind* and followed by *unbind*. Next, it eliminates consecutive unbind/bind operators, possibly renaming attributes, e.g. ${}_{[k,l]}\,{}_{[-i,-j]}A$ becomes $A[i \to k, j \to l]$, which indicates that the attributes i and j in A's schema should be renamed to k and l, by propagating the rename downward into A. As a result, the entire LA expression becomes an RA expression (RPlan), with *bind* on the leaves and *unbind* at the top.

2.2 Rules R_{EQ} : from RA to RA

The equational rules for RA consists of seven identities shown in Figure 2, and denoted by R_{EQ}. The seven rules are natural relational algebra identities, where $\times$ corresponds to natural join, $+$ to union (of relations with the same schema) and $\sum_i$ to group-by and aggregate. In rule 5, $i \notin Attr(A)$ means that i is not an attribute of A, and $dim(i)$ is the dimension of index i. For a very simple illustration of this rule, consider $\sum_i 5$. Here 5 is a constant, i.e. a relation of zero arity, with no attributes. The rule rewrites it to $5dim(i)$, where $dim(i)$ is a number representing the dimension of i.

2.3 Completeness of the Optimization Rules

As we have seen at the beginning of this section, when rewriting LA expressions using identities in linear algebra we may get stuck. Instead, by rewriting the expressions to RA, the seven identities in R_{EQ} are much more powerful, and can discover more rewrites. We prove here that this approach is *complete*, meaning that, if two LA expressions are semantically equivalent, then their equivalence can be proven by using rules R_{EQ}. The proof consists of two parts: (1) the rules R_{EQ} are sufficient to convert any RA expression e to its *normal form* (also called *canonical form*) $\mathcal{C}(e)$, and back, (2) two RA expressions e, e' are semantically equivalent iff they have isomorphic normal forms, $\mathcal{C}(e) \equiv \mathcal{C}(e')$.

We first give formal definitions for several important constructs. First, we interpret a relation as a function from *tuples* to a *semiring*. For simplicity we assume all attributes have the same domain $\mathbf{D}$.

Definition 2.1. (Relations) *Fix a semiring* $\mathbf{S}$. *An* **S-relation** *is a function* $A : \mathbf{D}^a \to \mathbf{S}$ *where* a *is the* arity *of the relation.*

When $\mathbf{S} = \mathbb{N}$, then an $\mathbb{N}$-relation is a standard relation under bag semantics. When the domain is $\mathbf{D} = [n]$ for some natural number n and $\mathbf{S} = \mathbb{R}$, then an $\mathbb{R}$-relation is a tensor over the reals. Next we define expressions over operators from Table 2.

Definition 2.2. (Expressions) *An expression in RA is either 1. an* **atom** r *of the form* $R(x_1, ..., x_a)$ *where* R *is*

a relation name and $(x_1, ..., x_a)$ *a tuple of variables and/or constants, or 2. a* natural join *of two expressions* $e_1 \times e_2$*, or 3. a* union *of two expressions,* $e_1 + e_2$*, or 4. an* aggregate*,* $\sum_x e$*.*

Because the order of consecutive aggregates does not matter, we write $\sum_{\{x_1,\ldots,x_n\}} e$ for $\sum_{x_1} \cdots \sum_{x_n} e$. Given an expression e, we say that a variable x is ***bound*** in e, if e contains an aggregate of the form $\sum_x e'$; otherwise it is ***free***. We write $vars(e)$ for the set of variables in e, $bv(t)$ for the set of bound variables, and $fv(t)$ for the set of free variables. We interpret every expression e as a lambda expression $\lambda fv(e).e$ where the parameters $fv(e)$ may follow a given order, e.g. from an unbind operator if e was converted from LA[3]. In the body of the lambda expression, any atom $R(x_1, \ldots, x_a)$ evaluates to some $s \in \mathbf{S}$, and $+, \times$ and $\sum$ compute over $\mathbf{S}$. We say two RA expressions are equivalent iff they evaluate to the same result given any same inputs:

DEFINITION 2.3. (Equivalence of Expressions) *Fix expressions* e_1, e_2 *over the relation symbols* $R_1, \ldots, R_n$*. We say that* e_1, e_2 *are* **equivalent** *over the semiring* $\mathbf{S}$ *and the domain* $\mathbf{D}$ *if they have the same free variables, and for all interpretations* $\boldsymbol{I} = (R_1^I, \ldots, R_n^I)$ *where* $R_j^I : \mathbf{D}^{a_j} \to \mathbf{S}$ *for* $j = 1, \ldots, n$ *the two expressions return the same answer,* $e_1(\boldsymbol{I}) = e_2(\boldsymbol{I})$*. We write* $e_1 =_{\mathbf{S},\mathbf{D}} e_2$ *to mean* e_1 *and* e_2 *are equivalent. When* $e_1 =_{\mathbf{S},\mathbf{D}} e_2$ *for all domains* $\mathbf{D}$*, then we abbreviate* $e_1 =_{\mathbf{S}} e_2$*; when this holds for all semirings* $\mathbf{S}$*, then we write* $e_1 = e_2$*.*

Now we give names for special forms of expressions at each level of the normal form.

DEFINITION 2.4. (R-monomials, Terms, R-polynomials) *An* **R-monomial***,* m*, is a product of any number of atoms. A* **term***,* t*, is a summation expression of the form* $\sum_{\boldsymbol{x}} m$*, where* $\boldsymbol{x}$ *is a set of variables and* m *is an R-monomial. An* **R-polynomial***,* e*, is an expression of the form* $c_0 + c_1 t_1 + \cdots + c_n t_n$ *where* $c_0, \ldots, c_n$ *are constants in the semiring* $\mathbf{S}$ *and* $t_1, \ldots, t_n$ *are terms. In summary:*

$$m := r_1 \times \cdots \times r_n \qquad \textit{R-monomials} \qquad (1)$$

$$t := \textstyle\sum_{\boldsymbol{x}} m \qquad \textit{terms} \qquad (2)$$

$$e := c_0 + c_1 t_1 + \cdots + c_n t_n \quad \textit{R-polynomials} \qquad (3)$$

We identify the R-monomial m *with a bag of atoms denoted by* $bag(m) \stackrel{\text{def}}{=} \{r_1, \ldots, r_n\}$*.*

EXAMPLE 1. *An R-polynomial over* $\mathbb{N}$*-relations is a union of conjunctive queries precisely. For example, the polynomial* $\sum_j A(i,j) \times A(i,j) \times B(j,k) \times B(j,k) + \sum_\ell A(i,\ell) \times C(\ell,k)$ *is precisely the UCQ* $Q(i,k) \equiv \exists j(A(i,j) \wedge A(i,j) \wedge B(j,k) \wedge B(j,k)) \vee \exists \ell(A(i,l) \wedge C(l,k))$ *under bag semantics. Notice that the monomial* $A(i,j) \times A(i,j) \times B(j,k) \times B(j,k)$ *is the same as* $A(i,j) \times B(j,k) \times A(i,j) \times B(j,k)$*, and we view it as the bag* $\{A(i,j), A(i,j), B(j,k), B(j,k)\}$*, and also abbreviate it as* $A^2(i,j) \times B^2(j,k)$*.*

Before we formally define our canonical form, we need to define two syntactical relationships between our expressions, namely *homomorphism* and *isomorphism*. Fix terms $t = \sum_{\boldsymbol{x}} m$ and $t' = \sum_{\boldsymbol{x}'} m'$, and let $f : \boldsymbol{x} \to \boldsymbol{x}'$ be any function.

[3]A bind operator $[i,j]$ converts a matrix A to an atom $A(i,j)$.

Let $r \in bag(m)$ be an atom of m. We write $f(r)$ for the result of applying f to all variables of r. We write $f(bag(m))$ for the bag obtained by applying f to each atom $r \in bag(m)$.

