Scalable Tensors for Big Data Analytics
(extended version)

Leonidas Fegaras, Tanvir Ahmed Khan, Md Hasanuzzaman Noor, and Tanzima Sultana

University of Texas at Arlington, CSE, Arlington, TX 76019
{fegaras,tanvirahmed.khan,mdhasanuzzaman.noor,tanzima.sultana}@uta.edu

***Abstract*—Many vectorization languages and linear algebra libraries are often implemented as light-weight wrappers around high-performance array libraries, such as BLAS, and rely on the limited array storage structures and routines provided by these libraries, which are hard to extend and customize. In this paper, we describe a customizable framework for large-scale array programs in which arrays and array operations are abstract but their implementation is guided by user-defined storage mappings. We introduce a new storage structure for arrays, called a distributed tensor, which is a distributed collection of array blocks that may have any number of sparse and dense dimensions. In addition, we present rules for translating abstract array programs to high-performance distributed code that can run on Apache Spark. The performance of our system is on par with highly optimized linear algebra libraries, thus providing implementation independence and extensibility without sacrificing performance. Finally, we justify our claims by evaluating the performance of our system relative to Spark MLlib and TensorFlow.**

I. Introduction

Arrays are prevalent in scientific computing and in data analysis and machine learning applications. Currently, most array programming is done using vectorization languages, such as NumPy, Matlab, R, and Julia, which allow programmers to write high-level array code that closely resembles mathematical formulas. Unlike general-purpose imperative languages, which access and update array elements one-by-one using loops, vectorization languages provide special syntax and linear algebra operations that can be applied to whole arrays, instead of individual elements, thus making loops inessential. These operations are often implemented as light-weight wrappers around high-performance array libraries, such as BLAS [13] and LAPACK [3], and rely on the limited array storage structures and routines provided by these libraries. These array libraries have also been the core of many machine learning (ML) libraries, such as TensorFlow [1], PyTorch [30], and Spark MLlib [6], which use these array libraries for fast in-memory array processing but have also added support for multicore parallelism, hardware acceleration (on GPUs, TPUs, and SIMD), and distributed processing.

Implementing tools and APIs on top of existing linear algebra libraries, which we call the *library approach*, suffers from several drawbacks. These libraries support a limited number of array storages, mainly because array operations must work on any combination of input and output array storages and this can lead to an explosion in the number of operation implementations. Furthermore, some of these combinations may not be supported, such as multiplying a distributed matrix with an in-memory matrix. More importantly, array storages supported by linear algebra libraries are hard to customize, because the introduction of a new array storage would require adding many more operator implementations to handle operations like matrix multiplication that may involve multiple dissimilar array storages. In terms of performance, array programs must be expressed using the supported array operators, which may result in suboptimal code when a program does not completely match any of these operators. Moreover, chains of array operations may introduce superfluous arrays, created by one operation just to be consumed by the next operation in the chain. Such arrays could have been avoided if this chain was written as a single imperative program. Finally, inter-operation optimization is often hardwired in rules that involve specific operation combinations and cannot be easily extended with new operators.

Our goal is to design a customizable framework for translating large-scale array programs in which arrays and array operations are abstract but their implementation is guided by user-defined storage mappings. To translate these programs to code that is as efficient as code written by expert programmers, we need to provide meaning-preserving program transformations that fuse abstract programs with storage mappings so that the generated code works directly on array storages using efficient array operations and avoids creating abstract arrays. This fusion can only be accomplished if both array operations and storage mappings are expressed in the same well-formed algebra or calculus with strong normalization properties. Furthermore, we want to design an effective storage structure for multi-dimensional arrays that facilitates distributed array processing and generalizes existing storages used by current linear algebra and ML libraries.

An abstract array in our framework is a mapping from array indices to values, represented as an association list of key-value pairs in which the key contains the array indices and the value is the array value. Abstract arrays are implemented using efficient storage structures based on customized storage mapping functions. Our main storage structure for multi-dimensional arrays is a *tensor*, which can be in memory or distributed and can have any number of sparse and dense dimensions. Unlike the library approach, which implements each array operation by providing a variety of array storages and algorithms, our framework expresses array computations in a powerful, well-formed abstraction called an *array comprehension*, which constructs a new array from existing arrays in a single shot using a syntax that resembles SQL. An array comprehension traverses elements from arrays in an unspeci-

fied order and constructs a new array by associating indices to values. Abstract computations based on array comprehensions are translated to efficient code over array storages based on customized array storage mapping functions and using simple program transformations - hence the name, a *transformational approach*.

For example, a matrix A with values of type Double is represented as an association list of type List[((Int,Int), Double)] so that an element A_{ij} is the key-value pair $((i,j),A_{ij})$, which associates the indices i and j with the value A_{ij}. The product of two matrices A of size $n*l$ and B of size $l*m$, which is a matrix R of size $n*m$ such that $R_{ij} = \sum_k A_{ik} * B_{kj}$, can be expressed as follows as an array comprehension:

```
tensor*(n,m)[ ((i,j),+/v) | ((i,k),a) ← A, ((kk,j),b) ← B,          (1)
                            kk == k, let v = a*b, group by (i,j) ].
```

This comprehension retrieves the values $A_{ik} \in A$ and $B_{kj} \in B$ as index-value pairs $((i,k),a)$ and $((kk,j),b)$ such that $kk = k$, and sets $v = a * b = A_{ik} * B_{kj}$. After the values are grouped by the indices i and j, the variable v is lifted to a bag of numerical values $A_{ik} * B_{kj}$, for all k, for each different combination of (i,j). The aggregation $+/v$ sums up all the values in the bag v, deriving $\sum_k A_{ik} * B_{kj}$ for the ij element of the resulting matrix. This comprehension constructs a dense distributed tensor of size $n*m$ (indicated by the annotation tensor*(n,m)). Internally, this tensor is stored as a Resilient Distributed Dataset (RDD) in Spark [5]:

```
( (Int,Int), RDD[( (Int,Int), Array[Double] )] ),                   (2)
```

where the first pair of integers is the tensor dimensions (n,m) and each element $((ci,cj),\text{block})$ of the RDD contains a matrix block in row-major format and its block coordinates (ci,cj). That is, the resulting matrix is a distributed dataset of non-overlapping dense blocks.

In general, a multi-dimensional tensor may have any number of sparse and dense dimensions. For instance, tensor*(n)(m) is the tensor store function for a distributed matrix of size $n*m$, where each block of the matrix is stored in the Compressed Sparse Row (CSR) format, where the rows are dense and the columns are sparse. Array comprehensions, which specify array computations on abstract arrays, are translated to concrete high-performance algorithms over the storages of the arrays traversed in the comprehension and store the results into array storages. An array storage is specified by two functions, a *view* function that maps the storage to an abstract type and its inverse function, *store*. An array traversal in an array comprehension becomes a list traversal by converting the array storage to a list of index-value pairs using its view function, which is inferred from the storage type of the traversal. The array comprehension result, on the other hand, is converted to an array storage using the store function that annotates the comprehension, such as, tensor*(n,m) in (1). Our optimizer fuses the embedded store and view functions with the array comprehension deriving an efficient algorithm that works directly on array storage structures and creates a new array storage structure without materializing any intermediate abstract arrays. These type mappings from abstract to concrete storages are customizable and extensible, while translating programs based on new storage mappings does not require any fundamental extension to our framework.

Based on the storage given in (2), our framework translates (1) to the following expression that works directly on the RDDs A._2 and B._2 and constructs a new RDD:

```
((n,m),rdd[ ((ci,cj),sum(L))                                        (3)
          | ((ci,ck),a) ← A._2, ((ckk,cj),b) ← B._2,
            ckk == ck, let L = prod(a,b), group by (ci,cj) ]),
```

where the rdd store function creates an RDD from the comprehension result. Unlike (1), this comprehension retrieves the blocks a and b from the input RDDs and uses the in-memory block multiplication, prod(a,b), and the block addition of a list of blocks, sum(L), to derive the output blocks. It is translated to a Spark join followed by a reduceByKey operation. These in-memory block operations, prod and sum, are derived as array comprehensions over in-memory dense tensors and are built using the in-memory store mapping tensor(N,N), which has fixed block dimensions $N*N$:

```
prod(a,b) = tensor(N,N)[ ((i,j),+/v) | ((i,k),x) ← a, ((kk,j),y) ← b,
                                       kk == k, let v = x*y, group by (i,j) ],
sum(L) = tensor(N,N)[ ((i,j),+/x) | X ← L, ((i,j),x) ← X,
                                    group by (i,j) ].
```

Like all storage types, our framework translates comprehensions over in-memory tensors by unfolding and fusing their view-store functions with the comprehension body, deriving Scala code that uses low-level array operations, such as array indexing, to build and access an array storage directly. Finally, our framework translates the Dataset comprehension (3) to a Spark DataFrame SQL query:

```
select a._1, sum(collect_list(prod(a,b))) as _2
from A._2 as a, B._2 as b
where a._1._2 = b._1._1
group by a._1._1, b._1._2,
```

where the DataFrame function collect_list collects all the prod(a,b) values of a group into a list. Alternatively, our framework can store distributed tensors as Spark RDDs of blocks and can translate array comprehensions to RDD methods from the Spark Core API. For example, the comprehension (3) is translated to a Spark join followed by a reduceByKey operation.

