Optimizing Tensor Programs on Flexible Storage

MAXIMILIAN SCHLEICH, RelationalAI, USA
AMIR SHAIKHHA, University of Edinburgh, United Kingdom
DAN SUCIU, University of Washington, USA

Tensor programs often need to process large tensors (vectors, matrices, or higher order tensors) that require a specialized storage format for their memory layout. Several such layouts have been proposed in the literature, such as the Coordinate Format, the Compressed Sparse Row format, and many others, that were especially designed to optimally store tensors with specific sparsity properties. However, existing tensor processing systems require specialized extensions in order to take advantage of every new storage format. In this paper we describe a system that allows users to define flexible storage formats in a declarative tensor query language, similar to the language used by the tensor program. The programmer only needs to write *storage mappings*, which describe, in a declarative way, how the tensors are laid out in main memory. Then, we describe a cost-based optimizer that optimizes the tensor program for the specific memory layout. We demonstrate empirically significant performance improvements compared to state-of-the-art tensor processing systems.

CCS Concepts: • **Information systems → Query optimization**; **Query planning**; **Data layout**.

Additional Key Words and Phrases: databases, tensor programs, query optimization, dense data

ACM Reference Format:
Maximilian Schleich, Amir Shaikhha, and Dan Suciu. 2023. Optimizing Tensor Programs on Flexible Storage. *Proc. ACM Manag. Data* 1, 1, Article 37 (May 2023), 27 pages. https://doi.org/10.1145/3588717

1 INTRODUCTION

Linear algebra and, more generally, tensor algebra is used in a wide variety of domains, such as science, engineering, machine learning, data analysis. Tensors are natural generalizations of vectors and matrices from 1 and 2 dimensions to arbitrary dimensions, and highly optimized implementations of tensor algebra operations are available today in several popular libraries, such as SciPy, PyTorch, Julia, TensorFlow, or Matlab. While these libraries are highly optimized for individual operations, compound operations require users to create temporary tensors, which often destroys the locality and may even lead to out of memory errors, when the intermediate results are too large. Such operations are frequently encountered in complex tensor programs, or tensor kernels, terms that we will use interchangeably in this paper.

Several domain specific languages have been proposed for expressing and optimizing entire tensor programs. Examples include SystemML [10], TVM [12], Halide [39], Taco [27], TASO [24]. The compiler community has addressed one challenge of the optimization problem, namely the separation of the *algorithm* from the *schedule*. This idea was introduced by the Halide language, which was designed for high-performance code generation for image processing pipelines [38, 39]. The programmer writes the algorithm in an imperative, high-level language, and writes separately a *schedule*, which specifies low level optimizations, such as tiling, vectorization, or loop unrolling.

Authors' addresses: Maximilian Schleich, RelationalAI, USA; Amir Shaikhha, University of Edinburgh, United Kingdom; Dan Suciu, University of Washington, USA.

2836-6573/2023/5-ART37
https://doi.org/10.1145/3588717

TVM [12] extends this principle from image processing to tensor processing for general-purpose ML applications.

In this work we are not concerned with schedules, but with a different challenge in tensor processing: optimizing the query plan based on how the tensors are stored in memory. *Storage* refers in this paper to *memory layout*, and not to persistent representation. When a tensor is sparse, the programmer has many choices for representing it in main memory, and the best plan for a tensor program varies dramatically depending on what storage was chosen, the statistics of the data (e.g. how sparse or dense), and the particular tensor program. For example, a vector $a(i)$ can be stored as a dense array, or as a hash table indexed by i, or as two parallel arrays storing the indices i and the values $a(i)$. When the tensors are dense, then the best plan may be to use a linear algebra library [31, 48], while for sparse tensors a better plan may be to use relational query operators, e.g. hash joins. Tensors can be easily represented as relations, and tensor programs can be expressed as SQL queries [9], but relational engines are not designed to support storage formats specifically optimized for sparse tensors (e.g. the CSR format discussed later).

To address the storage problem, Taco [27] separates the *tensor storage format* from the tensor program. In the storage format the user can specify separately for each dimension whether it is dense or sparse, and can also order the dimensions, leading to $d! \cdot 2^d$ possible formats for a d-dimensional tensor.[1] Given a tensor program, the Taco compiler generates code that optimizes the access to the storage formats. Taco does not perform cost-based optimizations, which means that the programmer still needs to be aware of the storage specification. For example, if the vector a is sparse while b, c are dense, then $a(i) * (b(i) + c(i))$ is best rewritten as $a(i) * b(i) + a(i) * c(i)$, because computing $b(i) + c(i)$ results in a large, dense vector, while $a(i) * b(i)$ is a small, sparse vector, no larger than a, and similarly for $a(i) * c(i)$. However, the task of rewriting the expression is left to the programmer.

In this paper we propose a rule-base approach to optimizing tensor programs over flexible tensor storage, using a cost-based optimizer. The main novelty in our approach is that the storage descriptors themselves are also defined in the same declarative language as the tensor program. To specify how a tensor is stored, the user writes a *storage mapping* from the physical data structures (arrays and/or hash tables) to the logical tensor. Our system evaluates the tensor program by first composing it with the storage mappings, then optimizing it using rewrite rules. This improves in two ways over previous systems. First, the storage formats are no longer hard coded, but the user is free to define their own. For example, users may describe one of the popular storage formats COO, CSR, etc, or define a format optimized for upper-triangular matrices, or for band-matrices, or a space-filling curve, etc. There is no bound on the number of storage representations, the only limit is the expressivity of the query language and the power of the optimizer. Second, the optimizer is now able to perform a rich collection of high-level optimizations, such as factorization, loop fusion, or join reordering, and optimize the tensor program specifically for the given storage. For example, the optimizer may consider both expressions $a(i) * (b(i) + c(i))$ and $a(i) * b(i) + a(i) * c(i)$, and choose the optimal one based on their physical storage and data statistics. To the best of our knowledge, our system, called STOREL, is the first cost-based optimizer for a declarative tensor language. We show in Sec. 6 that, due to the rewrite rules, STOREL significantly outperforms both Taco [27] (a tensor algebra) and DuckDB [37] (an optimized relational engine) for several tensor programs, although their physical execution engines are as good as, or even better than ours.

Fig. 1 illustrates how STOREL processes the matricized tensor times Khatri-Rao product, MTTKRP [27], $A(i, j) = \sum_{k,l} B(i, k, l) \cdot C(k, j) \cdot D(l, j)$. Fig. 1 (a) shows the tensor program written in our declarative tensor language SDQLite (described in Sec. 3), while (b) shows the Compressed

[1] Taco was later extended to support 6 formats per dimension [14].

```
CREATE TENSOR A AS
  sum(<(i,k,l), B_v> in B, <(k,j), C_v> in C,
       <(j,l), D_v> in D)
    { (i, j) -> B_v * C_v * D_v }
```

(a) MTTKRP kernel $A(i,j) = \sum_{k,l} B(i,k,l) \cdot C(k,j) \cdot D(l,j)$ in SDQLite.

```
CREATE TENSOR C AS sum(<row,_> in 0:C_len1)
 { row ->
   sum(<off,col> in C_idx2(C_pos2(row):C_pos2(row+1)))
     { col -> C_val(off) }
 }
```

(c) The SDQLite storage mapping for CSR.

C:

6	0	9	8
0	0	0	0
5	0	0	7

C_len1:	3				
C_pos2:	0	3	3	5	
C_idx2:	0	2	3	0	3
C_val:	6	9	8	5	7

(b) Matrix C and its CSR format from [27, Fig.5(f)].

```
sum(<i_pos, i> in B_idx1)
  { i ->
    sum(<k_pos, k> in B_idx2( B_pos2(i_pos):B_pos2(i_pos+1) ))
      sum(<j_pos, j> in C_idx( C_pos(k):C_pos(k+1) ))
        {
          j ->
          C_val(j_pos) * (
            merge(<l_posB,l_posD,l> in <B_idx3( B_pos3(k_pos):B_pos3(k_pos+1) ),
                D_idx( D_pos(j):D_pos(j+1) )>)
              B_val(l_posB) * D_val(l_posD)
          )
        }
  }
```

(d) Optimized MTTKRP in SDQLite.

Fig. 1. Illustration of STOREL. (a) MTTKRP kernel, (b) CSR memory layout, (c) CSR storage mapping, (d) optimized plan.

Sparse Row (CSR) memory layout of matrix `C`, which is one of several formats described in [14, 27] (reviewed in Sec. 2). For each matrix or tensor, the user describes its memory layout by writing a storage mapping, also in SDQLite; the storage mapping for the matrix `C` is shown in Fig. 1 (c), and similar storage mappings need to be defined for `B` and `D`. To execute the program, the system composes the tensor program with the storage mappings, then chooses an optimal plan using a cost-based optimizer; the optimal plan is shown in Fig. 1 (d). While the plan could be further optimized for some sophisticated schedule (as done by Halide, TVM, and Taco), we currently do not support schedules and simply run the optimal plan directly in Julia.

The main challenge in developing the cost-based optimizer is the right choice of tensor processing language. All query optimizers use the relational algebra as intermediate language. However, we found that a calculus, rather than an algebra, is better suited for optimizing tensors; here *calculus* refers to a language with explicit use of variables, while *algebra* refers to a variable-free language. There are two reasons for that. First, the physical plan of a tensor program consists of for-loops with explicit variables. They look like this: `for i=1:m do for j=1:n do ...` instead of this: $A \bowtie (B \bowtie \cdots)$, and optimizing directly expressions with variables simplifies the generation of the physical plan. Second, the intermediate language for tensor programs needs to support nested collections, which occur in sparse formats like CSR, CSC, CSF, while standard relational algebra,

as well as some recent extensions to linear algebra [23] support only flat collections. Algebras for nested collections exists, but they tend to be much harder to read than calculus, making it harder to design and debug optimization rules, e.g. compare the rules in Table 2 [55] to those in Fig. 3 in our paper. For these reasons, we opted for a calculus-based intermediate language. We are not the first to use a calculus-based intermediate language for query optimization: Worst Case Optimal Join algorithms are also described as nested loops, in effect using a calculus as intermediate language [18, 35, 51].

We describe in this paper a language, called SDQLite, used both for writing tensor programs, and for performing optimizations. SDQLite has a syntax that is reminiscent of Unions of Conjunctive Queries, but where $\wedge, \vee, \exists$ are replaced by `*`, `+`, **`sum`**, to which we add **`let`**-bindings, and nested *dictionaries* as data model; our dictionaries are similar to those in SDQL (Semiring-Dictionary Query Language) [42], hence we call our language *SDQLite*. Our language can express tensor programs in a notation close to mathematics, and can express complex storage mappings corresponding to sophisticated tensor memory layouts, including those described in [14, 27]. Any *SDQLite* query can easily be converted directly to a physical, nested for-loop plan, because each quantified variable $i, j, \ldots$ becomes directly a `for` loop over that variable. However, it is more difficult to design an optimizer. For example, Selinger's dynamic programming algorithm for join re-ordering [5, 32] no longer applies, because in a calculus there is no explicit binary join. Instead, our system is entirely rule-based, and the rules must be designed for a calculus rather than an algebra. We designed a suite of 44 SDQLite-rewrite rules, and use the equality saturation system Egg [54] as rewrite engine. Egg uses a compact data structure called an *e-graph* to represent a large number of equivalent expressions as a graph. However, like most term rewriting systems, Egg does not understand variables in rules. For our optimizer, we developed a variant of the De Bruijn index that removes the need for explicit variable representation.