DEFINITION 2.5. (Homomorphism) *Fix two terms* t, t'*. A* **homomorphism***,* $f : t \to t'$*, is a function* $f : \boldsymbol{x} \to \boldsymbol{x}'$ *such that* $f(bag(m)) = bag(m')$*.*

EXAMPLE 2. *Let* $t_1 = \sum_{vws} A(i,v) \times B(v,w) \times A(i,s)$ $t_2 = \sum_{jk} A^2(i,j) \times B(j,k)$ *and* $t_3 = \sum_{jk} A(i,j) \times B(j,k)$*, and consider the function* $f : \{v,w,s\} \to \{j,k\}$ *defined by* $v \mapsto j, w \mapsto k, s \mapsto j$*. Then this is a homomorphism* $f : t_1 \to t_2$*. On the other hand* f *is* not *a homomorphism from* $t_1 \to t_3$*, because* $f(bag(t_1)) = \{A(i,j), B(j,k), A(i,j)\}$ *contains the atom* $A(i,j)$ *twice, while* $bag(t_3)$ *contains it only once.*

Notice that t_1, t_2 must have exactly the same free variables. By convention, we extend f to be the identity on the free variables. The following facts are easily verified:

FACT 1. *Every homomorphism* $f : t_1 \to t_2$ *is a* **surjective** *function* $vars(t_1) \to vars(t_2)$*.*

FACT 2. *Homomorphisms are closed under composition.*

A stronger correspondence between terms is an isomorphism:

DEFINITION 2.6 (TERM ISOMORPHISM). *Fix two terms* t, t'*. An* **isomorphism** *is a homomorphism* $f : t \to t'$ *that is a bijection from* $vars(t)$ *to* $vars(t')$*. If an isomorphism exists, then we say that* t, t' *are* **isomorphic** *and write* $t \equiv t'$*.*

LEMMA 2.1. *Fix two terms* t_1 *and* t_2*. If there exists homomorphisms* $f : t_1 \to t_2$ *and* $g : t_2 \to t_1$ *then the terms are isomorphic,* $t_1 \equiv t_2$*. More generally, any cycle of homomorphisms* $t_1 \to t_2 \to t_3 \to \ldots \to t_n \to t_1$ *implies that all terms are isomorphic.*

PROOF. The composition $g \circ f$ is a homomorphism $t_1 \to t_1$ which, by Fact 1, is a surjective function $vars(t_1) \to vars(t_1)$; since $vars(t_1)$ is a finite set, it follows that $g \circ f$ is a bijection, hence so are f and g. □

We are now ready to formally define the canonical form for RA expressions:

DEFINITION 2.7. (Canonical Form) *An RPlan expression (as defined in Table 2) is* **canonical** *if it is a R-polynomial (Definition 2.4) containing no isomorphic terms.*

We can canonicalize any expression by pulling $+$ to the top and pushing $\times$ to the bottom, while combining isomorphic terms $c_1 t + c_2 t$ into $(c_1 + c_2)t$:

LEMMA 2.2. *For every RPlan expression there is a canonical expression equivalent to it.*

PROOF. The proof is a standard application of the rewrite rules R_{EQ} in Figure 2. □

We can identify canonical expressions syntactically using term isomorphism:

DEFINITION 2.8. (Isomorphic Canonical Expressions) *Fix two R-polynomials* $e = c_0 + c_1 t_1 + \cdots + c_n t_n$ *and* $e' = c'_0 + c'_1 t'_1 + \cdots + c'_m t'_m$*. We say that* e *and* e' *are* **isomorphic** *if* $m = n$*,* $c_0 = c'_0$*, and there exists a permutation* $\sigma : [n] \to [n]$ *such that* $\forall i \in [n]$*,* $c_i = c'_{\sigma(i)}$*, and* $t_i \equiv t'_{\sigma(i)}$*.*

In other words, e, e' are isomorphic if they are essentially the same expression, up to commutativity of $+$ and up to replacing terms t_i with isomorphic terms $t'_{\sigma(i)}$. In particular, isomorphic expressions have same free variables. Our ultimate goal is to identify canonical form isomorphism with equivalence. That is, two canonical expressions are equivalent iff they are isomorphic. For any R-polynomial $e = c_0 + c_1t_1 + \cdots c_nt_n$, we denote by $|vars(e)| \stackrel{\text{def}}{=} \max_i(|vars(t_i)|)$. Our main result is the following:

THEOREM 2.3. *(**Isomorphism Captures Equivalence**) Let e_1, e_2 be two canonical expressions. Then the following conditions are equivalent:*

1. $e_1 \equiv e_2$
2. $e_1 = e_2$ 3. $e_1 =_{\mathbb{C}} e_2$ 4. $e_1 =_{\mathbb{R}} e_2$ 5. $e_1 =_{\mathbb{N}} e_2$
6. $e_1 =_{\mathbb{N},\mathbf{D}} e_2$, *for some finite domain* $\mathbf{D}$ *s.t.* $|\mathbf{D}| \geq \max(|vars(e_1)|, |vars(e_2)|)$.

The implications (1) $\Rightarrow$ (2) $\Rightarrow \cdots \Rightarrow$ (6) are straightforward. We prove below that (6) $\Rightarrow$ (1). In other words, we prove that, if e_1, e_2 are equivalent over the semiring of natural numbers $\mathbb{N}$ and over some domain "large enough", then their canonical forms must be isomorpic. Requiring $|\mathbf{D}|$ to be large enough is necessary, because otherwise two non-isomorphic expressions may be equivalent. For example, if we restrict the relations X, Y to be matrices of dimensions 1×1, then the expressions $\sum_{i,j} X(i,j) \times Y(i,j)$ and $\sum_{i,j} X(i,j) \times Y(j,i)$ have the same semantics, but different canonical form. For another example, if x, y, z are vectors of length 2, these two expressions are equivalent: $\sum_{i,j,k} x(i) \times y(j) \times z(k) + 2\sum_i x(i) \times y(i) \times z(i)$ and $\sum_{i,j} x(i) \times y(i) \times z(j) + \sum_{i,j} x(i) \times y(j) \times z(i) + \sum_{i,j} x(j) \times y(i) \times z(i)$, although they are not equivalent when x, y, z are vectors of length ≥ 3.

PROOF. We prove (6) $\Rightarrow$ (1). Assume that $e_1(\boldsymbol{I}) = e_2(\boldsymbol{I})$ for all interpretations $\boldsymbol{I}$ over the domain $\mathbf{D}$. We start by observing that the constant terms in e_1 and e_2 must be equal, i.e. if $e_1 = c_0 + \ldots, e_2 = c'_0 + \ldots$ then $c_0 = c'_0$. This follows by choosing $\boldsymbol{I}$ to consists of empty relations, in which case $e_1(\boldsymbol{I}) = c_0$ and $e_2(\boldsymbol{I}) = c'_0$, proving $c_0 = c'_0$. Thus, we can cancel the constant terms and assume w.l.o.g. that e_1, e_2 have no constant terms. Next, we show that we can assume w.l.o.g. that e_1 and e_2 have no free variables. Otherwise, denote by $\boldsymbol{x}$ the free variables in e_1 and e_2 (they must be the same in order for e_1, e_2 to be equivalent), and define $e'_1 = \sum_{\boldsymbol{x}} e_2$ and $e'_2 = \sum_{\boldsymbol{x}} e_2$. It is easy to check that e'_1, e'_2 are also equivalent and, if we prove that they are isomorphic, then so are e_1, e_2. Thus, we will assume w.l.o.g. that e_1, e_2 have no free variables. Suppose that e_1 and e_2 contain two terms that are isomorphic: that is, e_1 contains c_it_i, e_2 contains $c'_jt'_j$, and $t_i \equiv t'_j$. In particular, $t_i = t'_j$ i.e. they are also equivalent. Assuming $c_i \geq c'_j$, we subtract $c'_jt'_j$ from both e_1 and e_2; now e_1 contains $(c_i - c'_j)t_i$, while e_2 no longer contains t'_j. By repeating this process, we remove any pair of isomorphic terms from e_1, e_2. If e_1, e_2 were isomorphic, then after this process we remove all terms and both e_1, e_2 becomes 0. Suppose by contradiction that this is not the case, thus $e_1 = c_1t_1 + c_2t_2 + \ldots c_mt_m$, $e_2 = c'_1t'_1 + \cdots + c'_nt'_n$, and, denoting $T \stackrel{\text{def}}{=} \{t_1, \ldots, t_m, t'_1, \ldots, t'_n\}$ the set of terms in both expressions, no two terms in T are isomorphic. Assuming $m > 0$ or $n > 0$, we prove that $e_1 =_{\mathbb{N},\mathbf{D}} e_2$ is a contradiction.