Comprehensions have been designed to serve as an intermediate language for high-level array-based languages. Vectorization languages, such as NumPy and R, can be easily translated to array comprehensions by defining each array operator as a comprehension, instead of providing numerous operator implementations. Such a scheme can be easily extended with custom operations and custom storage structures. In an earlier work [18], we presented a framework, called DIABLO (a Data-Intensive Array-Based Loop Optimizer), for translating array-based loops expressed in an imperative language to array comprehensions. The input language used by that system resembles the syntax of some loop-based imperative languages,

such as C and Java. This system can translate any array-based loop expressed in this loop-based language to an equivalent array comprehension as long as this loop satisfies some simple syntactic restrictions, which are more permissive than the recurrence restrictions imposed by many other systems. For example, matrix multiplication, expressed as follows in a loop-based language:

```
for i = 0, n–1 do
  for j = 0, m–1 do {
    R[i,j] := 0;
    for k = 0, l–1 do
      R[i,j] += A[i,k]*B[k,j];  },
```

is translated to the comprehension (1). Another high-level array language that can be translated to array comprehensions is the Einstein notation [36], which resembles array loops in an imperative language but without using loops. Essentially, each index variable that occurs in an assignment ranges implicitly within the array bounds. Furthermore, if the assignment is an increment operator, such as +=, then the aggregation is over all source values derived from the index variables that do not occur in the assignment destination. For example, the assignment R[i,j] += A[i,k]*B[k,j] is equivalent to the previous loop-based program for matrix multiplication because the aggregation is done across k since this variable does not occur in the destination. The Einstein notation though requires various extensions with additional syntax to capture more complex operations, such as matrix slicing and concatenation, which is easy to specify using loops and comprehensions.

Array comprehensions can capture many linear algebra operations, including inner and outer products of vectors, matrix addition and multiplication, matrix rotation and transpose, array slicing, padding, convolution, and concatenation. More complex array operations, such as SVD decomposition, linear equation solving, matrix inversion, and eigenvalue computation, can be coded using array comprehensions inside loops.

One important contribution of this paper is showing that our generic, operation-agnostic framework can produce quality code that is nearly as efficient as the highly optimized code written by expert programmers of linear algebra libraries. That is, it achieves implementation independence and extensibility without sacrificing performance. Linear algebra libraries, such as BLAS, use optimal algorithms, such as the SUMMA parallel algorithm for block matrix multiplication [19]. In this paper, we show that such algorithms are actually boilerplate code that can be applied to many more cases, which can be easily identified and translated when the code is expressed as comprehensions. More specifically, we will show that comprehensions on distributed tensors that contain at least one join followed by a group-by and aggregation can be evaluated with an optimal algorithm that generalizes SUMMA (Section VIII-D).

Our system, called *SAC* (Scalable Array Comprehensions), has been implemented on Apache Spark using Scala's compile-time reflection and macros. The system architecture is shown in Fig. 1. The modules inside the dashed area are part

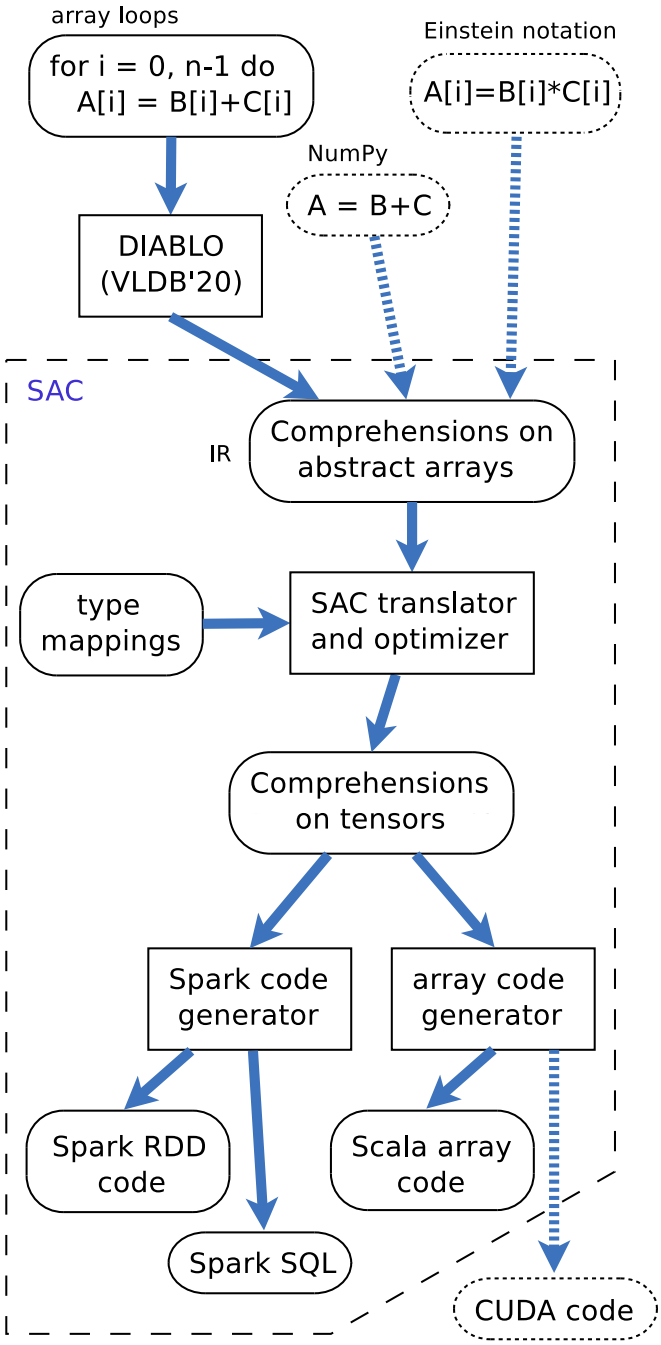

Fig. 1: SAC Architecture

of SAC, described in this paper. Comprehensions on abstract arrays are designed to serve as an Intermediate Representation (IR) for high-level array-based languages, such as imperative loop-based languages (addressed in our earlier work [18]), the Einstein notation, and vectorization languages (left for a future work). SAC translates comprehensions on abstract arrays to tensor comprehensions based on customized type mappings, which in turn are optimized and translated to Spark programs based on either the Spark Core API or Spark SQL. These Spark programs operate on array blocks using Scala array code

or CUDA code to run on NVidia GPUs (left for future work).

In conclusion, our transformational approach has many benefits over the library approach, as it

- is easy to extend with customizable array storage structures;
- prevents an explosion in the number of operation implementations when new storages are introduced;
- generates code that is easy to validate since it uses meaning preserving transformations;
- avoids the construction of intermediate arrays, created in chains of array operations;
- facilitates inter-operational optimization by fusing and optimizing the bodies of consecutive operations.

The contributions of this paper are summarized as follows:

- We introduce a new storage structure for arrays, called the distributed tensor, which is a distributed collection of array blocks that may have any number of sparse and dense dimensions.
- We provide a language for specifying customized storage mappings.
- We present a framework for translating in-memory array computations to efficient imperative code that uses array indexing and updating.
- We present a framework for translating distributed array comprehensions to Spark code over distributed tensors , expressed in either the Spark DataFrame SQL or the Spark Core API.
- We evaluate the performance of our system relative to Spark MLlib and TensorFlow on a variety of linear algebra and ML programs. Based on these results, the performance of our system is on par with these systems.

The work reported in this paper extends our previous work on array comprehensions ([16], [18], [28]) in many ways: it introduces two novel array storage structures, the in-memory and distributed tensors, that generalize the array storage structures used in our earlier work as they allow any number of sparse and dense dimensions. In addition, distributed tensors are organized in array blocks (instead of index-value pairs, as in [18], which makes them very compact and more efficient to process. Furthermore, storage mappings are now customizable and extensible through a special language that allows user-defined storages, instead of being hardwired into the translator. Finally, our translation and optimization rules have been extended and improved to work on the newly introduced tensors.

II. Background: Array Comprehensions

In our framework, array operations and mappings to low-level array storages are expressed as monoid comprehensions [14]. Monoid comprehensions generalize list comprehensions by mixing traversals on diverse collections (such as lists, bags, and arrays) in the same comprehension and by supporting a syntax for group-by and aggregation. The formal semantics, the translation to an algebra, and the optimization of comprehensions are described in our earlier work ([14], [17]) and are summarized in this section.

The syntax of a comprehension is $[\, e \mid q_1, \ldots, q_n \,]$, where the expression e is the head of the comprehension and each q_i is a qualifier Qualifiers and expressions have the following syntax:

Qualifier:

q	::=	$p \leftarrow e$	generator
	|	**let** $p = e$	let-binding
	|	e	condition
	|	**group by** $p\, [: e\,]$	group-by

Pattern:

p	::=	v	pattern variable
	|	$(p_1, \ldots, p_n)$	tuple pattern

Expression:

e	::=	$[\, e \mid q_1, \ldots, q_n \,]$	comprehension
	|	$\oplus/e$	aggregation
	|	$e_1 .. e_2$	range of integers
	|	any Scala expression.	

The domain e of a generator $p \leftarrow e$ must be a collection, such as a list or an array. This generator draws elements from this collection and, each time, it binds the pattern p to an element. A condition qualifier e is an expression of type boolean. It is used for filtering out elements drawn by the generators. A let-binding **let** $p = e$ binds the pattern p to the result of e. A group-by qualifier uses a pattern p and an optional expression e. If e is missing, it is taken to be p. The group-by operation **group by** $p\, [: e\,]$ groups all the pattern variables in the same comprehension that are defined before the group-by (except the variables in p) by the value of e (the group-by key), so that all variable bindings that result in the same key value are grouped together. After the group-by, the pattern p is bound to one of the group-by keys and each one of the rest pattern variables in the comprehension is lifted to a list of values. The result of $[\, e \mid q_1, \ldots, q_n \,]$ is a list that contains all values of e derived from the variable bindings in the qualifiers. Finally, an expression can be an aggregation $\oplus/e$ that uses the monoid $\oplus$ (a binary associative function with a zero value) to aggregate the values of the collection returned by e. That is, if e evaluates to $\{v_1, v_2, \ldots, v_n\}$, then $\oplus/e$ is equal to $v_1 \oplus v_2 \oplus \cdots \oplus v_n$.