One major motivation for our work is that most of existing tensor and linear algebra systems in the compilers and HPC communities focus on dense data; in contrast, our focus in this work is on sparse data. The reason for the traditional focus on dense data is that Linear Algebra packages were originally developed for use in Physics and Engineering, where tensors are dense, and they support highly optimized kernels for specific operations on dense data. Support in these packages for sparse data is rare.[2] TACO [27] was the first recognized the need to optimize tensor programs over sparse data; our work falls into the latter category.

In summary, we make the following contributions in this paper:

- We describe the architecture of STOREL, where tensor programs and tensor storage mappings are defined in a common language, and optimized jointly (Sec. 3).
- We describe a declarative tensor calculus, *SDQLite*, for both tensor programs and storage mappings, and show that it can express a rich variety of previously proposed storage formats, and beyond (Sec 4).
- We describe a cost-based optimizer for the tensor calculus, which supports a rich suite of optimizations, such as factorization, loop fusion, and join reordering (Sec. 5).
- Finally, we conduct an experimental evaluation showing that STOREL can significantly outperform other tensor processing systems, by using a cost-based optimizer to choose the best plan for the given storage representation (Sec. 6).

2 BACKGROUND

Tensors Given a number n, we denote by $[n) \stackrel{\text{def}}{=} \{0, 1, 2, \ldots, n-1\}$. Let $d \geq 1$ and let $n_1, n_2, \ldots, n_d$ be natural numbers. A *tensor with d dimensions*, is element $\boldsymbol{A} \in \mathbb{R}^{[n_1) \times \cdots \times [n_d)}$. A scalar, a vector, and a

[2] Cf. GitHub issues #43497 for TensorFlow, #72065 for PyTorch, #4332 for TVM.

matrix are tensors with 0, 1, and 2 dimensions respectively. Given d indices, $i_1 \in [n_1), \ldots, i_d \in [n_d)$, we denote by $\boldsymbol{A}(i_1, \ldots, i_d)$ the value of the tensor at those positions; we call each i_j a *dimension*.

Tensor formats We briefly review some popular tensor formats following [14, 27]. A *dense* representation of a tensor consists of a memory array with $n_1 n_2 \cdots n_d$ elements. The *coordinate format*, COO, stores only the non-zero elements in an array, and their coordinates in d separate arrays. For example, the dense and COO representations of the vector $\boldsymbol{v} = (9, 0, 7, 5)$ are:

DENSE:

v_len:	4			
v_val:	9	0	7	5

COO:

v_pos:	0	3	
v_idx:	0	2	3
v_val:	9	7	5

To access $v(i)$ using the COO representation one has to first find i in `v_idx`, in other words one has to find p such that `v_idx`$(p) = i$, then return `v_val`(p); the role of `v_pos` will become clear shortly. The COO representation of a matrix has two index arrays, `v_idx1`, `v_idx2`, storing the rows and columns of the non-zero element respectively. The COO representation is compact, but no longer enables constant-time lookup. A *hash-map* representation of the matrix is a hash-map where the keys are pairs (i, j). It is compact and allows access in time $O(1)$, but no longer supports a scan in either row-major or column-major order.

The Taco system [27] describes a general scheme for storage formats where the user can choose an order of the d dimensions, and specify, independently for each dimension, whether it is dense or sparse. This allows for $d! \cdot 2^d$ formats. The storage uses *segmented* arrays, which consist of the concatenation of several sub-arrays stored in a single array, with their starting positions stored in a separate array. For example, the *sparse-sparse* representation of the matrix $\boldsymbol{C}$ in Fig. 1 (b) is the following (taken from [27, Fig.5(g)]):

C_pos1:	0	2			
C_idx1:	0	2			
C_pos2:	0	3	5		
C_idx2:	0	2	3	0	3
C_val:	6	9	8	5	7

The arrays `C_idx2` and `C_val` contain two segments each: the first segment represents row 0 of the matrix $\boldsymbol{C}$, $(6, 0, 9, 8)$; the second segment represents row 2, $(5, 0, 0, 7)$. The segments are delimited by `C_pos2`, which indicates their starting point. The row number of each segment is stored in `C_idx1`: only the values $i = 0$ and $i = 2$ occur here because row 1 is empty. Alternatively, the *dense-sparse* representation, shown in Fig. 1 (b) stores *every* row, including row 1, and for that reason there is no need to store the vector `C_idx1` (since this vector would be $(0, 1, 2)$), but we only store its length, `C_len1` $= 3$. The dense-sparse representation is called *compressed sparse row*, or CSR, and the sparse-sparse representation is called *doubly CSR*, or DCSR. In a later reference [14] the authors extended the number of choices available at each dimension from 2 to 6.

Tensors as Relations Any d-dimensional tensor can be represented as a relation with $d + 1$ attributes. For example, a matrix A can be represented as a relation $R(i, j, v)$, where i, j is the primary key, and v the value of $A(i, j)$. A clustered index on (i, j, v) corresponds roughly to a row-major ordering of the matrix; a hash-index corresponds to a hash-map representation; while a column-oriented storage [2] corresponds to a COO representation. However, since relations are unordered, it is not possible to use some of the other formats, like CSR or CSC.

Semiring Dictionary A *semiring* is a quintuple $(S, +, *, 0, 1)$, where S is a set, the operations $+, *$ are associative with identities 0 and 1 respectively, $+$ is commutative, $*$ distributes over $+$, and $0 * x = x * 0 = 0$. For example, the real numbers form a semi-ring, $(\mathbb{R}, +, *, 0, 1)$. A *semiring dictionary*, or simply a *dictionary*, is a mapping $K \to S$, from a finite set of keys K to values in some

semiring S [42]. If $k_1, \ldots, k_m$ are distinct keys, then $\{k_1 \to v_1, \ldots, k_m \to v_m\}$ denotes the dictionary that maps each key k_i to the value v_i, and maps each key $k \notin \{k_1, \ldots, k_m\}$ to 0. In other words, missing keys default to 0; it follows that $\{k_1 \to 0, k_2 \to 0, \ldots\} = \{\,\}$, in other words a dictionary containing only 0 values is the same as empty dictionary. In this paper the key space is always of the form $K \stackrel{\text{def}}{=} [n_1) \times \cdots \times [n_d)$, and we view interchangeably a tensor $\boldsymbol{A} \in \mathbb{R}^K$ as a dictionary $\boldsymbol{A} : K \to \mathbb{R}$. When $d = 0$, then the dictionary is of the form $\{() \to v\}$, which we identify, with some abuse, with the scalar value v.

It was observed in [42] that semiring dictionaries generalize K-relations [20]; the set of semiring dictionaries over a fixed key-set K forms another semiring, where the plus and multiplication are defined element-wise. For example, if $\boldsymbol{A}, \boldsymbol{B}$ are two $m \times n$ matrices, then they both can be viewed as dictionaries $[m) \times [n) \to \mathbb{R}$, and $\boldsymbol{A} + \boldsymbol{B}$, $\boldsymbol{A} * \boldsymbol{B}$ denote their element-wise sum and product respectively. One consequence is that we obtain the following rule:

$$\{k_1 \to v_1\} + \{k_2 \to v_2\} = \begin{cases} \{k \to v_1 + v_2\} & \text{if } k_1 = k_2 = k \\ \{k_1 \to v_1, k_2 \to v_2\} & \text{if } k_1 \neq k_2 \end{cases}$$

Another consequence is that one can define nested dictionaries, by defining a dictionary whose values are other dictionaries. For example, let $S \stackrel{\text{def}}{=} [[n) \to \mathbb{R}]$ denote the set of dictionaries with keys $[n)$ and real values. A dictionary in S is a vector of length n. Then, a dictionary $\boldsymbol{A} : [m) \to S$ is a vector of length m of vectors of length n, which is equivalent to a matrix.

SDQL We briefly review here SDQL from [42]. In a nutshell, the query language SDQL is like Unions of Conjunctive Queries, where $\exists, \wedge, \vee$ are replaced with **sum**, $*$, $+$, and the head variables are moved to the end of the query expression. We show here side-by-side in CQ and SDQL how to transform a vector V by removing its negative values (equivalently: setting them to 0) and multiplying the others by 5:

CQ:

$$Q(i, 5 * v) := V(i, v) \wedge (v > 0)$$

SDQL:

```
sum( <i,v> in V )
  if v>0 then { i -> 5*v }
```

The semantics of SDQL uses the fact that dictionaries form a semiring, i.e. can be "added". The SDQL query above is executed by iterating over the pairs $< i, v >$ in V, and summing up singleton dictionaries. For example, if the vector V is $(v_0, v_1, v_2, v_3, v_4)$, where $v_0, v_3, v_4 > 0$ and $v_1, v_2 < 0$, then the query above returns $\{0 \to 5v_0\} + \{3 \to 5v_3\} + \{4 \to 5v_4\} = \{0 \to 5v_0, 3 \to 5v_3, 4 \to 5v_4\}$. For another illustration, the following two SDQL queries compute the dot product $\sum_i u_i v_i$ and the element-wise product $(u_i v_i)_i$ of two vectors U, V respectively:

```
sum(<i,u> in U, <i,v> in V) {() -> u*v}
sum(<i,u> in U, <i,v> in V) {i -> u*v}
```

The operator `*` is overloaded to define the multiplication of scalars and dictionaries. For example, `a * V` represents a scalar-vector multiplication and is equivalent to the following SDQL query:

```
sum(<i,v> in V) {i -> a * v}
```

3 STOREL

Here we describe architecture of our system STOREL, and its declarative language SDQLite.

3.1 Architecture

The architecture of STOREL is shown in Fig. 2. All yellow-gold boxes represent SDQLite programs, described below, while the blue box represents rewrite rules described in Sec. 5. The end user (for example a data scientist) writes a Tensor Program (TP) in SDQLite. Separately, the data administrator

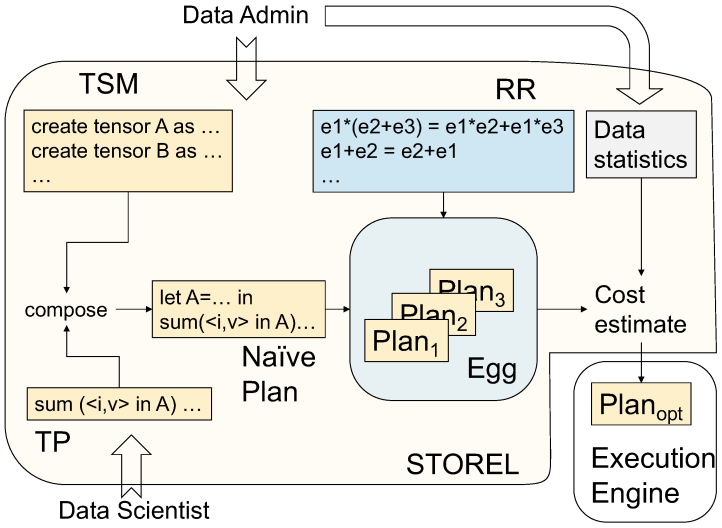

Fig. 2. System's architecture. TSM=Tensor Storage Mapping; TP=Tensor Program; RR=Rewrite Rules.