Let us denote by $t < t'$ when there exists a homomorphism $t \to t'$. Then $<$ defines a partial order on T, i.e. there is no $<$-cycle, otherwise Lemma 2.1 would imply that some terms are isomorphic. Let $t_1 \in T$ be any minimal element under $<$, in other words there is no $t' \in T$ s.t. $t' < t_1$. Assume w.l.o.g. that t_1 is a term in e_1. We will construct an instance $\boldsymbol{I}$ that is "canonical" for t_1, and prove that $e_1(\boldsymbol{I}) \neq e_2(\boldsymbol{I})$. Assume $t_1 = \sum_{\boldsymbol{x}} r_1^{k_1} \times \cdots r_m^{k_m}$, where $r_1, \ldots, r_k$ are distinct atoms, and $r_i^{k_i} \stackrel{\text{def}}{=} r_i \times \cdots \times r_i$ (k_i times). Let $n = |\boldsymbol{x}|$, and recall that, by assumption, $|\mathbf{D}| \geq n$. Choose any injective function $\theta : \boldsymbol{x} \to \mathbf{D}$; to reduce clutter we assume w.l.o.g. that $\mathbf{D} = \{1, 2, \ldots, n\}$ and $\theta(x_1) = 1, \ldots, \theta(x_n) = n$. Let $u_1, \ldots, u_m$ be m variables over $\mathbb{N}$, one for each distinct atom in t_1. We define the canonical $\boldsymbol{I}$ as follows. For each relational symbol R, and for any atom $r_i = R(x_{j_1}, \ldots, x_{j_a})$ that uses the symbol R, we define $R^I(j_1, \ldots, j_a) \stackrel{\text{def}}{=} u_i$; for all other tuples in $\mathbf{D}^a$ we define $R(\ldots) = 0$. This completes the definition of $\boldsymbol{I}$. We make two claims. First, $t_1(\boldsymbol{I})$ is a multivariate polynomial containing the monomial $cu_1^{k_1} \cdots u_m^{k_m}$. To see this, write $t = \sum_{\boldsymbol{y}} r'_1 \times \cdots \times r'_q$, and observe that $t(\boldsymbol{I}) = \sum_{\tau:\boldsymbol{y}\to\mathbf{D}} \tau(r'_1) \times \cdots \times \tau(r'_q)$. When $\tau = \theta$ the R-monomial $\tau(r'_1) \times \cdots \times \tau(r'_q) = \theta(r'_1) \times \cdots \times \theta(r'_q)$ is precisely $u_1^{k_1} \cdots u_m^{k_m}$. Second, we claim that, for any other term $t \in T$, its value $t(\boldsymbol{I})$ on the canonical instance is some multivariate polynomial in $u_1, \ldots, u_m$ that does *not* contain the monomial $u_1^{k_1} \cdots u_m^{k_m}$. Indeed, suppose it contained this monomial: then we prove that there exists a homomorphism $t \to t_1$, contradicting the assumption that t_1 is minimal. To see this, consider again $t(\boldsymbol{I}) = \sum_{\tau:\boldsymbol{y}\to\mathbf{D}} \tau(r'_1) \times \cdots \times \tau(r'_q)$. If this expression includes the monomial $u_1^{k_1} \cdots u_m^{k_m}$, then for some function $\tau : \boldsymbol{y} \to \mathbf{D}$, the bag $\{\theta'(r'_1), \ldots, \theta'(r'_q)\}$ must contain precisely the atom $\theta(r_1)$ k_1-times, the atom $\theta(r_2)$ k_2-times, etc. But that means that τ is a homomorphism $t \to t_1$ (since $\mathbf{D}$ and $vars(t_1)$ are isomorphic via θ), contradicting our assumption that t_1 is minimal.

Thus, we have established that both $e_1(\boldsymbol{I})$ and $e_2(\boldsymbol{I})$ are multivariate polynomials in $u_1, \ldots, u_m$, but the first expression contains the monomial $u_1^{k_1} \cdots u_m^{k_m}$ while the second does not contain it. Since $e_1 = e_2$, these two polynomials must have the same values for all choices of natural numbers $u_1, \ldots, u_m \in \mathbb{N}$. It is well known from classical algebra that, in this case, the two polynomials are identical, which is a contradiction.

For a simple illustration, assume $e_1 = t_1$ and $e_2 = t_2$, where $t_1 = \sum_{x,y,z} R(x,y) \times R(y,z) \times R(z,x)$ and $t_2 = \sum_i R(i,i)^3$. They are not isomorphic, and our proof essentially constructs an instance $\boldsymbol{I}$ on which their answers differ. Since we have a homomorphism $t_1 \to t_2$ but not vice versa, the instance is the canonical instance for t_1, i.e. $R^I(1,2) \stackrel{\text{def}}{=} u_1$, $R^I(2,3) \stackrel{\text{def}}{=} u_2$, $R^I(3,1) \stackrel{\text{def}}{=} u_3$, and all the other entries are 0. Then it is easy to verify that $t_1(\boldsymbol{I}) = 3u_1u_2u_3$ (there are three isomorphisms $t_1 \to t_1$), while $t_2(\boldsymbol{I}) = 0$. Notice that the canonical instance for t_2, $R^I(1,1) \stackrel{\text{def}}{=} u_1$ and all other entries are 0, does not make the two expressions different: $t_1(\boldsymbol{I}) = t_2(\boldsymbol{I}) = u_1^3$. $\square$

We are now ready to establish the completeness of RA equalities, by showing any equivalent LA expressions can be rewritten to each other through the translation rules R_{LR} and the canonicalization rules R_{EQ}:

THEOREM 2.4. *(**Completeness of** R_{EQ}) Two LA expressions are semantically equivalent if and only if their relational form can be rewritten to each other by following R_{EQ}.*

In the following $R_{LR}(e)$ translates LA expression e into RA and $\mathcal{C}(e)$ returns the normal form of e.

PROOF. Translating e_1 and e_2 to RA preserves semantics under R_{LR}. By Lemma 2.2, normalizing $R_{LR}(e_1)$ and $R_{LR}(e_2)$ preserves semantics. By Theorem 2.3,

$$R_{LR}(e_1) =_{\mathbb{R}} R_{LR}(e_2) \iff \mathcal{C}(R_{LR}(e_1)) \equiv \mathcal{C}(R_{LR}(e_2))$$

Since every rule in R_{EQ} is reversible, the right-hand-side is true iff $R_{LR}(e_1)$ and $R_{LR}(e_2)$ can be rewritten to each other via R_{EQ}. □

3. EXPLORING THE SEARCH SPACE

With a complete representation of the search space by relational algebra, our next step is to explore this space and find the optimal expression in it. Traditional optimizing compilers commonly resort to heuristics to select from available rewrites to apply. SystemML implements a number of heuristics for its algebraic rewrite rules, and we discuss a few categories of them here.

COMPETING OR CONFLICTING REWRITES The same expression may be eligible for more than one rewrites. For example, $sum(AB)$ rewrites to $sum(sum_{col}(A)^T * sum_{row}(B))$, but when both A and B are vectors the expression can also be rewritten to a single dot product. SystemML then implements heuristics to only perform the first rewrite when the expression is not a dot product. In the worst case, a set of rules interacting with each other may create a quadratic number of such conflicts, complicating the codebase.

ORDER OF REWRITES Some rewrite should be applied after others to be effective. For example, X/y could be rewritten to $X * 1/y$ which may be more efficient, since SystemML provides efficient implementation for sparse multiplication but not for division. This rewrite should occur before constant folding; otherwise it may create spurious expressions like $X/(1/y) \rightarrow X * (1/(1/y))$, and without constant folding the double division will persist. However, a rewrite like $1/(1 + exp(-X)) \rightarrow sigmoid(X)$ should come after constant folding, in order to cover expressions like $(3-2)/(1+exp(-X))$. Since SystemML requires all rewrites to happen in one phase and constant folding another, it has to leave out[4] rewrites like $X/y \rightarrow X * 1/y$.

DEPENDENCY ON INPUT/PROGRAM PROPERTIES Our example optimization from $sum((X - UV^T)^2)$ to $sum(X^2) - 2U^TXV + U^TU * V^TV$ improves performance only if X is sparse. Otherwise, computing X^2 and $X * UV^T$ would both create dense intermediates. Similarly, some rewrites depend on program properties like common subexpressions. Usually, these rewrites only apply when the matched expression shares no CSE with others in order to leverage common subexpression elimination. Testing input and program properties like this becomes boilerplate code, making implementation tedious and adds burden to maintenance.