For example, the comprehension

```
[ ( dno, +/salary ) | ( name, address, dno, salary ) ← Employees,
                      group by dno ]
```

groups Employees by their department number dno and, for each different department, it returns the total salary of the employees in the department. After the group-by, the pattern variable salary of type Double is lifted to a List[Double], which contains all salaries associated with a certain group key dno, and +/salary returns the sum of all these salaries by reducing the values of the collection salary in pairs using +.

Comprehensions can be normalized using the following rule that flattens nested comprehensions [14]:

$$[\, e_1 \mid \overline{q_1},\ p \leftarrow [\, e_2 \mid \overline{q_3} \,],\ \overline{q_2} \,] = [\, e_1 \mid \overline{q_1},\ \overline{q_3},\ \textbf{let}\ p = e_2,\ \overline{q_2} \,] \qquad (4)$$

for any sequence of qualifiers $\overline{q_2}$, and for any sequences of qualifiers $\overline{q_1}$ and $\overline{q_3}$ that do not contain a group-by qualifier.

This transformation may require renaming the variables in $[\, e_2 \mid \overline{q_3} \,]$ to prevent variable capture.

Arrays and array comprehensions have been introduced in our earlier work ([16], [18], [28]). An abstract array with dimensionality i has type array$i$[T], for an arbitrary type T. The most common abstract arrays are vector[T], equal to array1[T], and matrix[T], equal to array2[T]. An abstract array is specified as an association list of key-value pairs in which the key consists of the array indices. This array representation is also known as a sparse representation or a coordinate format. For example, a matrix M of type matrix[Double] is represented as an association list of type List[((Int,Int),Double)] so that an element M_{ij} is represented by the key-value pair $((i,j), M_{ij})$, which associates the indices i and j with the value M_{ij}. This association list can be sparse if some elements are missing.

Programs that operate on abstract arrays are expressed using array comprehensions. An array comprehension is a monolithic array construction that is as expressive as SQL by supporting a group-by syntax that allows us to capture many array computations in declarative form. Since abstract arrays are specified as association lists, array comprehensions are actually list comprehensions. That is, unlike linear algebra libraries that hide the implementation details, array operations fully expose their operation specification so that they can be fused with other array operations using Eq. (4) and optimized using relational optimizations, such as rearranging a chain of matrix multiplications using join reordering. Consider, for example, the following array comprehension that constructs a vector V from a matrix M such that $V_i = \sum_j M_{ij}$:

$$\mathsf{V} = [\ (\ \mathsf{i},\ +/\mathsf{m}\ )\ |\ ((\mathsf{i},\mathsf{j}),\mathsf{m}) \leftarrow \mathsf{M},\ \textbf{group by}\ \mathsf{i}\ ], \tag{5}$$

which constructs the entire array V in one shot. In our framework, the matrix M of type matrix[Double] is represented as an association list of type List[((Int,Int),Double)], while the vector V of type Vector[Double] is represented as an association list of type List[(Int,Double)]. The generator ((i,j),m) ← M in the comprehension (5) traverses M one element at a time and each time the traversed element is pattern-matched with the pattern ((i,j),m), which binds the pattern variables i, j, and m to the corresponding components of this element. A group-by operation in a comprehension lifts each pattern variable defined before the group-by (except the group-by keys) from some type T to a List[T], indicating that each such variable must now contain all the values associated with the same group-by key. Consequently, after we group by i, the pattern variables that are not in the group-by key, namely j and m, are lifted to lists that contain all their values associated with a certain group-by key i. The term +/m adds up all the bindings of the variable m associated with the key i, that is, it reduces the values of m in pairs using +, which calculates $\sum_j M_{ij}$.

Other examples of array operations are the addition of two matrices M and N:

$$[\ ((\mathsf{i},\mathsf{j}),\mathsf{m}+\mathsf{n})\ |\ ((\mathsf{i},\mathsf{j}),\mathsf{m}) \leftarrow \mathsf{M},\ ((\mathsf{ii},\mathsf{jj}),\mathsf{n}) \leftarrow \mathsf{N},\ \mathsf{ii} == \mathsf{i},\ \mathsf{jj} == \mathsf{j}\ ], \tag{6}$$

the multiplication of two matrices M and N:

$$[\ ((\mathsf{i},\mathsf{j}),+/\mathsf{v})\ |\ ((\mathsf{i},\mathsf{k}),\mathsf{m}) \leftarrow \mathsf{M},\ ((\mathsf{kk},\mathsf{j}),\mathsf{n}) \leftarrow \mathsf{N},\ \mathsf{kk} == \mathsf{k},$$

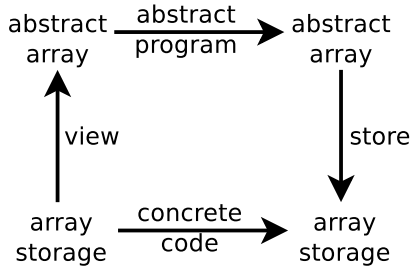

Fig. 2: Program Translation

$$\textbf{let}\ \mathsf{v} = \mathsf{m}*\mathsf{n},\ \textbf{group by}\ (\mathsf{i},\mathsf{j})\ ],$$

and the rotation of the rows of a matrix M:

$$[\ (\ (\ (\mathsf{i}+1)\%\mathsf{rows},\ \mathsf{j}\ ),\ \mathsf{m}\ )\ |\ ((\mathsf{i},\mathsf{j}),\mathsf{m}) \leftarrow \mathsf{M}\ ], \tag{7}$$

where rows is the number of rows.

III. Type Mappings

When designing storage structures for arrays in a distributed setting, there are many choices to consider, each exhibiting different performance characteristics for various array computations. One example of such a storage method is organizing contiguous array elements into dense non-overlapping blocks of fixed capacity. The conventional way to separate implementation from specification is to provide a library of abstract operations (an API) and implement these operations using the best storage structures and algorithms, as linear algebra libraries currently do. Then, to support multiple array storages, these abstract operations are overloaded to work on a variety of implementations. But, as we discussed in the introduction, this scheme is hard to extend with new storage structures and does not facilitate inter-operation optimization, where the aim is at generating efficient code from a chain of operations without creating superfluous intermediate arrays. Unlike the library approach, Our framework uses a two-layer approach to implement array operations, where array programs are expressed as array comprehensions while abstract arrays are mapped to customized storage structures. That way, the code of both the abstract programs and the type mappings is exposed to the optimizer, which can collapse the two mapping layers by fusing the type mappings with the abstract program, thus yielding concrete code that works directly on array storages without materializing the abstract arrays. Our framework is oblivious to linear algebra operations. Instead, abstract programs are expressed as array comprehensions, which in turn are translated to code without any knowledge of which array operations these comprehensions represent. This independence from linear algebra operations and the rules they obey generalizes and simplifies inter-operation optimization and allows system extension with custom array storages.

In our framework, the type mappings from abstract types to their concrete storage types is done using two layers of mappings. The first mapping, called the *default mapping*, maps an abstract type to its default *specification type* based on fixed

mapping rules. This specification type is typically a list type so that instances of the abstract type can be traversed in a list comprehension and the result of a list comprehension can be converted to abstract types. As described in Section II, the specification type of Array$i[T]$ is List$[(\text{Int}^i, T)]$, where Int^i is a tuple with i Int values and each (Int^i, T) is an index-value pair from the array. The second mapping, called the *storage mapping*, consists of two customized functions: the *view*, which converts the storage structure to the specification type, and the *store*, which constructs the storage structure from the specification type (Fig. 2). These two functions, which are inverse of each other, are used by our translator to transform any program on abstract arrays to concrete code that accesses and builds array storage structures using efficient code. In our framework, both mapping functions are expressed as list comprehensions with side-effects, which, unlike array comprehensions, use low-level array operations, such as array indexing, to build and access an array storage directly.

This layered approach introduces levels of indirection and generates superfluous intermediate structures that need to be removed. Collapsing the composition of these three functions (the view, the abstract program, and the store) is accomplished by fusing these functions into one function that represents the concrete code. This program fusion is done by unnesting nested comprehensions into flat comprehensions using Eq. (4). The main array storage structures used for implementing multi-dimensional arrays in our framework are general *in-memory tensors*, which may have dense and sparse dimensions, and *distributed tensors*, which are distributed arrays partitioned into a set of non-overlapping blocks, where each block is implemented as an in-memory tensor.

In our framework, array comprehensions are actually list comprehensions that traverse lists and return lists. When an abstract array is created, its storage must be explicitly specified in the form of a storage mapping. When a comprehension traverses an array storage, the array is implicitly up-coerced to a list by the view function of the storage mapping. The result of the comprehension, on the other hand, must be explicitly down-coerced to some array storage via a store function when the storage cannot be inferred. For example, consider the following two statements:

```
var V = tensor(d)[ (i,i*3) | i ← 0..(d–1) ];          (8)
V = [ (i,v+1) | (i,v) ← V ];                          (9)
```

The vector V in statement (8) is stored as a one-dimensional tensor (a vector) of size d. Here, the result of the comprehension is explicitly down-coerced to a tensor storage by the storage constructor tensor(d). The abstract type of V is vector[Int], which is inferred from the type mapping tensor(d). The comprehension in statement (9) on the other hand does not need a storage constructor because it can be inferred from the assignment to V.