(possibly the same user) familiar with low level optimizations, writes Tensor Storage Mappings (TSM), one for each tensor. STOREL composes the two expressions, by substituting each tensor variable in TP with its corresponding definition in TSM. This results in a SDQLite expression which we call the Naive Plan. The Naive Plan is then submitted to the Egg equality saturation system [54]. Egg has access to a knowledge base of Rewrite Rules (RR), and applies all rules until saturation, i.e. until no more rule can be applied. The current collection of rewrite rules includes about 44 rules, described in Sec. 5. Egg stores all equivalent plans in a very compact data structure called an *e-graph*. Next, STOREL uses data statistics and a simple cost model to associate a cost to each equivalent plan; currently, the user needs to provide the data statistics manually. Egg then extracts the cheapest plan from the e-graph, and this plan is finally submitted to the execution engine. We currently use Julia [8] as our execution engine. Alternatively, the optimal plan could be further optimized by applying *schedules*, but this is not currently supported in our system.

3.2 SDQLite

STOREL needs a language to express the tensor programs, a formalism for expressing tensor storage formats, an intermediate language in which to express the optimizations, and a physical language in which the programs are executed. In this paper we are introducing a declarative language called SDQLite, which serves the first three purposes: it can express Tensor Programs (TP), it can express sophisticated Tensor Storage Mappings (TSM), and we also use it as intermediate language for performing optimizations. SDQLite can be easily converted to physical plans, as we describe in Sec. 5. A language that satisfies all these goals requires a careful design: we describe SDQLite in this section, and note that it is derived from SDQL [42]; we discuss the differences at the end of this section.

The data model for SDQLite consists of *scalars* (integers or reals), and *nested dictionaries*. The latter have type $[n) \to S$, where the value space S is the set of integers, reals, or another dictionary.

Thus, the data model in SDQLite consists of the following types:

$$[n_1) \to [n_2) \to \cdots [n_d) \to \mathbb{R}$$
$$[n_1) \to [n_2) \to \cdots [n_d) \to \mathbb{Z}$$

where $d \geq 0$; when $d = 0$ then these are scalar types. One can equivalently view the data model of SDQLite as consisting of real- and integer-valued tensors. To see this, we recall the *curry* operation, which converts a function of type $K \times K' \to S$ into a function of type $K \to [K' \to S]$, and the *uncurry* operation which goes the other way. By repeatedly uncurrying a nested dictionary, we can convert it to a $n_1 \times n_2 \times \cdots \times n_d$ tensor. For that reason, in this paper we blur the distinction between (nested) dictionaries and tensors, and view the data model of SDQLite as consisting of tensors.

Syntax The expressions in SDQLite are the following:

```
e1+e2, e1*e2, {k -> e}, e(i), i1:i2, e(i1:i2),
if (c) then e, let v = e1 in e2, sum(<k,v> in e1) e2
```

where `k`, `v` are variables, `e`, `e1`, `e2`, `e3` are dictionary expressions, `i`, `i1`, `i2` are index expression, i.e. of type **`int`**, and `c` is a Boolean expression. Each of the SDQLite expressions above returns a dictionary, which may, in particular, be a scalar. We also include standard primitive operations over scalars of type real or integer, like division `/`, modulo `%`, exponentiation `exp(...)` comparison `e1<e2`, etc.

Semantics We briefly describe the semantics of SDQLite. `e1+e2` and `e1*e2` compute the sum and product of dictionaries; `e(i)` applies the dictionary `e` to the key `i`.

The range expression `i1:i2` returns the dictionary consisting of the sum of `{i -> i}`, for all `i` in the range `i1,...,i2-1`, i.e.:

```
i1:i2 = { i1 -> i1, i1+1 -> i1+1, ..., i2-1 -> i2-1 }
```

The sub-array expression `e(i1:i2)` is used for representing segmented arrays and returns the following dictionary:

```
e(i1:i2) = { i1->e(i1),i1+1->e(i1+1),...,i2-1->e(i2-1) }
```

The conditional **`if`** `(c)` **`then`** `e` returns `e` if `c` is true, and returns zero (`0` or `{}` depending on the type of `e`) otherwise, and the **`let`** construct introduces a temporary variable `v`, with value `e1`, which may be used in `e2`.

We explain the summation. Assume the value of `e1` is:

```
e1 = { k1 -> v1, ..., kn -> vn }
```

Then the value of **`sum`**`(<k,v>` **`in`** `e1) e2` is:

```
e2[k1/k,v1/v] + e2[k2/k,v2/v] + ... + e2[kn/k,vn/v]
```

where `e2[k_i/k,v_i/v]` represents the result of substituting in `e2` the variables `k`, `v` with the values `k_i` and `v_i`.

Syntactic Sugar We also included some convenient syntactic sugar extensions in SDQLite, described in Table 1.

Example 3.1. For a simple illustration, the following SDQLite query computes the product of two matrices `A` and `B`:

```
sum( <(i,j),a> in A, <(j,k),b> in B) { (i,k) -> a*b }
```

It is internally desugared to:

```
sum( <i,rowA> in A) sum(<j1,a> in rowA)
    sum(<j2,rowB> in B) sum(<k,b> in rowB)
        if (j1==j2) { i -> { k -> a*b }}
```

Notice the power of viewing dictionaries as semirings. The semantics of query above consists of emitting n^3 singleton dictionaries of the form `{i->{k->a*b}}`, which are "added" up and result in

Construct	Desugars to	Notes
`e(e1,e2)`	`e(e1)(e2)`	curry
`{ (e1, e2) -> e }`	`{ e1 -> { e2 -> e } }`	curry
`sum(<(k1,k2),v> in e1) e2`	`sum(<k1,w> in e1) sum(<k2,v> in w) e2`	uncurry
`let v1=e1, v2=e2 in ...`	`let v1=e1 in let v2=e2 in ...`	
`sum(<k1,v1> in e1, <k2,v2> in e2) e3`	`sum(<k1,v1> in e1) sum(<k2,v2> in e2) e3`	
`sum(<k,v1> in e1, <k,v2> in e2) e3`	`sum(<k1,v1> in e1) sum(<k2,v2> in e2) if (k1 == k2) then e3`	`k` is used twice in LHS

Table 1. Syntatic sugar extensions in SDQLite.

only n^2 pairs `i,k`. "Addition" acts like a group-by, in other words we have:

```
  {i -> {k -> ai1*b1k}} + {i->{k->ai2*b2k}} + ...
= {i -> {k -> ai1*b1k} + {k->ai2*b2k} + ... }
= {i -> {k -> ai1*b1k+ai2*b2k+...}}
```

Alternatively, if the dimensions of the two matrices are known to be $m \times n$ and $n \times p$, then matrix multiplication can be written as:

```
sum( <i,_> in 0:m, <j,_> in 0:n, <k,_> in 0:p)
  { (i,k) -> A(i,j)*B(j,k) }
```

Discussion SDQLite is a declarative language, in that it does not specify the order of operations. This similar to, say, SQL, where the order of the tables in the `FROM`, or the order of the predicates in the `WHERE` clause do not specify that the joins, or the evaluation of predicates, need to be executed in that order. In fact, SDQLite is basically UCQ, where $\exists$, $\wedge$, $\vee$ are replaced with **sum**, $*$, $+$, as we explained in Sec. 2; UCQ is generally accepted to be a declarative language, and SDQLite is similarly declarative. While SDQLite borrows several ideas from SDQL [42], it differs in some important ways, as follows. SDQLite adds subarray expressions (needed in Sec. 4), the `merge` operator (needed in Sec. 5.6), and has several syntactic sugar extensions, shown in Table 1. We restricted the keys to be numbers only (while SDQL allows records and dictionaries), which is necessary to enable the optimization rules in Sec. 5.2. Finally, we defined cardinality and cost estimation rules and extended optimization rules, as described in Sec. 5.

4 TENSOR STORAGE MAPPINGS

We have described in the previous section the declarative tensor language SDQLite, which can be used to write tensor programs in a notation close to a mathematical notation; as we discussed, we use tensors and dictionaries interchangeably in this paper. So far, all tensors manipulated in SDQLite have only a logical data model, and we have not defined yet a physical model. In this section we extend SDQLite with a physical data model, and show how we use it to define Tensor Storage Mappings (TSM), from a physical to a logical representation.

The physical data model in SDQLite consists of four data types: (a) scalar values, which can be of type **real** or **int**, (b) arrays of type **int** or **real**, and (c) hash-maps that map tuples of integers to

int or **real**, or (d) tries, which are trees of hash-maps. The data admin (Fig. 2) defines named data values using the following syntax:

```
CREATE [real | int] SCALAR U;
CREATE [real | int] ARRAY A(n);
CREATE [real | int] HASHMAP B(n1, n2, ..., nd);
CREATE [real | int] TRIE C(n1)(n2)...(nd);
```

Here U, A, B, C are names, i.e. identifiers, n, n1, ..., nd are expressions of type **int**. An array is a continuous memory array of fixed size n. Both a hash-map and a trie logically represent a dictionary $[n_1) \times \cdots \times [n_d) \to \mathbb{R}$ (or $\cdots \to \mathbb{Z}$), but differ at the physical level. The hash-map maps a key $(i_1, \ldots, i_d)$ to a real or integer value, while a trie is a hash map that maps a key i_1 to another hash-map of type $[n_2) \times \cdots \times [n_d) \to \mathbb{R}$. All symbols U, A, B are global symbols, in contrast to symbols introduced by the **let** binding, which are local.

Next, the data administrator writes a Tensor Storage Mapping (TSM) for each logical tensor, using the following statement:

```
CREATE TENSOR T AS ...;
```

where the ellipsis represent a SDQLite expressions that uses the named data values defined earlier, and returns a logical tensor (dictionary) T.

By combining a simple physical data model with a powerful tensor language, SDQLite allows the data administrator to define sophisticated storage mappings, which helps the administrator exploit the particular characteristics of her tensors.

We illustrate with several TSM examples, showing various formats for representing a matrix C.

Example 4.1. The following TSM defines a dense, row-major representation of C:

```
CREATE int SCALAR M, N; CREATE ARRAY V(M*N);
CREATE TENSOR C AS
  sum (<i,_> in 0:M, <j,_> in 0:N) { (i,j) -> V(i*N+j) };
```

Example 4.2. Suppose we want to store C using the DCSR format (sparse-sparse) described in Section 2. Then we need to define the following physical data types, and TSM:

```
CREATE int ARRAY C_pos1(2);
CREATE int ARRAY C_idx1(C_pos1(1));
CREATE int ARRAY C_pos2(C_pos1(1)+1);
CREATE int ARRAY C_idx2(C_pos2(C_pos1(1)));
CREATE real ARRAY C_val(C_pos2(C_pos1(1)));
CREATE TENSOR C AS
  sum (<i_pos, i> in C_idx1)
    let j_start = C_pos2(i_pos),
        j_end = C_pos2(i_pos+1)
    in sum( <j_pos, j> in C_idx2( j_start:j_end ))
          { (i,j) -> C_val(j_pos)}
```

We then materialize the physical data types as follows:

- C_pos1 has size 2 and values C_pos1(0)=0, C_pos1(1)= the number of non-empty rows in the matrix C.
- C_idx1 contains all indexes i of the nonempty rows in C, in increasing order.
- C_pos2 defines the segments in the arrays C_idx2 and C_val; there is one segment for each non-empty row in the matrix, hence the size of C_pos2 is the number of non-empty rows plus 1. Its last position defines the sizes of C_idx2 and C_val.
- Finally, C_idx2 and C_val contain the segmented arrays that represent the non-empty rows of C as sparse vectors.