COMPOSING REWRITES Even more relevant to us is the problem of composing larger rewrites out of smaller ones. Our equality rules R_{EQ} are very fine-grained, and any rule is unlikely to improve performance on its own. Our example optimization from $sum((X - UV^T)^2)$ to $sum(X^2) - 2U^TXV + U^TU * V^TV$ takes around 10 applications of R_{EQ} rules. If an optimizer applies rewrites one by one, it is then very difficult, if not impossible, for it to discover the correct sequence of rewrites that compose together and lead to the best performance.

Stepping back, the challenge of orchestrating rewrites is known as the *phase-ordering problem* in compiler optimization. Tate et al. [40] proposed a solution dubbed *equality saturation* which we adapt and extend in SPORES.

[4]Another reason to leave out this rewrite is that $X * 1/y$ rounds twice, whereas X/y only rounds once.

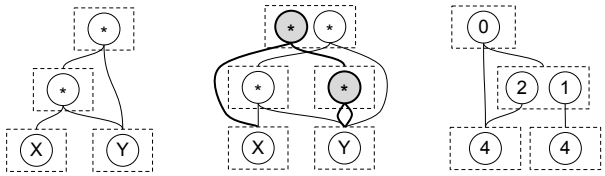

Figure 3: Left: E-Graph representing $(X \times Y) \times Y$, and the graph after applying associativity to the root (middle). New nodes are in gray. Each dashed box is an E-Class. Right: the CSE problem. Each node shows its cost.

3.1 Equality Saturation

Equality saturation optimizes an expression in two steps:

Saturation: given the input expression, the optimizer enumerates equivalent expressions and collects them into a compact representation called the E-Graph [32].

Extraction: given a cost function, the optimizer selects the optimal expression from the E-Graph. An expression is represented by a subgraph of the E-Graph, and the optimizer uses a constraint solver to find the subgraph equivalent to the input that is optimal according to the cost function.

The E-Graph Data Structure

An E-Graph represents sets of equivalent expressions. A node in the graph is called an E-Class, which contains the root operators of a set of equivalent expressions. The edges are similar to the edges in an abstract syntax tree; but instead of pointing from an operator directly to a child, each edge points from an operator to an E-Class of expressions. For example, in Figure 3 the top class in the middle represents the set of equivalent expressions $\{(X \times Y) \times Y, X \times (Y \times Y)\}$. Note that the class represents two expressions, each with 2 appearances of Y and one appearance of X, whereas each variable only appears once in the E-Graph. This is because the E-Graph makes sure its expressions share all possible common subexpressions. As the size of the graph grows, this compression becomes more and more notable; in some cases a graph can represent a number of expressions exponential to its size [40]. We take advantage of this compression in SPORES to efficiently cover vast portions of the search space. If saturation, as described below, carries out to convergence, the E-Graph represents the search space exhaustively.

An E-Graph can also be seen as an AND-OR DAG over expressions. Each E-Class is an OR node whose children are equivalent expressions from which the optimizer chooses from. Each operator is an AND node whose children must all be picked if the operator itself is picked. In this paper we favor the terms E-Graph and E-Class to emphasize each OR node is an equivalence class.

```
1  def saturate(egraph, equations):
2    for eq in equations:
3      matches = egraph.match(eq.lhs)
4      for eclass in matches:
5        ec = egraph.add(eq.rhs)
6        egraph.merge(eclass, c)
7
8  def add(expr):
9    ID = egraph.find(expr)
10   if ID != NULL:
11     return ID
12   else:
13     cids = expr.children.map(add)
14     ID = egraph.insert(expr.op, cids)
15     return ID
```

Figure 4: Equality saturation pseudocode.

Saturating the E-Graph

At the beginning of the optimization process, the optimizer instantiates the graph by inserting the nodes in the syntax tree of the input expression one by one in post order. For example, for input $(X \times Y) \times Y$, we construct the left graph in Figure 3 bottom-up. By inserting in post order, we readily exploit existing common subexpressions in the input. Once the entire input expression is inserted, the optimizer starts to extend the graph with new expressions equivalent to the input. It considers a list of equations, and matches either side of the equation to subgraphs of the E-Graph. If an equation matches, the optimizer then inserts the expression on the other side of the equation to the graph. For example, applying the associativity rule extends the left graph in Figure 3 with $X \times (Y \times Y)$, resulting in the right graph. Figure 4 shows the pseudo code for this process. While inserting new expressions, the optimizer checks if any subexpression of the new expression is already in the graph. If so, it reuses the existing node, thereby exploiting all possible common-subexpressions to keep the E-Graph compact. In Figure 3, only two $\times$ are added since the variables X and Y are already in the graph. Once the entire new expression has been added, the optimizer then merges the newly created E-Class at its root with the E-Class containing the matched expression, asserting them equal. Importantly, the optimizer also propagates the congruent closure of this new equality. For example, when $A+A$ is merged with $2\times A$, the optimizer also merges $(A + A)^2$ with $(2 \times A)^2$. Figure 4 shows the pseudo code for adding an expression to E-Graph. This process of match-and-insert is repeated until the graph stops changing, or reaching a user-specified bound on the number of saturation iterations. If this process does converge, that means no rule can add new expressions to the graph any more. If the set of rules are complete, as is our R_{EQ}, convergence of saturation implies the resulting E-Graph represents the transitive closure of the equality rules applied to the initial expression. In other words, it contains *all* expressions equivalent to the input under the equality rules.

The outer loop that matches equations to the graph can be implemented by a more efficient algorithm like the Rete algorithm [10] when the number of equations is large. However, we did not find matching to be expensive and simply match by traversing the graph. Our implementation uses the E-Graph data structure from the `egg` [44] library.

Dealing with Expansive Rules

While in theory equality saturation will converge with well-constructed rewrite rules, in practice the E-Graph may explode for certain inputs under certain rules. For example, a long chain of multiplication can be rewritten to an exponential number of permutations under associativity and commutativity (AC rules). If we apply AC rules everywhere applicable in each iteration, the graph would soon use up available memory. We call this application strategy the *depth-first* strategy because it eagerly applies expansive rules like AC. AC rules by themselves rarely affect performance [23], and SystemML also provides the fused `mmchain` operator that efficiently computes multiplication chains, so permuting a chain is likely futile. In practice, AC rules are useful because they can enable other rewrites. Suppose we have a rule $R_{factor} : A \times X + B \times X \rightarrow (A + B) \times X$ and an expression $U \times Y + Y \times V$. Applying commutativity to $Y \times V$ would then transform the expression to be eligible for R_{factor}. With this insight, we change each saturation iteration to sample a limited number of matches to apply per rule, instead of applying all matches. This amounts to adding `matches = sample(matches, limit)` between line 3 and line 4 in Figure 4. Sampling encourages each rule to be considered equally often and prevents any single rule from exploding the graph. This helps ensure good exploration of the search space when exhaustive search is impractical. But when it is possible for saturation to converge and be exhaustive, it still converges with high probability when we sample matches. Our experiments in Section 4.3 show sampling always preserve convergence in practice.

Extracting the Optimal Plan

A greedy strategy to extract the best plan from the saturated E-Graph is to traverse the graph bottom-up, picking the best plan at each level. This assumes the best plan for any expression also contains the best plan for any of its subexpressions. However, the presence of common subexpressions breaks this assumption. In the right-most graph in Figure 3 each operator node is annotated with its cost. Between the nodes with costs 1 and 2, a greedy strategy would choose 1, which incurs total cost of $1 + 4 = 5$. The greedy strategy then needs to pick the root node with cost 0 and the other node with cost 4, incurring a total cost of 9. However, the optimal strategy is to pick the nodes with 0, 2 and share the same node with cost 4, incurring a total cost of 6.

We handle the complexity of the search problem with a constraint solver. We assign a variable to each operator and each E-Class, then construct constraints over the variables for the solver to select operators that make up a valid expression. The solver will then optimize a cost function defined over the variables; the solution then corresponds to the optimal expression equivalent to the input. We implement both the greedy strategy and the solver-based strategy and compare them in Section 4.3.