All type mappings in our framework must be defined using a **typemap** declaration. The simple tensor for vectors used in the previous example can be defined as follows: (Tensors are generalized to multi-dimensional dense and sparse storages in Section IV.)

```
typemap tensor[T] ( size: Int ): vector[T] {
  def view ( a: Array[T] )
    = [ (i,a(i)) | i ← 0..(size–1) ]
  def store ( L: List[(Int,T)] )
    = {  var a = Array.ofDim[T](size);
         [ a(i) = v | (i,v) ← L ];
         a  } },
```

where a(i) is the syntax for array indexing in Scala and the Scala function Array.ofDim[T](d) constructs an array of type T and size d. This type mapping implements the abstract type vector[T] as a storage with Scala type (Int,Array[T]), where the Int is the array size. More specifically, the view function is from (Int,Array[T]) to (Int,List[(Int,T)]) and the store function is the inverse. The tensor arguments (i.e., the size) are passed to both functions as implicit arguments and returned as is in the results. Both functions use efficient array indexing and updates. Based on this mapping, the comprehension in statement (9) is transformed into:

```
store( V._1, [ (i,v+1) | (i,v) ← view(V._1,V._2)._2 ] ),
```

which, after flattening the nested comprehensions and translating the result to Scala code, becomes:

```
( V._1, {  var a = Array.ofDim[T](V._1);
           0.until(V._1).foreach( i => a(i) = V._2(i)+1 );
           a  } ).
```

IV. In-Memory Tensors

An abstract array of type array$i[T]$ (an array with i dimensions) can be stored in memory as a tensor in which some of the dimensions are dense and the rest are sparse. A tensor of type T with n dense and m sparse dimensions is constructed from a list e using the storage constructor

$$\text{tensor}(d_1, \ldots, d_n)(s_1, \ldots, s_m)\ e$$

where d_1 through d_n are the dense dimensions, s_1 through s_m are the sparse dimensions, and e is a list of type (Int^i, T), where $i = n + m$. A dense tensor is constructed using the storage constructor $\text{tensor}(d_1, \ldots, d_n)\ e$, which is equal to $\text{tensor}(d_1, \ldots, d_n)()\ e$. To support sparse arrays of any type T, our framework defines the zero element zero^T of this type, since zero elements are omitted. Examples of zero elements are 0 for Int and Long, 0.0 for Float and Double, false for Boolean, and null for objects.

For example, the following comprehension constructs a matrix 10×20 by multiplying the matrices M and N:

```
tensor(10)(20)[ ((i,j),+/v) | ((i,k),m) ← M, ((kk,j),n) ← N, kk == k,
                              let v = m*n, group by (i,j) ].
```

The row dimension (of size 10) of the resulting matrix is dense while the column dimension (of size 20) is sparse.

The layout we use for storing in-memory tensors generalizes the row-major layout for dense matrices and the Compressed

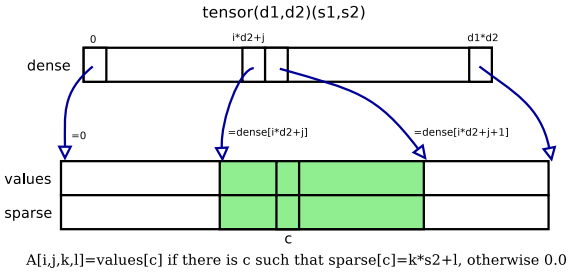

Fig. 3: A tensor with 2 dense and 2 sparse dimensions

Sparse Row (CSR) format for sparse matrices. To describe the mapping of a multi-dimensional array to a tensor, we use the following function that maps a multi-dimensional index $(i_1, \ldots, i_n)$, where each i_k ranges from 0 to $d_k - 1$, to a unique location between 0 and $d_1 \times \cdots \times d_n - 1$:

$$\begin{array}{rcll} \mathcal{I}(i_1 : d_1) & = & i_1, & \text{if } n = 1 \\ \mathcal{I}(i_1 : d_1, \ldots, i_n : d_n)] & = & \mathcal{I}(i_1 : d_1, \ldots, i_{n-1} : d_{n-1}) * d_n + i_n. & \end{array}$$

For example, $\mathcal{I}(i_1 : d_1, i_2 : d_2) = i_1 * d_2 + i_2$.

A tensor of type T with n dense dimensions $d_1, \ldots, d_n$ and m sparse dimensions $s_1, \ldots, s_m$ is stored into three vectors: a vector dense of type Array[Int], a vector sparse of type Array[Int], and a vector values of type Array[T]. It is stored as a triple of type:

$$\text{tensor\_}n\text{\_}m[T] = (\text{Int}^n, \text{Int}^m, (\text{Array[Int]}, \text{Array[Int]}, \text{Array[T]})),$$

where the first component contains the dense tensor dimensions, the second component contains the sparse tensor dimensions, and the arrays are the dense, sparse, and values vectors. The array dense has size $d_1 \times \cdots \times d_n + 1$. Given the dense indices $i_1, \ldots, i_n$, the non-zero elements $A[i_1, \ldots, i_n, j_1, \ldots, j_m]$ for all sparse indices $j_1, \ldots, j_m$ can be found in the vectors sparse and values between the locations dense$[k]$ and dense$[k+1]-1$, where $k = \mathcal{I}(i_1 : d_1, \ldots, i_n : d_n)$. More specifically, $A[i_1, \ldots, i_n, j_1, \ldots, j_m]$ is equal to values(c) if there is a c between dense$[k]$ and dense$[k+1]-1$ such that sparse$[c] = \mathcal{I}(j_1 : s_1, \ldots, j_n : s_n)$, otherwise, it is equal to zero^T. The index-value pairs in the vectors sparse and values between dense$[k]$ and dense$[k+1]-1$ are kept sorted for each k to enable binary search on the sparse indices.

For example, the tensor$(d_1, d_2)(s_1, s_2)$ in Fig. 3 implements a 4-dimensional array as a tensor with 2 dense and 2 sparse dimensions. Then, $A[i, j, k, l]$ is equal to values$[c]$ for some c between dense$[i * d_2 + j]$ and dense$[i * d_2 + j + 1] - 1$ such that sparse$[c] = k * s_2 + l$, otherwise it is equal to zero^T.

To minimize storage, the values vector is not used for sparse boolean tensors because all non-zero values are equal to true. This is useful when we represent a graph as a boolean matrix (the adjacency matrix) and we store it as a sparse boolean tensor since most matrix values are false for many graphs. Furthermore, a dense tensor (a tensor with no sparse dimensions) is stored using a single vector, dense.

The storage mapping of a tensor with n dense and m sparse dimensions has the name tensor_n_m and is generated lazily (on demand) when the tensor is first used in a program. For a boolean tensor, it is a special tensor with the name bool_tensor_n_m. For example, the generated storage mapping of a dense tensor tensor(d_1, d_2) is tensor_2_0, defined as follows:

```
typemap tensor_2_0[T] ( d: (Int,Int), s: ( ) ): array2[T] {
  def view ( dense: Array[T] )
    = [ ((i,j),dense(i*d._2+j)) | i ← 0..(d._1−1), j ← 0..(d._2−1) ]
  def store ( L: List[((Int,Int),T)] )
    = {  var a = Array.ofDim(d._1*d._2);
         [ a(i*d._2+j) = v | ((i,j),v) ← L ];
         a  } }.
```

That is, this tensor implements the abstract type array2$[T]$ as a storage of type tensor_2_0[T] equal to $((\text{Int}, \text{Int}), (), \text{Array}[T])$, for any type T, in which the two Ints are the dimensions in d, the sparse dimensions are empty ( ), and the Array is the vector dense. The view function maps the array storage to its array specification, which is a list of type List$[((\text{Int}, \text{Int}), T)]$. The store function maps the array specification L to the array storage, starting with a vector with zero values and by updating the vector in-place using the index-value pairs read from the list L. Another example is the sparse tensor, tensor$(d)(s)$, shown in Fig. 3:

```
typemap tensor_1_1[T] ( d: Int, s: Int ): array2[T] {
   def view ( dense, sparse: Array[Int], values: Array[T] )
     = [ ((i,sparse(col)),values(col))
       | i ← 0..(d−1), col ← (dense(i))..(dense(i+1)−1) ]
   def store ( L: List[((Int,Int),T)] )
     = {  var buffer = Array.fill[List[(Int,T)]](d)(Nil);
          [ buffer(i) = (j,v)::buffer(i) | ((i,j),v) ← L, v != zero[T] ];
          /* build the sparse and values vectors from buffer */ } }.
```

The buffer used in the store function is a vector of d lists of type List[(Int,T)]. The list buffer(i) will contain the (j,v) pairs from ((i,j),v) in L with a non-zero v. The buffer is then used to populate the vectors sparse and values (code omitted).

The view function of a sparse tensor is efficient because it only touches the non-zero elements. But there are some array operations that need to process the zero elements too, such as the comprehension tensor(10)(20)[((i,j),v+1) | ((i,j),v) ← M], which maps zero values to non-zeros. Another example is matrix addition that needs to return a non-zero value when an element in one matrix is zero while the associated element in the other matrix is not. For such operations, the view function would produce incorrect translations because it ignores the zero elements. To correct this problem, we provide an alternative (but less efficient) view function for sparse tensors that considers the zero elements:

```
  def view ( dense, sparse: Array[Int], values: Array[T] )
     = [ ((i,j),binarySearch(dense(i),dense(i+1)−1,sparse,values))
       | i ← 0..(d−1), j ← 0..(s−1) ],
```

which uses binary search to find the value of each i and j, but returns zero if it does not find one. It is generally undecidable

to tell whether it is necessary to use the latter view function for traversing a sparse matrix in a comprehension. Instead, we provide an alternative qualifier for traversing a sparse matrix, $p \Leftarrow e$, that uses the less efficient view function. The semantics of an $\Leftarrow$ traversal is different from that of $\leftarrow$ since the latter skips the zero elements. For example, the previous comprehension should be written as:

```
tensor(10)(20)[ ((i,j),v+1) | ((i,j),v) ⇐ M ],
```

which will now increment the zero values. Hence, it is up to the programmer to decide which variant to use: the complete but less efficient scan using $\Leftarrow$ or the more efficient scan using $\leftarrow$.