The **TENSOR** expression defines how to build C from these arrays.

For a similar example, Fig. 1 (c) defines the Tensor Storage Mapping from the CSR representation Fig. 1 (b) to the matrix C.

Example 4.3. We briefly illustrate **HASHMAP** and **TRIE**. The following represents C using a **HASHMAP**:

```
CREATE real HASHMAP H(M,N);
CREATE TENSOR C AS sum(<(i,j),v> in H) {(i,j) -> v};
```

This is commonly known as the Dictionary of Keys (DOK) format in SciPy [15].

Alternatively, we could store C in a **TRIE** of depth 2:

```
CREATE real TRIE T(M)(N);
CREATE TENSOR C AS
  sum (<i,row> in T, <j,v> in row) { (i,j) -> v };
```

The difference is that now the key of the hash-map T is a single index i, and the value is another hash-map that maps the columns j to values.

We end this section by emphasizing that storage mappings defined in a declarative language like SDQLite are significantly more expressive than fixed, predefined storage formats. For example, it is easy to represent in SDQLite a storage mapping for a dense lower-triangular matrix A, a band matrix B (where B(i,j)<>0 only when abs(i-j)<=1), or the Z-order space-filling curve C, although SDQLite was not explicitly designed for these types of storages:

```
CREATE real ARRAY A_val(N*(N+1)/2);
CREATE TENSOR A AS // lower triangular
  sum(<i,_> in 0:N, <j,_> in 0:(i+1))
     {(i,j) -> A_val(i*(i-1)/2+j)}

CREATE real ARRAY B_val(3*N-2);
CREATE TENSOR B AS // band matrix
  sum(<p,_> in 0:N)
    { (p,p) -> B_val(3*p)} +
    if (p<N)
      then { (p,p+1) -> B_val(3*p+1),
             (p+1,p) -> B_val(3*p+2) }

CREATE real ARRAY C_val(N*N); // N is power of 2
CREATE TENSOR C AS // Z-order curve
   sum (<d,v> in C_val)
      let i = even_bits(d), // even bits of d
          j = odd_bits(d) // odd bits of d
      in { (i,j) -> v }
```

5 OPTIMIZATIONS

We have seen that tensor processing in STOREL consists of two separate tasks: writing the Tensor Program (TP), and writing the Tensor Storage Mappings (TSM), see Fig. 2. The TP simply refers to logical tensor names, like A or B or C, while the TSM describes how these tensors are stored in physical arrays, hash-maps, or tries. Both programs are expressed at a *logical* level, in the same declarative language SDQLite. In this section we describe how STOREL combines these two into a single, optimized *physical* plan, which can be directly executed by an engine; we currently use Julia as our physical execution engine. In this section we will refer to any logical SDQLite query as a

Associativity/Commutativity Rules:			
A1:	`e1 * (e2 * e3)`	↔	`(e1 * e2) * e3`
A2:	`{ e1 -> e2 * e3 }`	↔	`{ e1 -> e2 } * e3`
A3:	`{ e1 -> e2 * e3 }`	↔	`e2 * { e1 -> e3 }`
A4:	`if(e1) then e2 * e3`	↔	`e2 * if(e1) then e3`
C1:	`e1 + e2`	↔	`e2 + e1`
C2:	`e1 == e2`	↔	`e2 == e1`

Algebraic Simplifications:			
L1:	`e + 0`	⇝	`e`
L2:	`e * 0`	⇝	`0`
L3:	`e * 1`	⇝	`e`
L4:	`-0`	⇝	`0`
L5:	`e - 0`	⇝	`e`
L6:	`e - e`	⇝	`0`

Distributivity (Factorization) Rules:					
D1:	`e1 * e2 + e1 * e3`		↔	`e1 * (e2 + e3)`
D2:	`sum(<k,v> in e1) e2 * e3`	if $k,v \notin FV(e2)$	↔	`e2 * (&sum(<k,v> in e1) e3)`
D3:	`sum(<k,v> in e1) e2 * e3`	if $k,v \notin FV(e3)$	↔	`(sum(<k,v> &in e1) e2) * e3`
D4:	`sum(<k,v> in e1) { e2 -> e3 }`	if $k,v \notin FV(e2)$	↔	`{ e2 -> sum(<k,v> in e1) e3 }`

Fusion Rules:				

F1:	`sum(<k,v> in e1)`
	`  if(k == e2) then`
	`  e3`
		if $k,v \notin FV(e2)$	↔	`let k = e2 in`
				`let v = e1(k) in`
				`e3`

F2:	`sum(<k1,v1> in`
	`  (sum(<k2,v2> in e1) {k2 -> e2}))`
	`e3`
			↔	`sum(<k2,v2> in e1)`
				`  let k1=k2, v1=e2 in`
				`  e3`

F3:	`sum(<k1,v1> in`
	`  (sum(<k2,v2> in e1) {@unique e2 -> e3}))`
	`e4`
			↔	`sum(<k2,v2> in e1)`
				`  let k1=e2, v1=e3 in`
				`  e4`

F4:	`sum(<k1,v1> in e1)`
	`  sum(<k2,v2> in e2)`
	`  if(v1==v2) then e3`
		if $k1,v1 \notin FV(e2)$	↔	`merge(<k1,k2,v1> in <e1,e2>)`
				`  let v2 = v1 in`
				`  e3`

Dictionary Rules:			
T1:	`sum(<k,v> in e) { k -> v }`	↔	`e`
T2:	`e2(e1) + e3(e1)`	↔	`(e2 + e3)(e1)`
T3:	`{ e1 -> e2 } + { e1 -> e3 }`	↔	`{ e1 -> e2 + e3 }`
T4:	`(e1:e2)(e3)`	↔	`if(e3 >= e1 && e3 < e2)` `  then e1 + e3`
T5:	`sum(<k,v> in e1:e2) e3`	↔	`sum(<k,_> in e1:e2)` `  let v=k+e1 in e3`

Fig. 3. Selected transformation rules of the 44 rules that form the basis of our cost-based optimizer.

logical plan. Then, we describe some refinements of the logical operators into physical operators: a query expression using the physical operators will be called *physical plan*.

5.1 The Naive Logical Plan

The first step of the optimizer consists of composing the Tensor Program with the Tensor Storage Mappings, to obtain the *Naive Logical Plan*, obtained by simply appending the TSM and the TP. More precisely, if the TP operates over tensors `A`, `B`, . . ., and each is defined by one TSM, then the naive logical plan looks like this:

```
let A = TSM-for-A
    B = TSM-for-B
    ...
in TP
```

The input to the Naive Logical Plan consists of the physical arrays, hash maps, and tries mentioned in the TSMs. Its output is the final answer of the TP.

Evaluating the naive plan directly is very inefficient, because it involves materializing all tensors in some naive representation, which is what we wanted to avoid in the first place. Instead, the system performs a sequence of *logical rewritings* in order to optimize the program.

5.2 Logical Rewritings

Our optimizer is a cost-based optimizer that applies a set of rules to find expressions equivalent to the given query, then uses a cost model to select the cheapest expression. It uses Egg [54] for simplifying a SDQLite expression using the rules. Our rule base currently consists of 44 rules of the form:

```
Pattern1  ⇝  Pattern2
```

When we assert rules in both directions then we write:

```
Pattern1  ↔  Pattern2
```

We show a few selected rules in Fig. 3. We start by showing some simple associativity and commutativity rules, followed by algebraic simplifications rules, which are unidirectional. The factorization rules allow us to move constant factors in or out of the summation; as we explain in Sec. 6, this leads to some significant performance improvements. The next group contains loop fusion rules, which are known to be of key importance for linear algebra or tensor algebra [10]. Finally, the dictionary rules capture the way that summation interacts with dictionaries.

Example 5.1. For a simple illustration, we show how to convert an iteration into a lookup. Consider the following inner product of two vectors:

```
sum (<i,a> in A, <i,b> in B) { () -> a*b }
```

After desugaring the query becomes:

```
sum (<i,a> in A) sum (<j,b> in B) if (i==j) { () -> a*B }
```

At this point the optimizer can apply fusion rule `F1` and rewrite the query to:

```
sum (<i,a> in A) let k=j, v=b(k) in a*v
```

When `A` is sparse and `B` is stored as a hash map, then this expression is much more efficient, because it iterates only over the non-zero elements of `A`, and uses a lookup to retrieve the values of `B`.

Unique Constraint To increase the power of our optimizer, we have extended SDQLite with a constraint called `@unique`, which may be specified in a dictionary construction:

```
{ @unique k -> e }
```

The semantics of `@unique` is that, in a **`sum`**, all keys are asserted to be distinct. Fusion rule `F3` requires the `@unique` constraint, and allows two nested loops to be fused into a single loop. The role of `@unique` is only to inform the optimizer: it has no effect at runtime.

We explain now the rule `F3`. Consider the following subexpression of the LHS of the rule:

```
sum(<k2,v2> in e1) {@unique e2 -> e3}
```

Suppose that `e` is a dictionary with `n` elements. Then the meaning of the **`sum`** is the summation of `n` terms:

```
{ @unique e2_1 -> e3_1 } + { @unique e2_2 -> e3_2 } + ...
```

and the `@unique` constraints guarantees that their keys `e2_1`, `e2_2`, `e2_3`, ... are distinct. Then, the outer sum will bind the variables `k1,v1` to exactly one pair `e2_i,e3_i`. Rule `F3` fuses the two loops into a single loop, and uses a **`let`** construct to bind `k1,v1` to `e2,e3`.

In some cases the `@unique` constraint can be inferred from the query, but in most cases it is data dependent, and must be asserted by data administrator when defining the TSM. For example, the TSM for the CSR representation of the tensor `C` in Fig. 1 (c) should be written as follows:

```
CREATE TENSOR C AS sum(<row,_> in 0:C_len1)
    { @unique row ->
      sum(<off,col> in C_idx2(C_pos2(row):C_pos2(row+1)))
        { @unique col -> C_val(off) }
    }
```

The expression `sum(... <(k,j), C_v> in C ...) ...` in Fig. 1 (a) desugars into two nested iterations, one for `k` and one for `j`, and the optimizer can now use rule `F3` twice to fuse these two iteration in TP with the two iterations in the TSM, leading to the much more efficient program in Fig. 1 (d).

5.3 Rule Engine: Egg

The rule engine takes as input the naive tensor plan and the collection of rules, and repeatedly applies the rules in order to obtain all equivalent query plans. This task is non-trivial: the rule engine needs to memorize all generated plans and check for duplicates, and also needs to avoid running into an infinite loop. Every cost-based query optimizer that we are aware of implements its own rule engine, which does pattern matching, duplicate detection, and memoization of expressions.

Instead of implementing our own expression manager, we adopt a state-of-the-art rewriting system called an Equality Saturation (EQSAT) system [49]. Specifically, we used Egg [54]. An EQSAT system has access to a collection of rewrite rules, and receives as input an expression `e`. It then constructs the plan space by maintaining a data structure, called an e-graph, that compactly represents a set of expressions, together with an equivalence relation over this set that can be derived from the rules. The e-graph consists of a set of e-classes, each e-class consists of a set of e-nodes, and each e-node is a function symbol with e-classes as children.