Constraint Solving and Cost Function

We encode the problem of extracting the cheapest plan from the E-Graph with integer linear programming (ILP). Figure 5 shows this encoding. For each operator in the graph, we generate a boolean variable B_{op}; for each E-Class we generate a variable B_c. For the root class, we use the variable B_r. Constraint $F(op)$ states that if the solver selects an operator, it must also select all its children; constraint $G(c)$ states that if the solver selects an E-Class, it must select at least one of its members. Finally, we assert B_r must be selected, which constrains the extracted expression

$$Constraints \equiv B_r \wedge \bigwedge_{op} F(op) \wedge \bigwedge_{c} G(c)$$

$$F(op) \equiv B_{op} \rightarrow \bigwedge_{c \in op.children} B_c$$

$$G(c) \equiv B_c \rightarrow \bigvee_{op \in c.nodes} B_{op}$$

$$\textbf{minimize} \sum_{op} B_{op} \cdot C_{op} \textbf{ s.t. } Constraints$$

Figure 5: ILP constraint and objective for extraction.

$$\mathbf{S}[X \times Y] = min(\mathbf{S}[X], \mathbf{S}[Y])$$
$$\mathbf{S}[X + Y] = min(1, \mathbf{S}[X] + \mathbf{S}[Y])$$
$$\mathbf{S}[\sum_{i} X] = min(1, |i| \cdot \mathbf{S}[X])$$

Figure 6: Sparsity estimation. We define $sparsity = nnz/size$, i.e. a 0 matrix has sparsity 0.0[5]. $|i|$ is the size of the aggregated dimension.

to be in the same E-Class as the unoptimized expression. These three constraints together ensure the selected nodes form a valid expression equivalent to the unoptimized input. Satisfying these constraints, the solver now minimizes the cost function given by the total cost of the selected operators. Because each B_{op} represents an operator node in the E-Graph which can be shared by multiple parents, this encoding only assigns the cost once for every shared common subexpression. In our implementation, we use Gurobi [14] to solve the ILP problem.

Each operation usually has cost proportional to the output size in terms of memory allocation and computation. Since the size of a matrix is proportional to its the number of non-zeroes (nnz), we use SystemML's estimate of nnz as the cost for each operation. Under our relational interpretation, this corresponds to the cardinality of relational queries. We use the simple estimation scheme in Figure 6, which we find to work well. We rely on SystemML's estimation for non-sum-product operators. Future work can hinge on the vast literature on sparsity and cardinality estimation to improve the cost model.

3.2 Schema and Sparsity as Class Invariant

In the rules R_{EQ} used by the saturation process, Rule (3) If $i \notin A$, $A \times \sum_i B = \sum_i (A \times B)$ contains a condition on attribute i which may be deeply nested in the expression. This means the optimizer cannot find a match with a simple pattern match. Fortunately, all expressions in the same class must contain the same set of free attributes (attributes not bound by aggregates). In other words, the set of free variables is invariant under equality. This corresponds precisely to the schema of a database - equivalent queries must share the same schema. We therefore annotate each class with its schema, and also enable each equation to match on the schema.

In general, we find class invariants to be a powerful construct for programming with E-Graphs. For each class we track as class invariant if there is a constant scalar in the class. As soon as all the children of an operator are found to contain constants, we can fold the operator with the constant it computes. This seamlessly integrates constant folding with the rest of the rewrites. We also treat sparsity as a class invariant and track it throughout equality saturation. Because our sparsity estimation is conservative, equal expressions that use different operators may have different estimates. But as soon as we identify them as equal, we can merge their sparsity estimates by picking the tighter one, thereby improving our cost function. Finally, we also take advantage of the schema invariant during constraint generation. Because we are only interested in RA expressions that can be translated to LA, we only generate symbolic variables for classes that have no more than two attributes in their schema. This prunes away a large number of invalid candidates and helps the solver avoid wasting time on them. We implement class invariants using `egg`'s Metadata API.

[5]Some may find this definition counter-intuitive; we define it so to be consistent with SystemML.

3.3 Translation, Fusion and Custom Functions

Since equality saturation can rewrite any expression given a set of equations, we can directly perform the translation between LA and RA within saturation, simply by adding the translation rules R_{LR} from Figure 1. Furthermore, saturation has flexible support for custom functions. The simplest option is to treat a custom functions as a black box, so saturation can still optimize below and above them. With a little more effort, we have the option to extend our equations R_{EQ} to reason about custom functions, removing the optimization barrier. We take this option for common operators that are not part of the core RA semantics, e.g. square, minus and divide. In the best scenario, if the custom function can be modeled by a combination of basic operators, we can add a rule equating the two, and retain both versions in the same graph for consideration. In fact, this last option enables us to encode fused operators and seamlessly integrate fusion with other rewrite rules. As a result, the compiler no longer need to struggle with ordering fusion and rewrites, because saturation simultaneously considers all possible ordering. We note that although supporting custom functions require additional rules in SPORES, these rules are all identities, and they are much simpler than the heuristics rules in SystemML which need to specify when to fire a rule and how each rule interacts with another. Finally, although SystemML does not directly expose "physical" operators, e.g. different matrix multiplication algorithms, SPORES readily supports optimization of physical plans. For example, we could use two distinct operators for two matrix multiplication algorithms, and both would always appear in the same E-Class. Both operators would share the same child E-Classes, therefore the additional operator only adds one node for every class that contains a matrix multiply.

3.4 Saturation v.s. Heuristics

Using equality saturation, SPORES elegantly remedies the drawbacks of heuristics mentioned in the beginning of section 3. First, when two or more conflicting rewrites apply, they would be added to the same E-Class, and the extraction step will pick the more effective one based on the global cost estimate. Second, there is no need to carefully order rewrites, because saturation simultaneously considers all possible orders. For example, when rules R_1 and R_2 can rewrite expression e to either $R_1(R_2(e))$ or $R_2(R_1(e))$, one iteration of saturation would add $R_1(e)$ and $R_2(e)$ to the graph, and another iteration would add both $R_1(R_2(e))$ and $R_2(R_1(e))$ to the same E-Class. Third, rules do not need to reason about their dependency on input or program proper-

ties, because extraction uses a global cost model that holistically incorporates factors like input sparsity and common subexpressions. Finally, every rule application in saturation applies one step of rewrite on top of those already applied, naturally composing complex rewrites out of simple ones.

3.5 Integration within SystemML

We integrate SPORES into SystemML to leverage its compiler infrastructure. SPORES plugs into the algebraic rewrite pass in SystemML; it takes in a DAG of linear algebra operations, and outputs the optimized DAG. Within SPORES, it first translates the LA DAG into relational algebra, performs equality saturation, and finally translates the optimal expression back into LA. We obtain matrix characteristics such as dimensions and sparsity estimation from SystemML. Since we did not focus our efforts in supporting various operators and data types unrelated to linear algebra computation (e.g. string manipulation), we only invoke SPORES on important LA expressions from the inner loops of the input program.

4. EVALUATION

We evaluate SPORES to answer three research questions about our approach of relational equality saturation:

- **Section 4.1**: can SPORES derive hand-coded rewrite rules for sum-product optimization?
- **Section 4.2**: can SPORES find optimizations that lead to greater performance improvement than hand-coded rewrites and heuristics?
- **Section 4.3**: does SPORES induce compilation overhead afforded by its performance gain?

We ran experiments on a single node with Intel E74890 v2 @ 2.80GHz with hyper-threading, 1008 GB RAM, 1 Nvidia P100 GPU, 8TB disk, and Ubuntu 16.04.6. We used OpenJDK 1.8.0, Apache Hadoop 2.7.3, and Apache Spark 2.4.4. Spark was configured to run locally with 6 executors, 8 cores/executor, 50GB driver memory, and 100GB executor memory. Our baselines are from Apache SystemML 1.2.0 and TensorFlow r2.1. We compile all TensorFlow functions with XLA through `tf.function`, and enable GPU.

4.1 Completeness of Relational Rules

Theoretically, our first hypothesis is validated by the fact that our relational equality rules are complete w.r.t. linear algebra semantics. To test completeness in practice[6], our first set of experiments check if SPORES can derive the hand-coded sum product rewrite rules in SystemML. To do this, we input the left hand side of each rule into SPORES, perform equality saturation, then check if the rule's right hand side is present in the saturated graph. The optimizer is able to derive all 84 sum-product rewrite rules in SystemML using relational equality rules. Refer to the long version of this paper [43] for a list of these rewrites. We believe replacing the 84 ad-hoc rules with our translation rules R_{LR} and equality rules R_{EQ} would greatly simplify SystemML's codebase. Together with equality saturation, our relational rules can also lead to better performance, as we demonstrate in the next set of experiments.