V. Distributed Tensors

A distributed tensor is a storage structure for multi-dimensional arrays implemented as a distributed dataset of non-overlapping blocks where each block is an in-memory tensor. Our framework uses two ways to implement these distributed datasets: as Resilient Distributed Datasets (RDDs) on Apache Spark based on the Spark Core API [41] and as DataFrames based on Spark SQL [7]. To simplify the description of our framework, we define distributed tensors in terms of RDDs only and we translate tensor comprehensions to Spark Core API and Spark SQL.

A distributed tensor of type T with n dense and m sparse dimensions implements an abstract array of type array$i[T]$ with $i = n + m$ dimensions. It is constructed from a list e using the storage constructor

$$\text{tensor}*(d_1, \ldots, d_n)(s_1, \ldots, s_m)\ e,$$

where d_1 through d_n are the dense dimensions, s_1 through s_m are the sparse dimensions, and e is a list of type $\text{List}[(\text{Int}^{m+n}, T)]$. This storage constructor looks similar to that of an in-memory tensor except that it uses the name tensor$*$ instead of tensor. Like in-memory tensors, a dense distributed tensor is constructed using the storage constructor $\text{tensor}*(d_1, \ldots, d_n)\ e$.

The distributed tensor $\text{tensor}*(d_1, \ldots, d_n)(s_1, \ldots, s_m)\ e$ is stored as a Spark RDD that has the following type:

$$\text{block\_tensor}\_n\_m[T] = (\text{Int}^n, \text{Int}^m, \text{RDD}[(\text{Int}^{n+m}, \text{tensor}\_n\_m[T])]),$$

where the first and second components of the triple contain the dense and sparse dimensions, respectively, and the third is an RDD of pairs where each pair contains the block coordinates and one block, which is an in-memory tensor with n dense and m sparse dimensions. Each block dimension in our framework has a fixed size N (default is 1000), which can be set at the beginning of the program but must remain constant to enable some optimizations. The last blocks in each dimension (such as the last row and column blocks) can be smaller in size. An element $A[i_1, \ldots, i_n, j_1, \ldots, j_m]$ of the abstract array implemented using this distributed tensor is located in the block B with coordinates $(ci_1, \ldots, ci_n, cj_1, \ldots, cj_m)$, where $ci_k = i_k/N$ and $cj_k = j_k/N$ for all k. The value of this element inside this block B is $B[i_1\%N, \ldots, i_n\%N, j_1\%N, \ldots, j_m\%N]$. For example, the type mapping that implements a distributed tensor $d \times s$ and has one dense and one sparse dimension is the following:

```
typemap block_tensor_1_1[T] ( d: Int, s: Int ): array2[T] {
  def view ( x: RDD[((Int,Int),tensor_1_1[T])] )
    = [ ((ci*N+i,cj*N+j),v) | ((ci,cj),a) ← x, ((i,j),v) ← a ]
  def store ( L: list[((Int,Int),T)] )
    = rdd[ ((ci,cj),tensor(N)(N)(w))
         | ((i,j),v) ← L, let ci = i/N, let cj = j/N,
           let w = ((i%N,j%N),v), group by (ci,cj) ] }.
```

The variable a in the view function is an in-memory tensor of type tensor_1_1[T]. The store function calls the in-memory tensor constructor tensor(N)(N)(w) to construct a block, which is a $N \times N$ matrix in CSR format. The block dimensions td and ts are equal to N except for the last block of each row and column:

```
td = if ((ci+1)*N > d) then d%N else N,
ts = if ((cj+1)*N > s) then s%N else N.
```

Finally, the rdd type mapping maps a List[T] to an RDD[T].

VI. Translation of Tensor Comprehensions

Before we describe our methods for translating distributed array comprehensions (Section VIII), we explain how in-memory array comprehensions are translated to efficient Scala code. These translation methods are needed because distributed tensors consist of blocks, which are implemented as in-memory tensors.

An in-memory tensor comprehension is an array comprehension that traverses in-memory tensors and returns an in-memory tensor. As explained in Section III, an in-memory tensor comprehension can be translated to an efficient concrete program on arrays that uses low-level array operations, such as array indexing, to build and access array storages directly. This is accomplished by unrolling the implicit view function of the traversed in-memory tensors and the explicit store function of the comprehension result. Then, the resulting nested comprehension is flattened to an unnested comprehension that works directly on storage by using Eq. (4), which flattens nested comprehensions. Finally, the resulting comprehension is optimized by eliminating redundant array index traversals. More specifically, if two index generators i ← 0 .. n and j ← 0 .. m are related with i==j in a comprehension, then they are fused to one generator and one let-binding: i ← 0 .. (min(n,m)), **let** j = i.

For example, the in-memory matrix addition $X + Y$ on two dense matrices of size $d \times d$ is:

```
tensor(d,d)[ ((i,j),x+y) | ((i,j),x) ← X, ((ii,jj),y) ← Y, ii == i, jj == j ],
```

which is equal to the following comprehension:

```
tensor_2_0.store( (d,d), ( ),
                  [ ((i,j),x+y) | ((i,j),x) ← tensor_2_0.view(X),
                                  ((ii,jj),y) ← tensor_2_0.view(Y),
                                  ii == i, jj == j ] ).
```

After unrolling the view function of the typemap tensor_2_0 defined in Section IV and then flattening and optimizing the resulting comprehension, we get

```
tensor_2_0.store( (d,d), ( ),
                  [ ((i,j),X(i*d+j)+Y(i*d+j))
                  | i ← 0..(d−1), j ← 0..(d−1) ] ).
```

Finally, if we unroll the store function of tensor_2_0, we get:

```
((d,d),(),{  val a = Array.ofDim[Double](d*d);
             [ a(i*d+j) = X._3(i*d+j)+Y._3(i*d+j)
             | i ← 0..(d−1), j ← 0..(d−1) ];
             a  }),
```

which is translated to the following Scala code:

```
((d,d),(),{  val a = Array.ofDim[Double](d*d);
             for ( i ← 0 until d; j ← 0 until d )
                 a(i*d+j) = X._3(i*d+j)+Y._3(i*d+j);
             a  }).
```

A simple in-memory comprehension with a group-by is:

$$\mathcal{T}[\,(\overline{k},\oplus/v)\mid \overline{q},\ \textbf{group by}\ \overline{k}\,], \tag{10}$$

where $\mathcal{T}$ is an in-memory tensor, tensor$(\overline{d})(\overline{s})$, v is a local variable in $\overline{q}$ but not in $\overline{k}$, which has been lifted to a list of values after the group-by, and $\oplus/v$ uses the monoid $\oplus$ to aggregate these values. To translate this comprehension, instead of using the store function of the tensor $\mathcal{T}$, we use a special incr function for $\mathcal{T}$, which can be generated for any in-memory tensor. For example, the incr for tensor_2_0 is:

```
typemap tensor_2_0[T] ( d: (Int,Int), s: ( ) ): array2[T] {
  def incr ( L: List[((Int,Int),T)], ⊕: T=>T=>T, zero: T )
    = { var a = Array.ofDim(d._1*d._2)(zero);
        [ a(i*d._2+j) = a(i*d._2+j) ⊕ v | ((i,j),v) ← L ];
        a  } },
```

which, unlike the store function that uses a(i*d._2+j) = v, it increments the array values using a(i*d._2+j) = a(i*d._2+j) ⊕ v. Then, the group-by comprehension (10) is translated to:

$$\text{incr}(\overline{d},\overline{s},[\,(\overline{k},v)\mid \overline{q}\,],\oplus,\text{zero}),$$

while the resulting comprehension can be translated to efficient code as it was done for the in-memory comprehensions without group-by.

For example, the matrix multiplication between the dense tensors X and Y is:

```
tensor(d,d)[ ((i,j),+/v) | ((i,k),x) ← X, ((kk,j),y) ← Y,
                           let v = x*y, group by (i,j) ],
```

which is translated and optimized to

```
tensor_2_0.incr( (d,d), ( ),
                 [ ((i,j),v)
                 | i ← 0..(d−1), j ← 0..(d−1), k ← 0..(d−1),
                   let v = X._3(i*d+k)*Y._3(k*d+j) ],
                 +, 0 ),
```

which, after unrolling the incr function, becomes:

```
((d,d),(),{  val a = Array.ofDim[Double](d*d)(0.0);
             for ( i ← 0 until d; j ← 0 until d; k ← until d ) {
                 val v = X._3(i*d+k)*Y._3(k*d+j);
                 a(i*d+j) = a(i*d+j) + v  };
             a  }).
```

The above translations can be generalized to handle multiple aggregations in the comprehension head, as it was done in our earlier work [16].

VII. Translation of RDD Comprehensions

Our framework is based on two kinds of (distributed) datasets: Spark RDDs [41] and Spark DataFrames [7]. Dataset comprehensions are comprehensions over datasets that construct a dataset. In this section, we only describe how dataset comprehensions are translated to programs that can run on Spark SQL over DataFrame tables and on Spark Core API using RDD methods. Translations of comprehensions on distributed tensors are given in Section VIII. The dataset type mapping maps a list to a Spark DataFrame table [7]. Unlike tensor type mappings, the view and store functions of this dataset mapping are Spark methods that convert a DataFrame to a list and vice versa. However, dataset comprehensions must be translated to DataFrame SQL queries that work directly on DataFrame tables using special rules. Since SQL does not support patterns, the first transformation is to eliminate patterns from comprehensions by introducing a new variable for each pattern and by binding each pattern variable to a term that depends on this new variable. Then, the resulting dataset comprehension is compiled to a Spark SQL query, by first separating the comprehension generators, $p \leftarrow e$, into three categories:

1) distributed generators (when e is a distributed collection, such as a DataFrame, an RDD, or a distributed tensor),
2) in-memory generators (when e is an in-memory collection, such as a list or a memory tensor), and
3) dependent generators (when e depends on a distributed generator or some other dependent generator).