For example, Fig. 4 shows the compact representation of all expressions equivalent to `a * {k -> b+c}`. The top e-class has 3 operators. The first is `->` and has children `k` and the e-class for `a * (b+c)`, which corresponds to the associativity rule `A3`. The second is a `*` and its children are `a` and the e-class for `{ k -> b + c }`, representing the original input. The third is another `*`, with `{ k -> a }` and `b+c` as its children, and corresponds to the associativity rule `A2`. This e-graph corresponds to the following equivalent expressions:

```
{ k -> a * (b+c) } = a * { k -> b+c } = { k -> a } * (b+c)
```

This e-graph is obtained by only applying the associativity rules and contains 11 nodes and 9 e-classes. By applying the rest of transformation rules (e.g., distributivity), we obtain a more complicated e-graph with 28 nodes and 15 classes.

The e-graph (i.e., plan space) is iteratively expanded by applying all the provided rewrite rules. This process is continued until either the e-graph is *saturated* (i.e., applying rewrite rules does not change the e-graph) or a threshold (e.g., number of iterations or timeout) is reached. Finally, Egg performs the search for the best plan through the *extraction* procedure by a user-provided cost model.

5.4 Managing Free Variables

A major challenge for our cost-based optimizer is that, unlike the traditional Cascades based framework [19], our rules operate on a *calculus* instead of an *algebra*. This creates significant challenges for managing the free variables in the expressions.

For example, consider a `let`-rule like this:

```
let x = e1 in e2   ⇝   e2[e1/x]
```

There are two important challenges here. First, this rule must match with its α-equivalent terms, i.e., terms that become equivalent by substituting their variable names such as `let x = e1 in x * 2` and

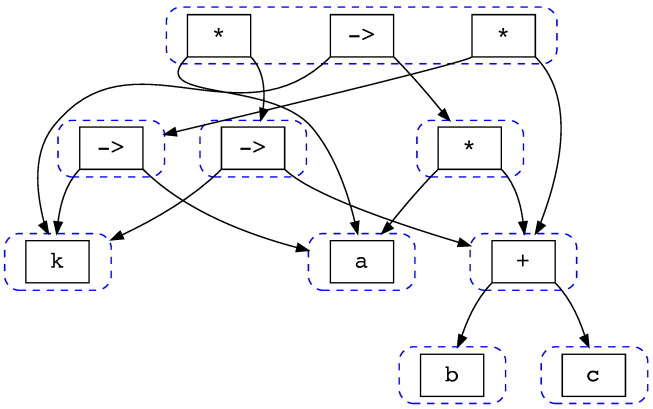

Fig. 4. The e-graph of `a * { k -> b + c }`.

`let y = e1 in y * 2`. Keeping track of α-equivalence requires a sophisticated matching strategy that imposes scalability challenges for equality saturation. Second, `e2[e1/x]`, which represents the result of substituting `x` by `e1` in `e2`, is not a valid pattern in Egg. Egg uses a compact representation of equivalent expressions, which makes it impossible to express substitution, since different equivalent representations of `e2` may or may not have `x` as a free variable, or `x` may mean different things.

We use De Bruijn indexing [17] to provide a nameless representation for variables. This way, the term `let x = e1 in x * 2` is represented as `let e1 in %0 * 2`, where `%0` refers to the variable introduced by the closest `let`-binding. It was shown [29] that De Bruijn indexing can solve the scalability of equality saturation by avoiding the e-graph to be overloaded with α-equivalent terms.

5.5 Cardinality Estimation

After applying all rules, STOREL uses a cardinality and cost estimator to select the best rewriting. We adopt ideas from [28] to represent cardinalities of nested dictionaries. A *cardinality expression* is given by the following grammar, where s is a symbol that means that the quantity is a scalar (e.g. has size 1), n is a real number, and $\#m$ represents a scalar expression that stores the size m:

$$c ::= s | n[c] | \#m$$

For example, if `A` is a dictionary of type $[n_1) \to [n_2) \to [n_3) \to \mathbb{R}$, then we may estimate its cardinality as $100[10[50[s]]]$, which means: for an estimated 100 indices `i`, `A(i)` is non-zero; for each such `i`, for an estimated 10 `j`'s, `A(i)(j)` is non-zero, and for each of these, for an estimated 50 `k`'s, `A(i)(j)(k)` is non-zero.

We use the rules in Fig. 5 to estimate the cardinality of a SDQLite expression. For example, consider the cardinality of the expressions:

```
sum (<i,v> in A) if (v==25) then {i -> i*3}
```

and assume that the cardinality of `A` is $1000[s]$. Further assume that the selectivity of the predicate is `sel(v==25) = 0.02`. Then:

$$\begin{aligned} &card(\texttt{\{i -> i*3\}}) && = 1[s] \\ &card(\texttt{if (v==25) then \{i -> i*3\}}) && = 0.02 * 1[s] = 0.02[s] \\ &card(\texttt{sum(<i,v> in A) if ...}) && = 1000 * 0.02[s] = 20[s] \end{aligned}$$

For the cardinality of input tensors (e.g., $card(\texttt{A})$) and the selectivity estimates (e.g., $sel(\texttt{e1})$), STOREL currently relies on the information provided by DBAs or uses constants (e.g., 0.1 for

$$
\begin{array}{rcll}
elem(\texttt{e}) & = & c & \text{if } card(\texttt{e}) = n[c] \\
size(\texttt{e}) & = & n & \text{if } card(\texttt{e}) = n[c] \\
\hline
card(\texttt{e1(e2)}) & = & elem(\texttt{e1}) & \\
card(\texttt{\{ e1 -> e2 \}}) & = & 1[card(\texttt{e2})] & \\
card(\texttt{0:e1}) & = & m[s] & \text{if } card(\texttt{e1}) = \#m \\
card(\texttt{let x = e1 in e2}) & = & card(\texttt{e2}) & \\
card(\texttt{if(e1) then e2}) & = & \begin{cases} s & \text{if } card(\texttt{e2}) = s \\ sel(\texttt{e1}) \cdot n[c] & \text{if } card(\texttt{e2}) = n[c] \end{cases} & \\
card(\texttt{sum(<k,v> in e1) e2}) & = & \begin{cases} s & \text{if } card(\texttt{e2}) = s \\ size(\texttt{e1}) \cdot n[c] & \text{if } card(\texttt{e2}) = n[c] \end{cases} &
\end{array}
$$

Fig. 5. Cardinality estimation rules.

$$
\begin{aligned}
cost(\texttt{e1(e2)}) &= cost(\texttt{e1}) + cost(\texttt{e2}) + \gamma_{lookup}(\texttt{e1}) \\
cost(\texttt{\{ e1 -> e2 \}}) &= \infty \\
cost(\texttt{\{@dense e1 -> e2 \}}) &= cost(\texttt{e1}) + cost(\texttt{e2}) + \gamma_{arr-insert}(\texttt{e1}, \texttt{e2}) \\
cost(\texttt{\{@hash e1 -> e2 \}}) &= cost(\texttt{e1}) + cost(\texttt{e2}) + \gamma_{hash-insert}(\texttt{e1}, \texttt{e2}) \\
cost(\texttt{let x = e1 in e2}) &= \gamma_{mater}(\texttt{e1}) \cdot cost(\texttt{e1}) + cost(\texttt{e2}) \\
cost(\texttt{if(e1) then e2}) &= cost(\texttt{e1}) + sel(\texttt{e1}) \cdot cost(\texttt{e2}) \\
cost(\texttt{sum(<k,v> in e1) e2}) &= cost(\texttt{e1}) + \gamma_{iter}(\texttt{e1}) \cdot size(\texttt{e1}) \cdot cost(\texttt{e2}) \\
cost(\texttt{merge(<k1,k2,v> in <e1,e2>) e3}) &= \\
cost(\texttt{e1}) + cost(\texttt{e2}) &+ (\gamma_{iter}(\texttt{e1}) \cdot size(\texttt{e1}) + \gamma_{iter}(\texttt{e2}) \cdot size(\texttt{e2})) \cdot cost(\texttt{e3})
\end{aligned}
$$

Fig. 6. Cost estimation rules.

selectivity estimates). We leave the usage of histograms and more advanced cardinality estimation techniques for the future.

5.6 Physical Plans

So far all expressions in SDQLite are logical plans. We describe here how we convert SDQLite expressions into physical plans, which we execute on our runtime system, Julia. Simple scalar operators like `a+b` or `a*b` get converted immediately into physical operations. Julia also supports plus and times operators on dictionaries (tensors); if that were not the case, then we can force the optimizer to write such operations explicitly as loops, e.g. we rewrite the expression `a*b`, where `a` is a scalar and `b` is a dictionary, into:

```
sum (<i,vb> in b) { i -> a*vb }
```

and assign a cost of ∞ to `+` and `*` operators applied to dictionaries.

The physical operator associated to `sum (<k,v> in e1) e2` is a `for` loop iterating over the dictionary `e1`. To make this loop concrete, STOREL needs to know how the dictionary `e1` is represented. In our system, there are two choices: as a dense vector, or as a hash-map. STOREL knows the type of storage for the arrays, hash maps, and tries of the physical storage, since they were explicitly declared in the TSM. For all other constructed dictionaries, STOREL needs to choose whether to construct a dense vector, or a hash map. We do this by adding the following two rules to the collection of rules:

```
{k -> e}   ↝   { @dense k -> e}
{k -> e}   ↝   { @hash k -> e}
```

In the first rule `{k -> e}` becomes an entry of a dense array, in the second rule it becomes an entry of a hash-map. We assign a cost of ∞ to any expression that still contains a logical dictionary `{k -> e}`, thus forcing the optimizer to choose either a dense array or a hash-map representation.

Tensor	Dimensions	Density	# non-zeros
cant	62K × 62K	1×10^{-3}	2.03M
consph	83K × 83K	9×10^{-4}	3.05M
cop20k_A	121K × 121K	2×10^{-4}	1.36M
pdb1HYS	36K × 36K	3×10^{-3}	2.19M
rma10	46K × 46K	1×10^{-3}	2.37M
webbase	1M × 1M	3×10^{-6}	3.11M
NIPS	2.4K × 2.8K × 14K	3×10^{-5}	31.31M
NELL	12K × 9.2K × 29K	2×10^{-5}	76.88M
Facebook	1.6K × 64K × 64K	1×10^{-7}	0.74 M
Enron	6K × 5.7K × 244K	3×10^{-6}	3.10 M

Table 2. Real-world matrices and rank-3 tensors used in the experiments.

Finally, we add one additional physical operator to SDQLite:

```
merge(<k1,k2,v> in <e1,e2>) e3
```

Both `e1` and `e2` must be dictionaries of real values, in other words they must be vectors, and, in that case, the semantics of **merge** is:

```
sum(<k1,v> in e1, <k2,u> in e2) (if (v==u) then e3)
```

This is captured by the Fusion Rule `F4` in Fig. 3.

5.7 Cost Estimate

Finally, the cost of a physical plan is estimated using the rules shown in Fig.6. These inference rules include parameters that are dependent on the type of the underlying collection (e.g., γ_{lookup} and γ_{iter} for a dense-array is smaller than the one for a hash-map). We notice that a logical plan for which we have not chosen between a dense array and hash map will have cost ∞.