[6] *"I have only proved it correct, not tried it"* – Donald Knuth

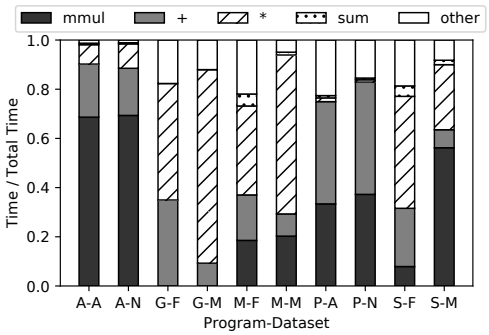

Figure 7: Run-time profile of benchmark programs.

4.2 Run Time Measurement

We compare SPORES against SystemML's native optimizations for their performance impact. As baseline, we run SystemML with optimization level 2 (`opt2`), which is its default and includes all advanced rewrites like constant folding and common subexpression elimination. We additionally enable SystemML's native sum-product rewrites and operator fusion. When using SPORES, we disable SystemML's native sum-product rewrites, which means disabling the 84 rules discussed in Section 4.1. We compile and execute 5 real-world algorithms including Generalized Linear Model (GLM), Multinomial Logistic Regression (MLR), Support Vector Machine (SVM), Poisson Nonnegative Matrix Factorization (PNMF), and Alternating Least Square Factorization (ALS). We configure GLM and MLR to learn a probit model as a binary classifier. We take the implementation of these algorithms from SystemML's performance benchmark suite [39]. All algorithms were used as benchmarks in previous optimization research [9] [4]. We use the same input datasets from [4], specifically the Amazon books review dataset (Amazon/A) [15], the Airline on-time performance dataset (Flights/F) [1], the Netflix movie rating dataset (Netflix/N) [22], and the MNIST8M dataset (MNIST/M) [5]. In order to fit the computation in memory, we down sample each dataset to obtain inputs of small, medium and large sizes. For the Amazon and Netflix data, AS/NS contian 25k reviews, AM/NM 50k, and AL/NL 100k. We convert the data to a review matrix, where columns are items and rows are customers, then use it as input to ALS and PNMF. For the Flights and MNIST datasets, FS/MS contain 2.5M rows, FM/MM 5M, and FL/ML 10M. We use these datasets as input to GLM / MLR / SVM. Each algorithm learns if a flight is delayed more than 5 hours, or if an image shows the digit 2. In TensorFlow experiments we generate random inputs that match the size of the intermediate data in the corresponding benchmark. Our approach focuses on optimizing sum-product operations with the assumption that these operations take up the majority of run time in machine learning programs. We test this assumption by profiling our benchmark programs on the largest version of each dataset. Figure 7 shows that for each program on each input dataset, sum-product operations including matrix multiply, addition, point-wise multiply and summation together take up the great majority of run time (from 77.4% to 98.9%). For other heavy-hitting operations, we implement standard rewrite rules as discussed in Section 3.3. Figure 8 shows the program run time under SPORES optimization against SystemML's optimization. SPORES is competitive with the hand-coded rules in SystemML: for

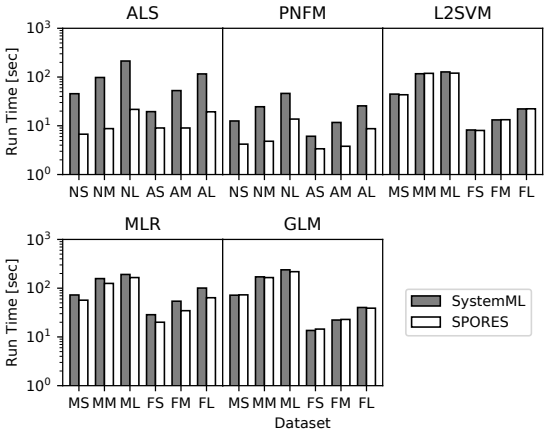

Figure 8: Run time under SystemML/SPORES compilation.

GLM and SVM, SPORES discovers the same optimizations as SystemML does. For ALS, MLR and PNMF, SPORES found new optimizations that lead to up to 10X speedup. We next analyze each benchmark in detail.

For **ALS**, SPORES leads to up to 10X speedup beyond SystemML's optimizations using our relational rules. Investigating the optimized code reveals the speedup comes from a rather simple optimization: SPORES expands $(UV^T - X)V$ to $UV^TV - XV$ to exploit the sparsity in X. Before the optimization, all three operations (2 matrix multiply and 1 minus) in the expression create dense intermediates because U and V are dense. After the optimization, XV can be computed efficiently thanks to the sparsity in X. UV^TV can be computed in one go without intermediates, taking advantage of SystemML's `mmchain` operator for matrix multiply chains. Although the optimization is straightforward, it is counter-intuitive because one expects computing $A(B + C)$ is more efficient than $AB + AC$ if one does not consider sparsity. For the same reason, SystemML simply does not consider distributing the multiplication and misses the optimization.

For **PNMF**, the speedup of up to 3.5X using RA rules attributes to rewriting $sum(WH)$ to $sum_{col}(W) \cdot sum_{row}(H)$ which avoids materializing a dense intermediate WH. Interestingly, SystemML includes this rewrite rule but did not apply it during optimization. In fact, SystemML only applies the rule when WH does not appear elsewhere, in order to preserve common subexpression. However, although WH is shared by another expression in PNMF, the other expression can also be optimized away by another rule. Because both rules uses heuristics to favor sharing CSE, neither fires. This precisely demonstrates the limitation of heuristics.

For **MLR**, SPORES leads to up to 1.3X speedup. The important optimization[7] is $P * X - P * sum_{row}(P) * X$ to $P*(1-P)*X$, where P is a column vector. This takes advantage of the `sprop` fused operator in SystemML to compute $P*(1-P)$, therefore allocating only one intermediate. Note that the optimization factors out P, which is the exact opposite to the optimization in ALS that distributes multiply. Naive rewrite rules would have to choose between the two directions, or resort to heuristics to break ties.

In summary, SPORES improves performance consistently for ALS, PNMF and MLR across different data sizes. The impact of sparsity on performance can be gleaned from the

[7]Simplified here for presentation. In the source code P and X are not variables but consist of subexpressions.

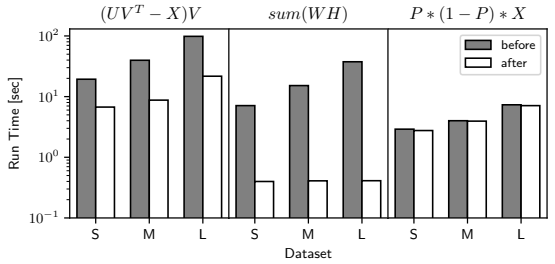

Figure 9: Run time before- and after-rewrite in TensorFlow.

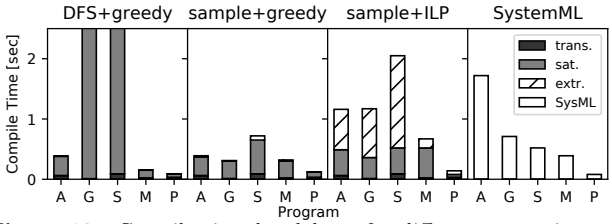

Figure 10: Compile time breakdown for different saturation and extraction strategies. Depth-first saturation reaches the 2.5s timeout compiling GLM and SVM.

particular optimizations: ALS optimization takes advantage of sparsity, while PNMF and MLR optimizations apply for either dense or sparse inputs.

For **SVM** and **GLM**, equality saturation finds the same optimizations as SystemML does, leading to speedup mainly due to operator fusion. Upon inspection, we could not identify better optimizations for **SVM**. For **GLM**, however, we discovered a manual optimization that should improve performance in theory, but did not have an effect in practice since SystemML cannot accurately estimate sparsity to inform execution.

Comparison Against TensorFlow

We ran additional experiments in TensorFlow to see if it can also benefit from SPORES's optimizations. For each of the 3 rewrites we discussed in Section 4.2, we coded the expressions before- and after-rewrite in TensorFlow. Then we compile each version with TensorFlow XLA and measure its run time. Figure 9 shows up to 90X and 50X speedup from the rewrites taken from ALS ($(UV^T - X)V$) and PNMF ($sum(WH)$) respectively. Upon inspection of the compiled code, we found XLA performs no optimization on the input expressions, likely due to certain heuristics preferring the unoptimized versions. For MLR ($P*(1-P)*X$), XLA compiles the before- and after-rewrite expressions to the same code, so the run time stays the same. We were unable to compile our full benchmarks with XLA because the latter cannot compile certain operations on sparse matrices. When this improves, we expect to see SPORES bring its full potential to TensorFlow.