A distributed generator becomes part of the from-clause of the SQL query. An in-memory generator over an enumeration (a range between integers) also becomes part of the from-clause since Spark SQL supports ranges. A dependent generator is used for traversing nested types, such as an array of arrays. For example, tensor*(n,m)[((i,j),v) | (i,a) ← M, (j,v) ← a] converts a vector of vectors to a matrix. This join between M and a (which depends on M) is called a dependent join. Spark SQL supports dependent joins indirectly using a special clause in the query, called a lateral view combined with an explode operation. Both in-memory and dependent generators become part of the SQL lateral view clause. Finally, all subterms in predicates and in the comprehension head that are not supported by SQL become user-defined functions (UDFs) coded in Scala.

In addition to Spark SQL, Our system translates distributed comprehensions to the Spark Core API using our own query optimizer, which has been developed for the query systems MRQL [14] and DIQL [17]. This second translation method

is based on the rdd type mapping, which consists of a store function that converts a List[T] to an RDD[T] using the Spark method 'parallelize' on this list, and a view function that uses the Spark method 'collect' to convert the RDD to a list. However, like in the dataframe mapping, RDD comprehensions are translated to RDD operations in a special way to generate efficient RDD code that minimizes data shuffling across computers. For example, the following rule identifies and generates joins between the RDDs X and Y when $\text{vars}(e_1) \subseteq \text{vars}(p_1)$ and $\text{vars}(e_2) \subseteq \text{vars}(p_2)$, where function 'vars' returns the free variables in a pattern or expression:

$$\begin{aligned}\text{rdd}[\, e \mid \overline{q_1},\, p_1 \leftarrow X,\, \overline{q_2},\, p_2 \leftarrow Y,\, \overline{q_3},\, e_1 == e_2,\, \overline{q_4}\,] \\ = \text{rdd}[\, e \mid \overline{q_1},\, (\_, (p_1, p_2)) \leftarrow Z, \overline{q_2},\, \overline{q_3},\, \overline{q_4}\,],\end{aligned}$$

where Z is the result of joining X and Y:

$$Z = X.\text{map}\{p_1 \Rightarrow (e_1, p_1)\}.\text{join}(Y.\text{map}\{p_2 \Rightarrow (e_2, p_2)\}).$$

Another example is an RDD comprehension with a group-by of the form $\text{rdd}[\, f(\overline{k}, \oplus/v) \mid \overline{q},\, \textbf{group by}\, \overline{k} : e\,]$, where v is a local variable in $\overline{q}$ but not in $\overline{k}$, which has been lifted to a list of values after the group-by, and $\oplus/v$ uses the monoid $\oplus$ to aggregate these values. It is translated to:

$$\text{rdd}[\, (e, v) \mid \overline{q}\,].\text{reduceByKey}(\oplus).\text{map}\{\, (\overline{k}, x) \Rightarrow f(\overline{k}, x)\,\},$$

which groups the values v generated by the qualifiers $\overline{q}$ by the key e, then reduces each group of values using $\oplus$, and then binds the key to $\overline{k}$ and uses it along with the reduced result x to form the final result $f(\overline{k}, x)$.

VIII. Translation of Distributed Array Comprehensions

Our goal in this paper is to translate any program on abstract arrays to efficient code that works directly on distributed tensors and constructs a new distributed tensor using efficient array operations, such as array indexing and in-place updating. That is, given an abstract program $f(A_1, \ldots, A_n)$ on abstract arrays $A_1, \ldots, A_n$ that returns an abstract array, we want to translate it to a concrete program F on distributed tensors such that

$$f(V(S_1), \ldots, V(S_n)) = V(F(S_1, \ldots, S_n)), \tag{11}$$

where V is the tensor view function that converts a distributed tensor to an abstract array and S_i is the distributed tensor that implements A_i. This translation is done in two stages. The first stage is to translate array comprehensions to dataset comprehensions that contain in-memory tensor comprehensions, and the second stage is to translate these comprehensions to efficient code (described in Section VII). For the first stage, we identify and handle certain patterns of array comprehensions without a group-by, called regular comprehensions, that can be directly translated to dataset comprehensions with similar traversal patterns (Section VIII-A). Then, we translate array comprehensions with a group-by (Section VIII-B) and, finally, we translate irregular comprehensions (Section VIII-C).

A. Regular Distributed Tensor Comprehensions

We consider a special class of distributed tensor comprehensions that can be directly translated to dataset comprehensions with similar traversal patterns.

Definition 8.1 (distributed tensor comprehension): A distributed tensor comprehension is a comprehension that takes the form $\mathcal{T}^*[\, (\overline{k}, e) \mid \overline{q}\,]$, where $\mathcal{T}^*$ is a distributed tensor constructor, $\text{tensor}*(\overline{d})(\overline{s})$ with $\overline{d}$ dense and $\overline{s}$ sparse dimensions, $\overline{k}$ is the comprehension key, and $\overline{q}$ are the comprehension qualifiers.

Definition 8.2 (key closure): Given a distributed tensor comprehension $\mathcal{T}^*[\, (\overline{k}, e) \mid \overline{q}\,]$, the key closure $keys(\overline{k})$ of the comprehension key $\overline{k}$ is the transitive closure of the set of variables in $\overline{k}$ that also contains all the index variables in the comprehension that are equal to (directly or indirectly) the variables in $\overline{k}$ through equality predicates in $\overline{q}$.

Definition 8.3 (unique index): A unique index in a comprehension $\mathcal{T}^*[\, (\overline{k}, e) \mid \overline{q}\,]$ is either a variable in $\overline{i}$ in a generator $(\overline{i}, v) \leftarrow e$ in $\overline{q}$, where e is an in-memory or a distributed tensor, or the variable i in a generator over a range of integers $i \leftarrow e_1..e_2$ in $\overline{q}$.

In other words, a unique index variable always draws unique values from its generator domain.

Definition 8.4 (regular distributed tensor comprehension): A distributed tensor comprehension $\mathcal{T}^*[\, (\overline{k}, e) \mid \overline{q}\,]$ is regular if:

1) it does not have any group-by qualifier,
2) each generator in $\overline{q}$ is over an in-memory tensor, a distributed tensor, or a range of integers,
3) the comprehension key $\overline{k}$ is a tuple of index variables, and
4) each unique index variable in $\overline{q}$ is a member of the key closure $keys(\overline{k})$.

In other words, a regular distributed tensor comprehension generates unique values for the comprehension key, which means that this comprehension assigns at most one value for each array index in the resulting tensor.

For example, given the distributed tensors X and Y of size d×d, the matrix addition X+Y is:

```
tensor*(d,d)[ ((i,j),x+y) | ((i,j),x) ← X, ((ii,jj),y) ← Y,
                            ii == i, jj == j ];
```
(12)

which is a regular distributed tensor comprehension because the unique indices are $\{i, j, ii, jj\}$ and each one of these is a member of the key closure $keys((i, j)) = \{i, j, ii, jj\}$.

In this section, we translate regular distributed tensor comprehensions to dataset comprehensions, which in turn are optimized to efficient Spark code using the methods in Section VII. Instead of up-coercing each input distributed tensor to an association list of index-value pairs using its view function, we traverse this input distributed tensor as a dataset of tensor blocks. In addition, instead of down-coercing the comprehension result to a distributed tensor using the store function, we construct this result as a dataset of blocks in which each block is constructed directly from the input tensor

blocks using an in-memory tensor comprehension. That way, the resulting comprehension is a dataset comprehension that works on the input tensor blocks directly without looking into their content, only to form the resulting blocks from the input blocks at the head of the RDD comprehension. Such a translation is very efficient because it allows us to process the input tensor blocks without deconstructing them. This method applies only to regular distributed tensor comprehensions.

Our framework is able to translate any regular comprehension. It translates matrix addition given in (12) to the following RDD comprehension:

```
rdd[ ((ci,cj),B) | ((ci,cj),bx) ← X._3, ((cii,cjj),by) ← Y._3,        (13)
                   cii == ci, cjj == cj ],
```

where the term B is the following in-memory tensor comprehension over the tensor blocks bx and by:

```
tensor(N,N)[ ( (i%N,j%N), x+y )
           | ((ti,tj),x) ← bx, let i = ci*N+ti, let j = cj*N+tj,
             ((tii,tjj),y) ← by, let ii = cii*N+tii, let jj = cjj*N+tjj,
             ii == i, jj == j ],
```

where ci, cj are the block coordinates of the block bx in the distributed tensor X and cii, cjj are the block coordinates of the block by in the distributed tensor N. Both bx and by are in-memory tensors of size $N \times N$, where N is the block dimension. The block bx in B is traversed as an association list giving pairs ((ti,tj),m), where ti and tj are in-block indices. Therefore, the array index i is ci*N+ti and the new block coordinate is i%N (which is equal to ti). As we saw in Section VII, the RDD comprehension (13) is translated to a join between the RDDs X._3 and Y._3.