6 EXPERIMENTS

In this section we present an empirical evaluation of STOREL, by running on several common tensor kernels, with a variety of real and synthetic matrices and tensors, and comparing it with six other systems. We studied the following questions:

(1) How much do tensor programs over flexible storage benefit from cost-based optimization?
(2) How do different choices of storage formats for different data sparsities affect the run-time performance, and does STOREL take best advantage of the given storage format?
(3) How much do specific sets of rewrite rules contribute to the optimization? In particular we would like to understand the contribution of loop fusion and factorization.
(4) How complex is the optimization task? How many applications of rules are needed to optimize Tensor Programs?
(5) How practical is the optimization process? Does the run time improvement outweight the optimization overhead?

In addition, we discuss our experience with using Egg as our rule rewrite system at the end of the section.

Experimental Setup. We conducted our experiments on an AWS t2.2xlarge instance with 8 vCPUs, 32 GBs of RAM, and Ubuntu 22.04 LTS. Our system uses Julia 1.7.3 [8] for executing the generated code. The other systems we benchmark are Taco [27], NumPy 1.22.3 [22], SciPy 1.8.1 [52], PyTorch 1.11.0, TensorFlow 2.9.1, and DuckDB 0.3.2 [37]. We use G++ 11.2.0 to compile the generated C code in Taco, and use Python 3.10.4 for NumPy, SciPy, PyTorch, and TensorFlow. In DuckDB, all tensors are encoded as relations, which are comparable to the coordinate (COO) format in tensor systems, and we provide all relevant indices. In addition, we use an in-memory database and the

Program		Sec. 6.1 Dimension	Sec. 6.2 Dimension
MMM	$Q(i,j) = \sum_k A(i,k) \cdot B(k,j)$	B: _ × 250	A:$10^3 \times 10^3$,B:_ × 10^3
ΣMMM	$Q() = \sum_{i,j,k} A(i,k) \cdot B(k,j)$	B: _ × 250	A:$10^5 \times 10^5$,B:_ × 10^5
BATAX	$Q(j) = \sum_{i,k} \beta \cdot A(i,j) \cdot A(i,k) \cdot X(k)$	X: _	A:$10^5 \times 10^5$,X:_
TTM	$Q(i,j,k) = \sum_l A(i,j,l) \cdot B(k,l)$	B: _ × 25	—
MTTKRP	$Q(i,j) = \sum_{k,l} A(i,k,l) \cdot B(k,j) \cdot C(l,j)$	B: _ × 25	—

Program	STOREL / Taco	SciPy	NumPy	PyTorch	TensorFlow	DuckDB
MMM	CSR, CSR	CSR, CSR	Dense, Dense	CSR, Dense	COO, Dense	COO, COO
ΣMMM	CSC, CSR	CSR, CSR	Dense, Dense	CSR, Dense	COO, Dense	COO, COO
BATAX	CSR, Dense	CSR, Dense	Dense, Dense	CSR, Dense	COO, Dense	COO, COO
TTM	CSF, CSC/CSR	—	—	—	—	COO, COO
MTTKRP	CSF, CSR, CSC	—	—	—	—	COO, COO, COO

Table 3. Tensor Programs, the dimensions used for experiments in Sec. 6.1 and 6.2, and the best storage formats for each system. SciPy, NumPy, PyTorch, and TensorFlow do not support higher-order sparse tensors. DuckDB encodes the tensors as relations, which are comparable to the coordinate (COO) format in tensor systems. The missing dimensions, denoted by _ or not included, can be inferred from the context (e.g., for Sec. 6.1 the dimension of A is specified in Table 2 and the number of rows of B is the same as the number of columns of A).

Python API to interact with DuckDB. Python-based frameworks do not support sparse tensors with more than two dimensions, so we only report the times for kernels that only contain matrices and vectors. In addition, NumPy only supports dense storage formats.

All experiments are run on one CPU core, and we report the average execution time of five runs. In all cases we measure only the execution time (which includes the assembly time for Taco), and we exclude from the run time the creation and indexing of storage format, loading time, compilation time, and optimization times respectively, for the systems that have these components.

Datasets. We use both real world and synthetic datasets. For the former, we collected six sparse matrices from the SuiteSparse Matrix Collection [16], and four rank-3 tensors from the FROSTT Tensor Collection [45]. Table 2 presents a summary of these datasets. For synthetic data, we generate random matrices and vectors with specified sparsity and dimensions.

Workloads. Table 3 presents the tensor programs we consider in this evaluation. *MMM* stands for matrix-matrix multiplication; *ΣMMM* computes the summation over a matrix-matrix multiplication; *BATAX* was previously studied in [34]; *TTM* computes the tensor times matrix multiplication; and *MTTKRP* stands for matricized tensor times Khatri-Rao product. Both *TTM* and *MTTKRP* have been studied extensively in papers on Taco (see e.g., [14, 27]).

6.1 Benchmarking Tensor Programs

Here we addressed the first question: how much do tensor programs over flexible storage benefit from the application of rewrite rules? We present the benchmark of STOREL, Taco, SciPy, NumPy, PyTorch, TensorFlow, and DuckDB on all considered tensor programs.

Storage Formats. Table 3 presents the best storage formats we found for each considered tensor program and system. For each experiment, the A matrix or tensor in the respective kernel is defined by one of the datasets in Table 2. All other matrices are synthetically generated with sparsity 2^{-5}.

DuckDB and NumPy only support a single storage format, i.e., relations and respectively dense matrices/vectors. For that reason, we only consider them for the kernels that operate on matrices

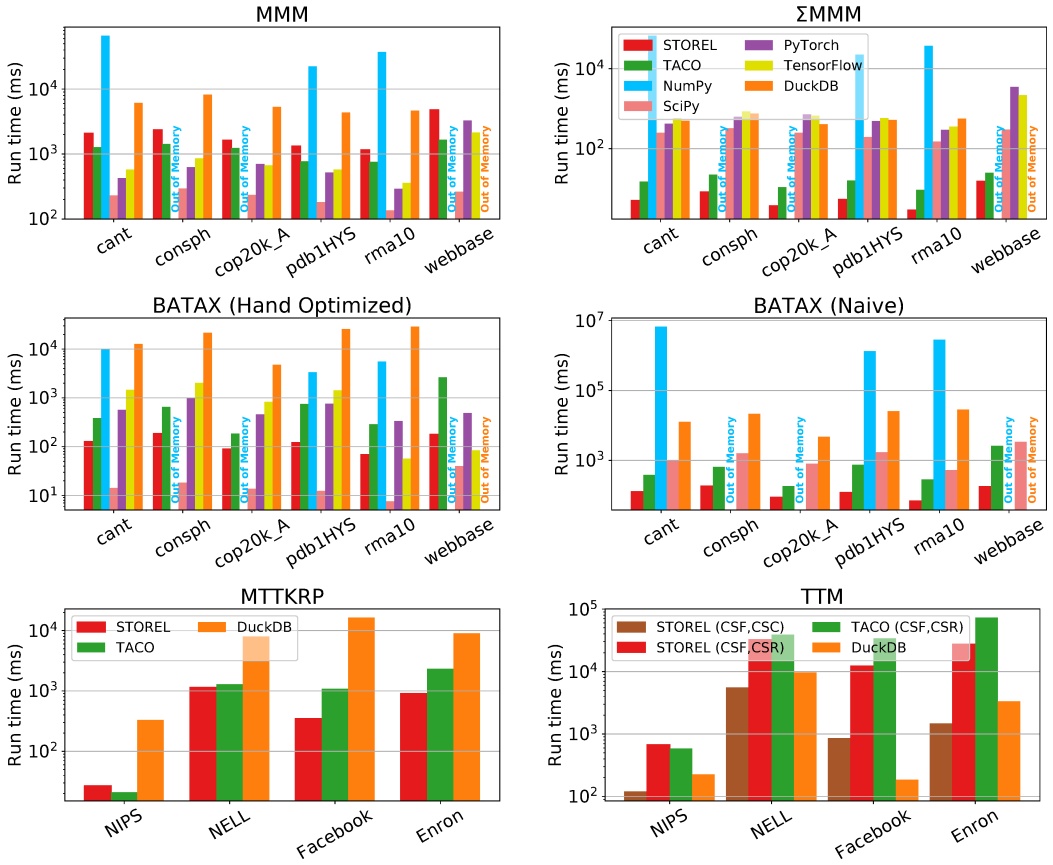

Fig. 7. End-to-end run time (in milliseconds) for STOREL, Taco, SciPy, NumPy, PyTorch, TensorFlow, and DuckDB for different kernels and real-world matrices and tensors.

and vectors. For SciPy and PyTorch we use the CSR format for all matrices, because our experiments with CSC matrices were consistently slower. For *TTM*, we report the performance for two storage formats in STOREL. The first uses a CSF tensor and a CSC matrix, which is the optimal storage specification for this kernel. Taco, however, fails to compile the kernel with the CSC matrix, which we reported to the Taco developers. Thus, we also report the performance for a CSF tensor and a CSR matrix for a direct comparison of Taco and STOREL. Finally, PyTorch and TensorFlow have a limited support for sparse matrix operations[3]. Thus, we include the results for a hand-optimized plan for the *BATAX* kernel.

Results. Figure 7 presents our run-time benchmarks for the above workloads. STOREL is always at least competitive with Taco, and achieves significant performance improvements for kernels that benefit from our factorization rewrite rules. This is the case for the *ΣMMM*, *BATAX*, and *MTTKRP* kernels. For instance, STOREL can compute *BATAX* up to 16.4× faster than Taco for webbase. Thus, our rewrite rules can lead to significant performance improvements for a variety of tensor programs.

The *MMM* benchmark is a simple matrix multiplication and offers almost no opportunity for optimization, but instead is a good benchmark for comparing the physical runtimes of the systems. SciPy has the best run time of all, while those of STOREL and Taco are comparable. SciPy has a

[3]PyTorch and TensorFlow only support a sparse-dense matrix multiplication.