4.3 Compilation Overhead

In our initial experiments, SPORES induces nontrivial compilation overhead compared to SystemML's native rule-based rewrites. Figure 10 (sampling, ILP extraction) shows the compile time breakdown for each benchmark, and the majority of time is spent in the ILP solver. We therefore experiment with the greedy algorithm described in Section 3.1 to see if we can trade off guarantees of optimality for a shorter compile time. Figure 11 shows the performance impact of greedy extraction, and Figure 10 (sampling, greedy

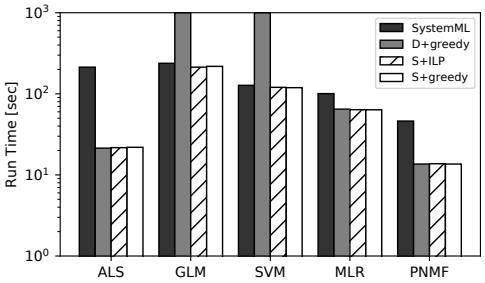

Figure 11: Performance impact of different saturation and extraction strategies. S is saturation with sampling, and D is depth-first saturation. Depth-first saturation runs into timeout compiling GLM and SVM.

extraction) shows the compile time with it. Greedy extraction significantly reduces compile time without sacrificing *any* performance gain! This is not surprising in light of the optimizations we discussed in Section 4.2: all of these optimizations improve performance regardless of common subexpressions, so they are selected by both the ILP-based and the greedy extractor.

We also compare saturation with sampling against depth-first saturation in terms of performance impact and compile time. Recall the depth-first saturation strategy applies all matches per rule per iteration. As Figure 10 shows, sampling is slightly slower for ALS, MLR and PNMF, but resolves the timeout for GLM and SVM. This is because sampling takes longer to converge when full saturation is possible, and otherwise prevents the graph from blowing up before reaching the iteration limit. Indeed, saturation converges for ALS, MLR and PNMF, which means SPORES can guarantee the optimality of its result under the given cost model. Saturation does not not converge before reaching the iteration limit for GLM and SVM because of deeply nested $*$ and $+$ in the programs. Convergence may come as a surprise despite E-Graph's compaction – expansive rules like associativity and commutativity commonly apply in practice. However, the expression DAGs we encounter are often small (no more than 15 operators), and large DAGs are cut into small pieces by optimization barriers like uninterpreted functions.

Figure 10 compares the overall DAG compilation overhead of SystemML against SPORES with different extraction strategies. Note that the overhead of SystemML also includes optimizations unrelated to sum-product rewrites that are difficult to disentangle, therefore it only gives a sense of the base compilation time and does not serve as head-to-head comparison against SPORES. Although SPORES induces significant compilation overhead in light of the total DAG compilation time of SystemML, the overhead is afforded by its performance gain. As we did not focus our efforts on reducing compile time, we believe there is plenty room for improvement, for example organizing rewrite rules to speed up saturation.

4.4 Numerical Considerations

Although our rewrite rules preserve semantics for the reals, they do not preserve semantics under floating point arithmetics. We therefore run experiments to see if SPORES sacrifices numerical accuracy. For L2SVM, we compare the accuracy of the trained model under SPORES / SystemML optimization; for MLR and GLM we compare the R^2 value; PNMF and ALS terminate after some loss falls below a threshold, therefore we compare the number of iterations until termination. Table 3 shows all these statistics are identical under SPORES / SystemML optimzation. Although SPORES does not optimize for numerical accuracy, equality saturation was used by Herbie [33] for that exact purpose. We are actively collaborating with Herbie's authors to develop a multi-objective optimizer for accuracy and run time.

Table 3: Numerical characteristics of compiled programs.

Optimizer	SysML	SPORES	SysML	SPORES
Dataset	Airline		MNIST	
SVM (*acc.*)	82.4%	82.4%	96.8%	96.8%
MLR (R^2)	0.773	0.773	0.742	0.742
GLM (R^2)	0.618	0.618	0.671	0.671
Dataset	Netflix		Amazon	
PNMF (*iter.*)	18	18	36	36
ALS (*iter.*)	8	8	9	9

5. RELATED WORK

There is a vast body of literature for both relational query optimization and optimizing compilers for machine learning. Since we optimize machine learning programs through a relational lens, our work relates to research in both fields. As we have pointed out, numerous state-of-the-art optimizing compilers for machine learning resort to syntactic rewrites and heuristics to optimize linear algebra expressions [3] [38] [16]. We distinguish our work which performs optimization based on a relational semantics of linear algebra and holistically explore the complex search space. A majority of relational query optimization focus on join order optimization [11] [29] [30] [35]; we distinguish our work which optimizes programs with join (product), union (addition), and aggregate (sum) operations. Sum-product optimization considers operators other than join while optimizing relational queries. Recent years have seen a line of excellent theoretical and practical research in this area [25] [20]. These work gives significant improvement for queries involving $\times$ and $\sum$, but fall short of LA workloads that occur in practice. We step past these frameworks by incorporating common subexpressions and incorporating addition ($+$).

In terms of approach, our design of relational IR ties in to research that explores the connection between linear algebra and relational algebra. Our design and implementation of the optimizer ties into research that leverage equality saturation and AND-OR DAGs for query optimization and compiler optimization for programming languages. Since we focus on optimizing sum-product expressions in linear algebra, our work naturally relates to research in sum-product optimization. We now discuss these three lines of research in detail.

5.1 Relational Algebra and Linear Algebra

Elgamal et al. [9] envisions SPOOF, a compiler for machine learning programs that leverages relational query optimization techniques for LA sum-product optimization. We realize this vision by providing the translation rules from LA to RA and the relational equality rules that completely represents the search space for sum-product expressions. One important distinction is, Elgamal et al. proposes *restricted relational algebra* where every expression must have at most two free attributes. This ensures every relational expression in every step of the optimization process to be expressible in LA. In contrast, we remove this restriction and only require the optimized output to be in linear algebra. This allows us to trek out to spaces not covered by linear algebra equality

rules and achieve completeness. In addition to sum-product expressions, Elgamal et al. also considers selection and projection operations like selecting the positive entries of a matrix. We plan to explore supporting selection and projection in the future. Elgamal et al. also proposes compile-time generation of fused operators, which is implemented by Boehm et al. [4]. SPORES readily takes advantage of existing fused operators, and we plan to explore combining sum-product rewrite with fusion generation in the future.

MorpheusFI by Li et al. [28] and LARA by Hutchison et al. [17] explore optimizations across the interface of machine learning and database systems. In particular, MorpheusFI speeds up machine learning algorithms over large joins by pushing computation into each joined table, thereby avoiding expensive materialization. LARA implements linear algebra operations with relational operations and shows competitive optimizations alongside popular data processing systems. Schleich et al.[34] and Khamis et al.[24] explore in-database learning, which aims to push entire machine learning algorithms into the database system. We contribute in this space by showing that even without a relational engine, the relational abstraction can still benefit machine learning tasks as a powerful intermediate abstraction. Kotlyar et.al. [27] explore compiling sparse linear algebra via a relational abstraction. We contribute by providing a simple set of rewrite rules and prove them complete.

5.2 Equality Saturation and AND-OR DAGs

Equality saturation and AND-OR DAGs have been applied to optimize low-level assembly code [21], Java programs [40], database queries [11], floating point arithmetics [33], and even computer-aided design models [31]. The design of our relational IR brings unique challenges in adopting equality saturation. Compared to database query optimizers that focus on optimizing join orders, unions and aggregates play a central role in our relational IR and are prevalent in real-world programs. As a result, our equality rules depend on the expression schema which is not immediately accessible from the syntax. We propose class invariants as a solution to access schema information, and show it to be a powerful construct that enables constant folding and improves cost estimation. Compared to optimizers for low-level assembly code or Java program, we commonly encounter linear algebra expressions that trigger expansive rules and make saturation convergence impractical. We propose to sample rewrite matches in order to achieve good exploration without full saturation. Equality saturation takes advantage of constraint solvers which have also been applied to program optimization and query optimization. In particular, the use of solvers for satisfiability modulo theories by [36] has spawned a paradigm now known as *program synthesis*. In query optimization research, [41] applies Mixed Integer Linear Programming for optimizing join ordering. Although constraint solvers offer pleasing guarantees of optimality, our experiments show their overhead does not bring significant gains for optimizing LA expressions.