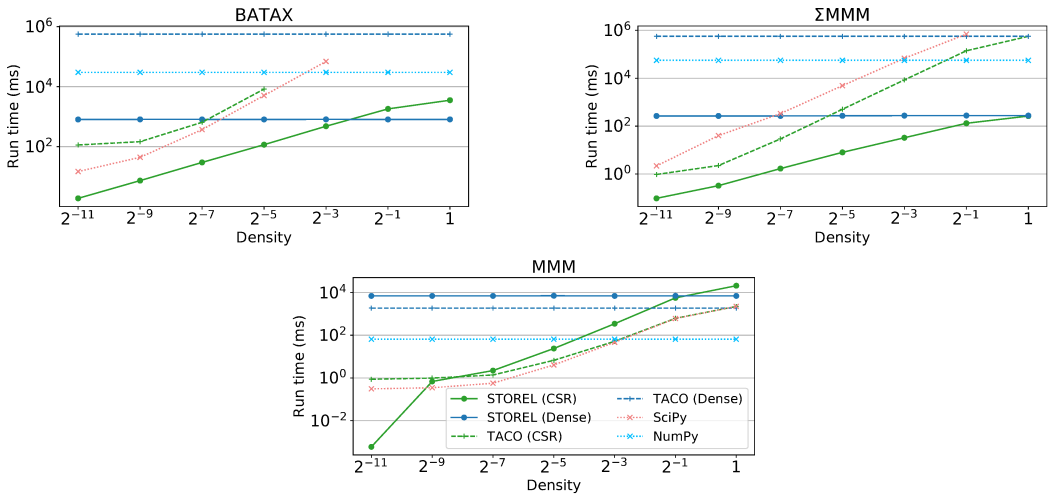

Fig. 8. Runtime of STOREL, Taco, SciPy, and NumPy for varying sparsity using sparse and dense storage formats.

suite of highly optimized low-level primitives, like sparse-sparse matrix multiplication. PyTorch and TensorFlow, however, support sparse-dense matrix multiplication, and thus show a worse performance for *MMM*. These frameworks require the composition of such primitives with costly materialization of intermediate results for the other benchmarks. We observe that STOREL can be up to two orders of magnitude faster than them when high-level optimizations are possible. However, for hand-optimized plans (e.g., *BATAX*) the highly optimized primitives show better performance than the Julia-based runtime of STOREL. NumPy requires all inputs to be dense, and runs out of memory for all but four experiments, where STOREL outperforms it by two orders of magnitude. This exemplifies the importance of flexible storage.

DuckDB uses quite different physical operators, and a direct comparison of the wall clock time is not very informative. We observe, however, that DuckDB is remarkably efficient for the kernels that do not offer opportunities for cost-based optimization. For instance, DuckDB has excellent performance for the *TTM* kernel, which translates into a simple aggregate-join query. In contrast, DuckDB is significantly slower for the Σ*MMM*, *BATAX* and *MTTKRP* kernels. For Σ*MMM*, this is because DuckDB does not push the summation past the join. For *BATAX* and *MTTKRP* kernels, DuckDB is not able to factorize the computation, and uses binary join plans which construct costly intermediate results.

6.2 Effect of the Storage Mapping

Next, we turn our attention to the second question: do different choices of storage format for different data sparsities affect the run-time performance, and does STOREL take best advantage of the given storage format?

We consider the *BATAX*, Σ*MMM*, and *MMM* kernels and present the run time for STOREL, Taco, NumPy, and SciPy for different sparsity factors in the input matrices. For STOREL and Taco, we further consider both the sparse storage format as in Sec. 6.1, as well as the fully dense storage format. We only use synthetic datasets for this benchmark. For Σ*MMM* and *MMM*, we vary the sparsity in both matrices, and we use the same sparsity factor for both. For *BATAX*, we consider the naive plan and only the matrix varies the sparsity, while the vector remains dense.

The results are presented in Figure 8. We observe that STOREL adapts to the given storage format: the sparse variant is more efficient in most cases, at high densities the dense format

becomes more efficient, as expected. Note that for Σ*MMM* and *BATAX* STOREL outperforms all other systems independent of the sparsity, due to the factorization rules. However, for *MMM* the low-level primitives of NumPy and SciPy outperform the nested loops generated by STOREL and Taco. As an example, for higher densities NumPy outperforms all competitors, thanks to the heavily-tuned low-level primitive provided by BLAS. We leave the synthesis of such primitives (e.g., BLAS routines), instead of the nested loops, for future.

6.3 Effect of Rewrite Rules

Here we address the third question: study the contribution of two classes of rewrite rules, loop fusion and factorization, on the overall optimization. For that purpose we use the *BATAX* kernel as an example. The results are presented in Figure 9 (left).

We first consider the case where the input matrix is a nested hash-map (trie), in which case we only benefit from the factorization rules. The following expression presents the unoptimized program, which we use as the baseline (the green line in Figure 9 (left)):

```
sum(<i, Ai> in A)
  sum(<j, Aij> in Ai)
    sum(<k, Aik> in Ai)
      { j -> beta * Aij * Aik * x(k) }
```

This kernel has two factorization opportunities. The first rewriting hoists the construction of the dictionary with key j out of the inner sum:

```
sum(<i, Ai> in A)
  sum(<j, Aij> in Ai)
  { j -> sum(<k, Aik> in Ai)
    beta * Aij * Aik * x(k) }
```

The rewritten kernel, represented by the blue line, is between one to two orders of magnitude faster than the non-optimized kernel, depending on the sparsity.

The second factorization opportunity hoists the inner sum over k outside the sum over j:

```
sum(<i, Ai> in A)
  let t = (sum(<k, Aik> in Ai) Aik * x(k))
  in (sum(<j, Aij> in Ai) { j -> beta * Aij * t})
```

This optimization can further improve the run time by an order of magnitude. For very sparse data, however, it is more beneficial to avoid hoisting the loop outside. This is because the inner sum may not be executed at all for many i values.

We further consider the case where the matrix is stored with a CSR format, in which case there is a fusion opportunity. The two dashed lines in Figure 9 (left) represent the run time of STOREL with and without the fusion of the CSR matrix, while at the same time exploiting both factorization opportunities as described above. We observe that the unfused variant comes with heavy overhead, because the program first materialized the matrix that is defined by the storage representation and then executes the program. This would be 2× worse than the non-optimized baseline, despite the use of factorization. It is only with the fusion of the storage representation and the actual program that we achieve the best performance, which is 3× faster than the optimized hash-based implementation.

6.4 Cost and Complexity of the Rewrite-based Optimization

Finally, we address here the fourth question: what is the cost and the complexity of the cost-based optimization? Recall that we have not included the optimization cost in our experiments so far.

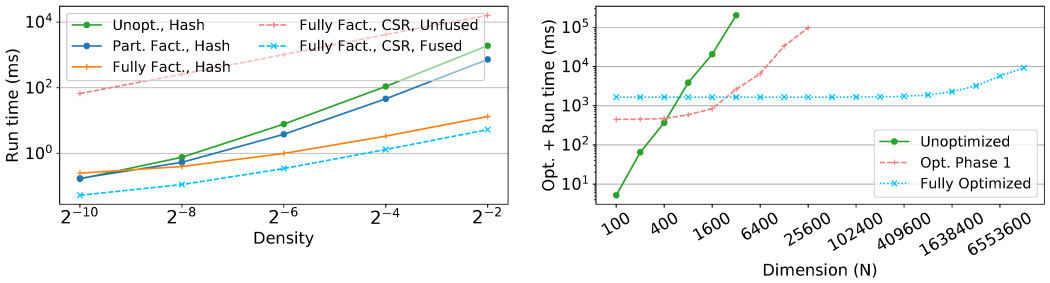

Fig. 9. (left) Impact of factorization and fusion rules on the *BATAX* kernel. The dimension of matrix A is $10^3 \times 10^3$. (right) The total execution time of different versions of the *BATAX* kernel. The dimension of matrix A is $10^2 \times N$.

Our rewrite rules define a huge search space, and it proved to be too large for the current version of Egg to saturate. Our solution was to restrict the search space by splitting our optimization pipeline into two stages. First, we apply our rewrite rules to the tensor program without taking the storage format into account. Then, we further optimize the resulting program in conjunction with the provided storage format. We notice that most Cascade-style optimizers also partition the optimization into several stages, in order to reduce the search space and make the optimization possible.

Table 4 presents the key metrics for the two optimization passes in Egg per tensor program. We observe that, even with the separation of the optimization pipeline, Egg explores a large search space and constructs an e-graph with tens of thousands of equality classes. As a result, the optimization time can take up to 1.7 seconds in total, which is longer than the execution time of the kernel for small tensors. In the next section, we investigate in more detail the trade-off between optimization and run time.

6.5 Optimization Overhead

In order to better demonstrate the practicality of the optimization process, we compare the run time and optimization time with the following coarse-grained rewrites: (1) storage-independent optimizations, and (2) optimizations that take storage into account. These two coincide with the optimization stages reported earlier. As the tensor program, we consider the *BATAX* kernel because (1) it has the longest optimization overhead, and (2) it largely benefits from the two stages of optimization.

Figure 9 (right) shows the total execution time of the *BATAX* kernel, including the optimization overhead, by varying the dimension, for which we considered a time out of five minutes. We observe that although for smaller matrices the unoptimized program is performing better, for larger matrices the optimization overhead is amortized by the improved run time. The 1.7 seconds spent for the fully optimized kernel are well justified. They enable the system to scale to matrices that are three orders of magnitude larger than those supported by the kernel with only storage-independent optimizations. Note that, while the optimization time is high, it needs to be compared to compilers for tensor systems, which typically take much longer. For instance, the BTO compiler can take several minutes to find the optimal execution plan [34]. In addition, Egg has been shown to outperform alternative approaches, such as using SMT solvers [54]. In the next section, we discuss the implications of these results.

Kernel	Time (ms)	Iters.	Nodes	Classes	Memos
BATAX	445	31	47441	30810	51508
	1212	59	46456	8043	59010
ΣMMM	1	6	42	25	42
	52	22	2077	530	2698
MTTKRP	10	18	571	135	821
	239	35	8414	1130	10700
MMM	10	11	910	123	1242
	1708	61	33058	6479	43407
TTM	11	12	1173	140	1480
	891	61	15891	3244	23981

Table 4. Compilation metrics reported by Egg.

6.6 Discussion

The use of the Equality Saturation System Egg was of great help for us. Egg supports the entire functionality needed for a rule engine, it is an open source system, and it has been developed on solid theoretical foundations [54]. Nevertheless, Egg is a research project, still under development, and has limitations that affected our system STOREL; we discuss them here.

Performance We used Egg version 0.6.0 and its optimizations adds significant overhead (c.f., Table 4). A very recent version was reported recently [57], and it improves the matching significantly by adopting a Worst Case Optimal Join [35], since pattern matching is, in essence, the same as computing a (usually cyclic) join query. That version is not yet available.

Cost computation The biggest limitation for us is the way Egg handles the cost. Egg allows the user to define a cost model, and uses this cost model to extract the cheapest expression from the root e-node. However, it does not separate between the cardinality estimate and the cost, and, worse, the cost can only be a number, while our cardinality, defined in Sec. 5, has a complex structure. For that reason we had to use hacks to approximate our cost using what is available in Egg. We were able to always extract the optimized plan for the given TSM, but were not able to compare in a meaningful way plans derived from alternative storage formats, i.e. alternative TSMs. If this was possible, the programmer could specify several alternative storage mappings for one tensor. The system would then optimize the program separately for each of them and return the cheapest plan.

Other minor limitations The inability of Equality Saturation Systems in general, and of Egg in particular, to handle expressions with variables is well known. In addition, a minor limitation is that the current version of Egg does not have a DSL for the rules, instead they need to be written in Rust. We are in contact with the Egg authors and are optimistic that Egg will continue to improve.

7 RELATED WORK

There is a vast literature on tensor and linear algebra systems in the compilers and HPC communities. Most of them focus on dense data (e.g., [13, 34, 36, 39, 46, 47]). Similarly, the database community studied Array DBMS [7, 48] and SQL extensions for matrices (e.g., [31, 41, 56]), which are also primarily designed for dense data. Both lines of work are not concerned with different tensor storage representations and thus orthogonal to this work. Packages like SciPy [52] or the MATLAB Tensor Toolbox [6] support different sparse matrix/tensor representations, but rely on composing hardcoded operations, which can become a severe bottleneck as shown in Sec. 6. The closest related work from the tensor systems literature is the Taco system [14, 26, 27], as highlighted in Sec. 1.

We drew many inspirations from the database literature. At the top is the classic work on GMAP [50], which pioneered the idea of using a declarative language for representing physical data layout: for example, a secondary index can be described as a view obtained by projecting the

relation on the indexed attribute and the primary key. GMAP uses Local As View (LAV), while our Tensor Storage Mappings are defined as Global As View (GAV) [21]. More recently the Hadad system [4] has applied a similar high level principle for hybrid RA/LA analytics. Hadad uses integrity constraints to express relationships between hybrid data sources, and uses *chase* to optimize a query given those relationships. The *chase* applies to relational queries, over the Boolean domain, and does not extend to queries over semirings, thus, it was not an option for our system. The SPORES system [53] describes an optimizer for linear algebra, in the context of SystemML [10]. The key approach in SPORES is to convert every query into a normal form, which is a sum of sum-of-products, i.e. similar to Unions of Conjunctive Queries. This is not possible in SDQLite, which we designed specifically to cope with complex storage formats. For example see the two quite different expressions for matrix multiplication in Example 3.1: there is no unique normal form for that query.

Our work is also related to factorized learning, a line of work that uses database optimizations to improve the performance of machine learning tasks [11, 25, 30, 40, 43, 44]. Factorized learning, however, optimizes for normalized relational data; whereas we optimize for dense and sparse tensor representations. Normalized schemas are very different from COO/CSC/CSR/CSF representations. The SystemML optimizer [10] has demonstrated the usefulness of loop fusion, an optimization that we capture with rule Rule F4 in Fig. 3. Our optimizer is closest in spirit to SPORES [53], which optimizes linear algebra expressions by first converting them to relational algebra, optimizing these, then converting back to linear algebra. The SPORES optimizer relies on the fact that the queries in that system have a unique normal form (since they are, essentially, UCQs). Our optimization task is harder, because queries in SDQLite do not have a unique normal form, for example, consider the two matrix multiplication expressions in Example 3.1, none of which can be considered to be the "normal form" of the other.

At the time of writing, TensorFlow develops a graph optimization system called Grappler [1]; it is currently restricted to dense tensors, and is heuristic-based, while our system is cost-based. A heuristic-based optimizer could, for example, prefer some physical plan when the tensors are dense, and another plan when they are sparse; in contrast, our cost-based optimizer can consider combinations of sparse and dense tensors and choose the most appropriate plan using a cost model.

8 CONCLUSIONS

We have described STOREL, which, to the best of our knowledge, is the first system to use a cost-based optimizer to optimize tensor programs over flexible storage. The key contributions are the use of a common declarative language for both the Tensor Program and the Tensor Storage Mappings, and a cost-based optimizer that can take advantage of rich storage formats. We have shown experimentally that the rule based optimizer can lead to performance improvements over other systems. In future work, we plan to extend STOREL to automatically choose between different storage formats. We also plan to integrate a scheduler, which is inspired by the significant progress in automating the scheduler in Halide [3, 33].

While our ultimate goal is to optimize entire ML pipelines, extending the current optimizer to large tensor programs will require significant engineering effort, and may also requires future research on how to propagate sparsity information of intermediate results.

ACKNOWLEDGEMENT

The authors would like to thank Remy Wang for his help with the Egg framework. Shaikhha would like to thank Huawei for their support of the distributed data management and processing laboratory at the University of Edinburgh. Suciu was partially supported by NSF IIS 1907997 and NSF-BSF 2109922. This project was partially supported by RelationalAI.

REFERENCES

[1] 2022. TensorFlow graph optimization with Grappler. https://www.tensorflow.org/guide/graph_optimization. Accessed: 2022-06-30.

[2] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Found. Trends Databases* 5, 3 (2013), 197–280. https://doi.org/10.1561/1900000024

[3] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–12.

[4] Rana Alotaibi, Bogdan Cautis, Alin Deutsch, and Ioana Manolescu. 2021. *HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries.* Association for Computing Machinery, New York, NY, USA, 23–35. https://doi.org/10.1145/3448016.3457311

[5] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Jim Gray, W. Frank King III, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Mario Schkolnick, Patricia G. Selinger, Donald R. Slutz, H. Raymond Strong, Paolo Tiberio, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. 1979. System R: A Relational Data Base Management System. *Computer* 12, 5 (1979), 42–48. https://doi.org/10.1109/MC.1979.1658743

[6] Brett W Bader and Tamara G Kolda. 2008. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing* 30, 1 (2008), 205–231.

[7] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. 1998. The multidimensional database system RasDaMan. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data.* 575–577.

[8] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

[9] Mark Blacher, Joachim Giesen, Sören Laue, Julien Klaus, and Vikor Leis. 2022. Machine Learning, Linear Algebra, and More: Is SQL All You Need?. In *CIDR.*

[10] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. 2018. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *Proc. VLDB Endow.* 11, 12 (2018), 1755–1768. https://doi.org/10.14778/3229863.3229865

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

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[13] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. 2012. Diderot: A Parallel DSL for Image Analysis and Visualization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI'12).* ACM, 111–120.

[14] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276493

[15] The SciPy community. 2022. scipy.sparse.dok_matrix – SciPy v1.8.0 Reference Guide. https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.dok_matrix.html. Accessed: 2022-04-14.

[16] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[17] Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, Vol. 75. Elsevier, 381–392.

[18] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proc. VLDB Endow.* 13, 11 (2020), 1891–1904. http://www.vldb.org/pvldb/vol13/p1891-freitag.pdf

[19] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29. http://sites.computer.org/debull/95SEP-CD.pdf

[20] Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.* 31–40.

[21] Alon Y. Halevy. 2001. Answering queries using views: A survey. *VLDB J.* 10, 4 (2001), 270–294. https://doi.org/10.1007/s007780100054

[22] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[23] Dylan Hutchison, Bill Howe, and Dan Suciu. 2017. LaraDB: A Minimalist Kernel for Linear and Relational Algebra Computation. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR@SIGMOD 2017, Chicago, IL, USA, May 19, 2017*, Foto N. Afrati and Jacek Sroka (Eds.). ACM, 2:1–2:10. https://doi.org/10.1145/3070607.3070608

[24] Zhihao Jia, Oded Padon, James J. Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 47–62. https://doi.org/10.1145/3341301.3359630

[25] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Learning Models over Relational Data Using Sparse Tensors and Functional Dependencies. *ACM Trans. Database Syst.* 45, 2 (2020), 7:1–7:66. https://doi.org/10.1145/3375661

[26] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor algebra compilation with workspaces. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 180–192.

[27] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

[28] Yannis Klonatos, Andres Nötzli, Andrej Spielmann, Christoph Koch, and Victor Kuncak. 2013. Automatic synthesis of out-of-core algorithms. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 133–144.

[29] Thomas Koehler, Phil Trinder, and Michel Steuwer. 2021. Sketch-Guided Equality Saturation: Scaling Equality Saturation to Complex Optimizations in Languages with Bindings. *arXiv preprint arXiv:2111.13040* (2021).

[30] Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. 2015. Learning Generalized Linear Models Over Normalized Data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1969–1984. https://doi.org/10.1145/2723372.2723713

[31] Shangyu Luo, Zekai J. Gao, Michael N. Gubanov, Luis Leopoldo Perez, Dimitrije Jankov, and Christopher M. Jermaine. 2020. Scalable linear algebra on a relational database system. *Commun. ACM* 63, 8 (2020), 93–101. https://doi.org/10.1145/3405470

[32] Guido Moerkotte and Thomas Neumann. 2006. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 930–941. http://dl.acm.org/citation.cfm?id=1164207

[33] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–11.

[34] Thomas Nelson, Geoffrey Belter, Jeremy G Siek, Elizabeth Jessup, and Boyana Norris. 2015. Reliable generation of high-performance matrix algebra. *ACM Transactions on Mathematical Software (TOMS)* 41, 3 (2015), 1–27.

[35] Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2013. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.* 42, 4 (2013), 5–16. https://doi.org/10.1145/2590989.2590991

[36] Markus Püschel, José M. F. Moura, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (2005), 232–275. https://doi.org/10.1109/JPROC.2004.840306

[37] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1981–1984. https://doi.org/10.1145/3299869.3320212

[38] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2018. Halide: decoupling algorithms from schedules for high-performance image processing. *Commun. ACM* 61, 1 (2018), 106–115. https://doi.org/10.1145/3150211

[39] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[40] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. 2019. A Layered Aggregate Engine for Analytics Workloads. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1642–1659. https://doi.org/10.1145/3299869.3324961

[41] Maximilian E. Schüle, Tobias Götz, Alfons Kemper, and Thomas Neumann. 2022. ArrayQL Integration into Code-Generating Database Systems. In *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*. OpenProceedings.org, 1:40–1:51. https://doi.org/10.5441/002/edbt.2022.04

[42] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2021. Functional Collection Programming with Semi-Ring Dictionaries. *CoRR* abs/2103.06376 (2021). arXiv:2103.06376 https://arxiv.org/abs/2103.06376

[43] Amir Shaikhha, Maximilian Schleich, Alexandru Ghita, and Dan Olteanu. 2020. Multi-layer optimizations for end-to-end data analytics. In *CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, February, 2020*. ACM, 145–157. https://doi.org/10.1145/3368826.3377923

[44] Amir Shaikhha, Maximilian Schleich, and Dan Olteanu. 2021. An Intermediate Representation for Hybrid Database and Machine Learning Workloads. *Proc. VLDB Endow.* 14, 12 (2021), 2831–2834. https://doi.org/10.14778/3476311.3476356

[45] Shaden Smith, Jee W Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. FROSTT: The formidable repository of open sparse tensors and tools.

[46] Daniele G Spampinato and Markus Püschel. 2014. A basic linear algebra compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 23–32.

[47] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. ACM, New York, NY, USA, 205–217.

[48] Michael Stonebraker, Paul Brown, Donghui Zhang, and Jacek Becla. 2013. SciDB: A Database Management System for Applications with Complex Analytics. *Comput. Sci. Eng.* 15, 3 (2013), 54–62. https://doi.org/10.1109/MCSE.2013.19

[49] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 264–276.

[50] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. 1996. The GMAP: A Versatile Tool for Physical Data Independence. *VLDB J.* 5, 2 (1996), 101–118. https://doi.org/10.1007/s007780050018

[51] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy (Eds.). OpenProceedings.org, 96–106. https://doi.org/10.5441/002/icdt.2014.13

[52] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[53] Yisu Remy Wang, Shana Hutchison, Dan Suciu, Bill Howe, and Jonathan Leang. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proc. VLDB Endow.* 13, 11 (2020), 1919–1932. http://www.vldb.org/pvldb/vol13/p1919-wang.pdf

[54] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434304

[55] Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. 2021. Tensor Relational Algebra for Distributed Machine Learning System Design. *Proc. VLDB Endow.* 14, 8 (2021), 1338–1350. https://doi.org/10.14778/3457390.3457399

[56] Ying Zhang, Martin Kersten, and Stefan Manegold. 2013. SciQL: Array data processing inside an RDBMS. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1049–1052.

[57] Yihong Zhang, Yisu Remy Wang, Max Willsey, and Zachary Tatlock. 2022. Relational e-matching. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–22. https://doi.org/10.1145/3498696

Received April 2022; revised July 2022; accepted August 2022