5.3 Low-level Code Generation

Novel machine learning compilers including TACO [26], TVM [7], TensorComprehension [42] and Tiramisu [2] generate efficient low-level code for kernel operators. These kernels are small expressions that consist of a few operators. For example the MATTRANSMUL kernel in TACO implements $\alpha A^T x + \beta z$. The kernels are of interest because they commonly appear in machine learning programs, and generating efficient low-level implementation for them can greatly impact performance. However, these compilers cannot perform algebraic rewrite on large programs as SPORES does. For example, TACO supports only plus, multiply and aggregate, whereas SPORES supports any custom functions as discussed in Section 3.3; Tiramisu requires tens of lines of code just to specify matrix multiply which is a single operator in SPORES. Furthermore, the basic polyhedral model in Tiramisu and TensorComprehension does not support sparse matrices. Sparse extensions exist, but require the user to make subtle tradeoffs between expressivity and performance [37]. At a high level, we view these kernel compilers as complementary to SPORES. The former can provide efficient kernel implementation just like the fused operators in SystemML, and we can easily include these kernels in SPORES for whole-program rewrite. The TASO compiler [19] combines kernel-level rewrite with whole-program rewrite, and is also driven by a set of equality rules like SPORES. However, it induces significant overhead – generating operator graphs with just up to 4 operators takes 5 minutes, and while [19] does not include detailed time for compiling whole programs, it reports the compilation finishes in "less than ten minutes". In contrast, SPORES takes seconds instead of minutes in compilation.

6. LIMITATIONS AND FUTURE WORK

We intend SPORES to be used to optimize machine learning programs that perform linear algebra operations. As such, SPORES is not a linear algebra solver, and operations like matrix inversion and calculating eigenvalues are out of scope. SPORES does not include a matrix decomposition operator, but the programmer can implement decomposition algorithms like our ALS and PNMF benchmarks. The operations we support already cover a variety of algorithms as we show in the evaluation. Similar scope is shared by existing research [4] [9] [26]. Although SPORES does not focus on deep learning, its design can be adapted to optimize deep models. We are experimenting to incorporate the identity rules from TASO into our framework. It would be challenging to extend the completeness theorem to the complex operators used in deep learning, but we expect our extension of equality saturation can find high-impact optimizations with short compile time. Finally, although our rewrite rules are complete, we had to resort to rule sampling and greedy extraction to cut down the overhead. Future work can investigate more intelligent rule application and extraction strategies. For example, the optimizer can learn which rules more likely lead to performance improvement and prioritizes firing those rules. Another direction is to incorporate plus into theoretical sum-product frameworks like FAQ [25] and guarantee optimality.

Acknowledgement

We would like to thank Alexandre Evfimievski, Matthias Boehm, and Berthold Reinwald for their insights in SystemML internals; Zachary Tatlock, Max Willsey and Chandrakana Nandi for their valuable feedback. This work is funded by NSF #1954222 and #1907997.

7. REFERENCES

[1] A. S. A. (ASA). Airline on-time performance dataset. `stat-computing.org/dataexpo/2009/the-data.html`.

[2] R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. P. Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In M. T. Kandemir, A. Jimborean, and T. Moseley, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, pages 193–205. IEEE, 2019.

[3] M. Boehm. Apache systemml. In S. Sakr and A. Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*. Springer, 2019.

[4] 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*, 11(12):1755–1768, 2018.

[5] L. Bottou. The infinite mnist dataset. `leon.bottou.org/projects/infimnist`.

[6] S. Chaudhuri and M. Y. Vardi. Optimization of *Real* conjunctive queries. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 25-28, 1993, Washington, DC, USA*, pages 59–70, 1993.

[7] T. Chen, L. Zheng, E. Q. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy. Learning to optimize tensor programs. In S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 3393–3404, 2018.

[8] S. Cohen, Y. Sagiv, and W. Nutt. Equivalences among aggregate queries with negation. *ACM Trans. Comput. Log.*, 6(2):328–360, 2005.

[9] T. Elgamal, S. Luo, M. Boehm, A. V. Evfimievski, S. Tatikonda, B. Reinwald, and P. Sen. SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning. In *CIDR*, 2017.

[10] C. Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982.

[11] G. Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.*, 18(3), 1995.

[12] T. J. Green. Containment of conjunctive queries on annotated relations. In *Database Theory - ICDT 2009, 12th International Conference, St. Petersburg, Russia, March 23-25, 2009, Proceedings*, pages 296–309, 2009.

[13] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In L. Libkin, editor, *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, pages 31–40. ACM, 2007.

[14] L. Gurobi Optimization. Gurobi optimizer reference manual, 2019.

[15] R. He and J. J. McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In J. Bourdeau, J. Hendler, R. Nkambou, I. Horrocks, and B. Y. Zhao, editors, *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, pages 507–517. ACM, 2016.

[16] B. Huang, S. Babu, and J. Yang. Cumulon: optimizing statistical data analysis in the cloud. In K. A. Ross, D. Srivastava, and D. Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1–12. ACM, 2013.

[17] D. Hutchison, B. Howe, and D. Suciu. Laradb: A minimalist kernel for linear and relational algebra computation. *CoRR*, abs/1703.07342, 2017.

[18] Y. E. Ioannidis and R. Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. *ACM Trans. Database Syst.*, 20(3):288–324, 1995.

[19] Z. Jia, O. Padon, J. J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In T. Brecht and C. Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 47–62. ACM, 2019.

[20] M. R. Joglekar, R. Puttagunta, and C. Ré. Ajar: Aggregations and joins over annotated relations. In *PODS*. ACM, 2016.

[21] R. Joshi, G. Nelson, and K. H. Randall. Denali: A goal-directed superoptimizer. In J. Knoop and L. J. Hendren, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 304–314. ACM, 2002.

[22] Kaggle. Netflix prize data. `kaggle.com/netflix-inc/netflix-prize-data`.

[23] D. Kernert, F. Köhler, and W. Lehner. Spmacho - optimizing sparse linear algebra expressions with probabilistic density estimation. In G. Alonso, F. Geerts, L. Popa, P. Barceló, J. Teubner, M. Ugarte, J. V. den Bussche, and J. Paredaens, editors, *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015*, pages 289–300. OpenProceedings.org, 2015.

[24] M. A. Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. In-database learning with sparse tensors. *CoRR*, abs/1703.04780, 2017.

[25] M. A. Khamis, H. Q. Ngo, and A. Rudra. FAQ: Questions Asked Frequently. In *PODS*. ACM, 2016.

[26] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. P. Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, 2017.

[27] V. Kotlyar, K. Pingali, and P. Stodghill. A relational approach to the compilation of sparse matrix programs. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par '97 Parallel Processing, Third International Euro-Par Conference, Passau, Germany, August 26-29, 1997, Proceedings*, volume 1300 of *Lecture Notes in Computer Science*, pages 318–327. Springer, 1997.

[28] 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.

[29] G. Moerkotte and T. Neumann. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *VLDB*, 2006.

[30] G. Moerkotte and T. Neumann. Dynamic Programming Strikes Back. In *SIGMOD*, 2008.

[31] C. Nandi, A. Anderson, M. Willsey, J. R. Wilcox, E. Darulova, D. Grossman, and Z. Tatlock. Using e-graphs for CAD parameter inference. *CoRR*, abs/1909.12252, 2019.

[32] C. G. Nelson. *Techniques for Program Verification.* PhD thesis, Stanford University, Stanford, CA, USA, 1980. AAI8011683.

[33] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. In D. Grove and S. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 1–11. ACM, 2015.

[34] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In F. Özcan, G. Koutrika, and S. Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 3–18. ACM, 2016.

[35] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[36] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In J. P. Shen and M. Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415. ACM, 2006.

[37] M. M. Strout, M. W. Hall, and C. Olschanowsky. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proceedings of the IEEE*, 106(11):1921–1934, 2018.

[38] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. Optiml: An implicitly parallel domain-specific language for machine learning. In L. Getoor and T. Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, pages 609–616. Omnipress, 2011.

[39] SystemML. Systemml performance tests. `https://github.com/apache/systemml/tree/master/scripts/perftest`.

[40] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. *Logical Methods in Computer Science*, 7(1), 2011.

[41] I. Trummer and C. Koch. Solving the join ordering problem via mixed integer linear programming. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1025–1040. ACM, 2017.

[42] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[43] 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. *CoRR*, abs/2002.07951, 2020.

[44] M. Willsey, Y. R. Wang, O. Flatt, C. Nandi, P. Panchekha, and Z. Tatlock. egg: Easy, efficient, and extensible e-graphs, 2020.