In general, a regular distributed tensor comprehension

$$\mathcal{T}^*[\,(\overline{k}, e) \mid \overline{q}\,],$$

where $\mathcal{T}^*$ is a distributed tensor constructor, $\text{tensor}*(\overline{d})(\overline{s})$ with $\overline{d}$ dense and $\overline{s}$ sparse dimensions, is translated to an RDD comprehension that operates on memory blocks:

$$\text{rdd}[\,(\overline{c_k}, \mathcal{B}[\,(\overline{k}, e) \mid \overline{q''}\,]) \mid \overline{q'}\,]. \tag{14}$$

The resulting RDD consists of in-memory tensor blocks built with the in-memory tensor constructor $\mathcal{B} = \text{tensor}(\overline{N})(\overline{N})$ that has the same shape as $\mathcal{T}^*$ but with all dimensions fixed to N (the block dimensions). The variables names $\overline{c_k}$ in (14) are the block coordinates derived from $\overline{k}$ by renaming each variable i in $\overline{k}$ to c_i, and the qualifiers $\overline{q'}$ and $\overline{q''}$ are derived from $\overline{q}$ using the rules described next. The generators in $\overline{q'}$ traverse the distributed tensors as RDD collections of blocks while the generators in $\overline{q''}$ traverse the blocks as in-memory tensors. A unique index i in $\overline{q}$ is translated to a coordinate index c_i in $\overline{q'}$ and to a block index t_i in $\overline{q''}$. A tensor value v in $\overline{q}$, on the other hand, is processed as a block b_v in $\overline{q'}$. A qualifier from $\overline{q}$ is translated to the following qualifiers in $\overline{q'}$ based on the qualifier type:

1) A generator over a distributed tensor e:
$$(\overline{i}, v) \leftarrow e \quad\longrightarrow\quad (\overline{c_i}, b_v) \leftarrow e.\_3 \tag{15}$$
2) A generator over a range of integers:
$$i \leftarrow e_1..e_2 \quad\longrightarrow\quad c_i \leftarrow e_1/N..e_2/N \tag{16}$$
3) An equality predicate with unique indices i and j:
$$i == j \quad\longrightarrow\quad c_i == c_j \tag{17}$$
4) An equality predicate with a unique index i:
$$i == e \quad\longrightarrow\quad c_i == e/N \tag{18}$$

All other qualifiers are omitted.

A qualifier from $\overline{q}$ is translated to the following qualifiers in $\overline{q''}$ based on the qualifier type:

1) A generator over a distributed tensor e:
$$(\overline{i}, v) \leftarrow e \quad\longrightarrow\quad (\overline{t_i}, v) \leftarrow b_v,\ \overline{\textbf{let } i = c_i * N + t_i} \tag{19}$$
2) A generator over an in-memory tensor e is used as is:
$$(\overline{i}, v) \leftarrow e \quad\longrightarrow\quad (\overline{i}, v) \leftarrow e \tag{20}$$
3) A generator over a range of integers:
$$i \leftarrow e_1..e_2 \quad\longrightarrow\quad t_i \leftarrow 0..(N-1),\ \textbf{let } i = c_i * N + t_i \tag{21}$$
4) Any other predicate is used as is in $\overline{q''}$:
$$q \quad\longrightarrow\quad q \tag{22}$$

For example, for the matrix addition given in (12), the qualifier ((i,j),x) ← X is translated to the qualifier ((ci,cj),bx) ← X._3 in $\overline{q'}$ and to the qualifiers ((ti,tj),x) ← bx, **let** i = ci*N+ti, **let** j = cj*N+tj in $\overline{q''}$, which gives program (13).

Another example is matrix initialization:

```
tensor*(d,d)[ ((i,j),1.0) | i ← 0..(d−1), j ← 0..(d−1) ],
```

which is translated to the following dataset comprehension:

```
rdd[ ( (ci,cj), tensor(N,N)[ ((i%N,j%N),1.0)
                           | ti ← 0..(N−1), let i = ci*N+ti,
                             tj ← 0..(N−1), let j = cj*N+tj ] )
   | ci ← 0..(d−1)/N, cj ← 0..(d−1)/N ],
```

which constructs the matrix in blocks. Here, the qualifier i ← 0..(d−1) is translated to the qualifier ci ← 0..(d−1)/N in $\overline{q'}$ and to the qualifiers ti ← 0..(N−1), **let** i = ci*N+t in $\overline{q''}$.

Theorem 8.1 (correctness): The translation of a regular distributed tensor comprehension satisfies Eq. (11):

$$V((\overline{d}, \overline{s}, \text{rdd}\,[\,(\overline{c_k}, \mathcal{B}[\,(\overline{k}, e) \mid \overline{q''}\,]) \mid \overline{q'}\,])) = [\,(\overline{k}, e) \mid \overline{Q}\,],$$

where V is the tensor view function and the $\overline{Q}$ qualifiers are equal to $\overline{q}$ except for the generators over distributed tensors $(\overline{i}, v) \leftarrow e$, which are translated to $(\overline{i}, v) \leftarrow V(e)$ in $\overline{Q}$.
The proof is given in Appendix A.

B. Tensor Comprehensions with a Group-By

In this section, we translate distributed tensor comprehensions with a group-by that take the form:

$$\mathcal{T}^*[\,(\overline{k}, e) \mid \overline{q},\ \textbf{group by } \overline{k}\,], \tag{23}$$

where the comprehension keys $\overline{k}$ are equal to the group-by keys (or a permutation of them), $\mathcal{T}^*$ is a distributed tensor,

tensor$*(\overline{d})(\overline{s})$, with $\overline{d}$ dense and $\overline{s}$ sparse dimensions, and $\overline{q}$ are the comprehension qualifiers. For example, the group-by distributed tensor comprehension

```
tensor*(d)[ (i,+/m) | ((i,j),m) ← M, group by i ]                    (24)
```

converts the matrix M to a vector by adding together the elements in each row.

Our framework is able to translate any comprehension of the form (23). We describe this translation by showing how the comprehension (24) is translated to a dataset comprehension. Based on Section V, the matrix M in (24) can be stored as a distributed tensor of type block_tensor_2_0[Double], which is equal to ((Int,Int), RDD[((Int,Int),tensor_2_0[Double])]), where tensor_2_0[Double] is an in-memory tensor of type ((Int,Int),Array[Double]), which represents one block of the matrix M. We translate (24) to the following RDD comprehension that constructs a distributed tensor of type block_tensor_1_0[Double]:

```
( d, rdd[ (ci,B) | ((ci,cj),bm) ← M._3 ].reduceByKey(+^b) ),          (25)
```

where the term B is an in-memory tensor comprehension that converts a matrix block bm of type tensor_2_0[Double] to a vector block of type tensor_1_0[Double] by adding the elements of each row:

```
B = tensor(N)[ (ti,+/m) | ((ti,tj),m) ← bm, group by ti ].            (26)
```

The RDD comprehension (25) generates multiple vector blocks bm for each block coordinate ci. All the blocks that correspond to the same coordinate ci are aggregated by the reduceByKey method, which uses the aggregation S1+^{b}S2 that merges two vector blocks S1 and S2:

```
tensor(N)[ (i,s1+s2)  | (i,s1) ← S1, (ii,s2) ← S2, ii == i ].
```

In general, a group-by distributed tensor comprehension of the form $\mathcal{T}^*[\,(\overline{k}, \oplus/w)\mid \overline{q},\ \textbf{group by}\ \overline{k}\,]$ is translated to the RDD comprehension:

$$\text{rdd}[\,(\overline{c_k}, B)\mid \overline{q'}\,].\text{reduceByKey}(\oplus^b),$$

where B is the in-memory comprehension

$$B = \mathcal{B}[\,(\overline{k}, \oplus/w_i)\mid \overline{q''},\ \textbf{group by}\ \overline{k}\,]$$

that builds one block using the in-memory tensor constructor $\mathcal{B}$ as described in Section VIII-A. The qualifiers $\overline{q'}$ and $\overline{q''}$ are derived from $\overline{q}$ as described in Section VIII-A and $\oplus^b$ merges two blocks using $\oplus$:

$$W \oplus^b V = \mathcal{B}[\,(\overline{k}, w \oplus v)\mid (\overline{k}, w) \leftarrow W,\ (\overline{k'}, v) \leftarrow V,\ \overline{k'} == \overline{k}\,].$$

In case of multiple aggregations in the comprehension head, the resulting code must merge blocks separately, using one merge for each aggregation. Consider, for example, the comprehension (24) but with two aggregations in the head:

```
tensor*(d)[ (i,avg(m)) | ((i,j),m) ← M, group by i ],                (27)
```

since avg(m) is equal to (+/m)/(+/m.map(x => 1)), which is the sum of the values in m divided by the size of m. This comprehension is translated to

```
tensor*(d)[ (ci,sb/^b cb)
          | (ci,(sb,cb)) ← (rdd[ (ci,(B,C)) | ((ci,cj),bm) ← M._3 ]
                              .reduceByKey(⊕^b)) ],
```

where the block operation sb/^{b}cb divides the values of the block sb with those of cb cell-wise:

```
sb/^b cb = tensor(N)[ (i,s/c)  | (i,s) ← sb, (ii,c) ← cb, ii == i ].
```

The term B in the head of the RDD comprehension is given in (26) while the term C counts the columns of each row (which is equal to the size of a vector block, N):

```
C = tensor(N)[ (ti,+/v)  | ((ti,tj),m) ← bm, let v = 1, group by ti ].
```

Here, the reduceByKey aggregation is over pairs of blocks, one to aggregate the sums and the other to aggregate the counts:

```
(S1,C1) ⊕^b (S2,C2)
  = ( tensor(N)[ (i,s1+s2)  | (i,s1) ← S1, (ii,s2) ← S2, ii == i ],
      tensor(N)[ (i,c1+c2)  | (i,c1) ← C1, (ii,c2) ← C2, ii == i ] ).
```

In general, let $w_1, \ldots, w_m$ be the occurrences of the lifted variables in e in $\mathcal{T}^*[\,(\overline{k}, e)\mid \overline{q},\ \textbf{group by}\ \overline{k}\,]$. A variable w_i may occur in e as a term that takes one of the following forms:

- $\oplus_i/w_i$, for some monoid $\oplus_i$, or
- $\oplus_i/w_i.\text{map}(g_i)$, for some monoid $\oplus_i$ and a function g_i, or otherwise
- w_i, which is equal to $+\!\!+/w_i.\text{map}(x \Rightarrow List(x))$,

where the last case is used when the other cases do not match. All these cases can be generalized to $\oplus_i/w_i.\text{map}(g_i)$, for some monoid $\oplus_i$ and some function g_i. Therefore, e can be put into a form $f(\overline{\oplus_i/w_i.\text{map}(g_i)})$, for some variables w_i lifted by group-by, some monoids $\oplus_i$, and some functions g_i and f. Then, the group-by comprehension is translated as follows:

$$\mathcal{T}^*[\,(\overline{k}, f(\overline{v}))\mid (\overline{k}, \overline{v}) \leftarrow (\text{rdd}[\,(\overline{c_k}, (B_1, \ldots, B_m))\mid \overline{q'}\,] .\text{reduceByKey}(\oplus^b))\,],$$

where the RDD comprehension followed by the reduceByKey generates multiple blocks $B_1, \ldots, B_m$ for each key $\overline{k}$, one for each aggregation. A block B_i is derived by aggregating the lifted variable w_i using $\oplus_i$:

$$B_i = \mathcal{B}[\,(\overline{k}, \oplus_i/w_i)\mid \overline{q''},\ \textbf{group by}\ \overline{k}\,].$$

Finally, the aggregation $\oplus^b$ used by reduceByKey merges the associated B_i blocks that correspond to the same key:

$$(w_1, \ldots, w_m) \oplus^b (v_1, \ldots, v_m) = (w_1 \oplus_1^b v_1, \ldots, w_m \oplus_m^b v_m),$$

where $w_i \oplus_i^b v_i$ merges the blocks w_i and v_i using $\oplus_i$:

$$\mathcal{B}[\,(\overline{k}, w \oplus_i v)\mid (\overline{k}, w) \Leftarrow w_i,\ (\overline{k'}, v) \Leftarrow v_i,\ \overline{k'} == \overline{k}\,]$$

in which the generators are over the full tensor view to take into account the zero elements when the tensors are sparse. To speed up execution, instead of the latter comprehension, our framework calls a special function merge_tensors$(w_i, v_i, \oplus_i)$ that combines sparse tensors by merging their sorted sparse entries.

C. Irregular Tensor Comprehensions

Consider the following distributed tensor comprehension that rotates the rows of a matrix X so that the first row is moved to the second, the second to third, etc, and the last to the first:

```
tensor*(m,n)[ ( ( (i+1)%m, j ), v ) | ((i,j),v) ← X ].
```
(28)

This comprehension is not a regular distributed tensor comprehension because the comprehension key is not a tuple of index variables (which are i and j). Each block (an in-memory tensor) of X with block coordinates (ci,cj) may be used to construct multiple blocks for the comprehension result. More specifically, the input block with coordinates (ci,cj) is used to construct the blocks with coordinates (ci,cj) and ((ci+1)%(m/N),cj), where m is the number of rows and N is the block dimension.

In general, let us consider an index e in the comprehension head that depends on a number of index variables $i_1 \ldots, i_k$. That is, e can be expressed as $f(i_1, \ldots, i_k)$, which is a term that depends on $i_1 \ldots, i_k$. The block coordinate of this index is e/N. The coordinates of the output blocks that need the input blocks with coordinates $ci_1, \ldots, ci_k$ are all the possible values of $f(ci_1 * N + ti_1, \ldots, ci_k * N + ti_k)/N$ for all in-block indices $ti_1, \ldots, ti_k$ that range between 0 and $N-1$. These output block coordinates are computed using the function unique_values, which calculates the set of block coordinates for any combination of $ti_1, \ldots, ti_k$ values. That is, the output block coordinates for e are

$$\text{uniqueValues}(\ (ti_1, \ldots, ti_k) \Rightarrow f(ci_1{*}N{+}ti_1, \ldots, ci_k{*}N{+}ti_k)/N\ ).$$

Here, we define uniqueValues for $k = 1$ and $k = 2$:

```
def uniqueValues ( f: Int => Int ) = 1.until(N).map(f).toSet
def uniqueValues ( f: (Int,Int) => Int )
      = 1.until(N).zip(1.until(N)).map(f).toSet.
```

As an optimization, if the index e is equal to an input index variable i, then uniqueValues(ti => (ci*N+ti)/N) can be replaced with Set(ci).

Given these block coordinates, the matrix rotation (28) is translated to the following RDD comprehension:

```
rdd[ ( (ki,kj), B )
     | ((ci,cj),bv) ← X._3,
       ki ← uniqueValues( ti => ((ci*N+ti)+1)%m/N ),
       kj ← Set(cj),
       group by (ki,kj)  ],
```

where the term B is the following in-memory tensor that gets all needed blocks from bv, which has been lifted to a list of blocks that contains all the input blocks needed for the output block with coordinates (ki,kj):

```
B = tensor(N,N)[ ( ( (i+1)%m, j ), v )
                 | ((ci,cj),tv) ← bv, ((ti,tj),v) ← tv,
                   let i = ci*N+ti, let j = cj*N+tj,
                   ki == (i+1)%m/N, kj == j/N ].
```

The expression uniqueValues(ti => ((ci*N+ti)+1)%m/N) will calculate the set Set(ci,(ci+1)%(m/N)) so that after the group-by, each coordinate (ki,kj) will be associated with two input blocks in bv. The in-memory tensor B uses the variable bv, which has been lifted to a list of blocks after group-by (a list of two blocks), to construct an output block.

In general, an irregular distributed tensor comprehension with k dimensions takes the form:

$$\mathcal{T}^*[\ ((f_1(\overline{i_1}), \ldots, f_k(\overline{i_k})), e)\ |\ \overline{q}\ ],$$

in which $f_j(\overline{i_j})$ is a term that calculates the jth index of the output tensor that depends on a number of tensor indices $\overline{i_j}$ from $\overline{q}$. It is translated to the following RDD comprehension:

$$\text{rdd}[\ ((key_1, \ldots, key_k), B)\ |\ \overline{q'},\ key_1 \leftarrow u_1, \ldots, key_k \leftarrow u_k,\ \textbf{group by}\ (key_1, \ldots, key_k)\ ],$$

where the qualifiers $\overline{q'}$ are those described in Section VIII-A, the u_j term calculates the unique values for the jth index given by:

$$u_j = \text{uniqueValues}(\ \overline{ti_j} \Rightarrow f_j(\overline{ci_j * N + ti_j})\ )$$

and the term B constructs one block for the output tensor:

$$B = \mathcal{B}[\ (\overline{k}, e)\ |\ \overline{q'''},\ key_1 == f_1(\overline{i_1}), \ldots, key_k == f_k(\overline{i_k})\ ].$$

The qualifiers $\overline{q'''}$ are those for $\overline{q''}$ described in Section VIII-A except the first rule (Rule (VIII-A)), which should now be:

$$(\bar{i}, v) \leftarrow e \quad \longrightarrow \quad \overline{(\overline{c_i}, t_v) \leftarrow b_v,\ (\overline{t_i}, v) \leftarrow t_v,\ \textbf{let}\ i = c_i * N + t_i},$$

which considers all blocks t_v in b_v before they are used to construct an output block.

D. Improving Performance with a Group-By-Join

A *group-by-join* (GBJ) is a join between two distributed datasets followed by a group-by with aggregation. Group-by-joins can be evaluated very efficiently using an algorithm that resembles the optimal distributed block matrix multiplication called SUMMA [19]. Matrix multiplication, defined in (1), is an example of a group-by-join. The GBJ algorithm creates an implicit grid of cells of size $D*D$ where each cell is handled by one processor (in our case, a Spark executor). Hence, if there are M executors available, then $D = \lceil \sqrt{M} \rceil$. Essentially, each block of the two join inputs is replicated D times and shuffled by cogroup to D cells in the grid. The blocks of the resulting matrix in each cell is derived from the shuffled blocks in the cell without having to shuffle any more data. That is, there is no need for a groupBy or reduceByKey after the cogroup. Our framework recognizes a GBJ after a comprehension is translated to Spark operations (using the methods described in Section VIII) by matching the operations with the following pattern:

```
X.map( px => (jx,tx) )
 .join( Y.map( py => (jy,ty) ) )
 .flatMap( prod )
 .reduceByKey( plus ),
```

for some patterns px and py, some terms jx and tx that depend on px, some terms jy and ty that depend on py, and some functions prod and plus. Here, the map on X generates the join key jx (a tensor coordinate) and the block tx (a tensor) from X. Similarly, the map on Y generates the join key jy (a tensor coordinate) and the block ty (a tensor) from Y. After the join (on jx==jy), the two tensors tx and ty are combined using the function prod and then the results are reduced by key (the tensor coordinates) using the function plus. The distributed tensors X and Y can also be complex terms that contain joins. The term above is translated to the following Spark code that implements GBJ:

```
left.cogroup(right)
   .flatMap{ case (_,(xs,ys)) => cell(xs,ys,prod,plus) }
```

where left repeats the X blocks D times (across grid rows) and right repeats the Y blocks D times (across grid columns):

```
left = rdd[ ( ( i, jx%D ), ((jx,gx),tx) ) | px ← X, i ← 0..(D−1) ],
right = rdd[ ( ( jy%D, j ), ((jy,gy),ty) ) | py ← Y, j ← 0..(D−1) ].
```

That is, each block of X and Y is replicated D times and shuffled by cogroup to D cells in the grid. On the other hand, a cross product between X and Y would have shuffled each block from either X or Y, $M = D^2$ times. The blocks of the resulting matrix in each cell can be derived from the shuffled blocks in the cell without having to shuffle any more data. Function cell is evaluated for each grid cell by one executor and performs the group-by-join for the tensors in xs and ys:

```
def cell ( xs, ys, prod, plus ) {
   val H = Map()   /* a hash table that maps (gx,gy) to a tensor */
   for ( ((jx,gx),tx) ← xs; ((jy,gy),ty) ← ys if jx == jy ) {
      val v = prod(tx,ty)        /* a product of tensors */
      val key = (gx,gy)
      H += (key −> if (H.contains(key)) plus(H(key),v) else v
   }
   H.toList }
```

For each cell, it calculates the output blocks that correspond to this cell from the shuffled blocks xs of X and ys of Y.

IX. Performance Evaluation

Our system, called *SAC* (Scalable Array Comprehensions), has been implemented on Apache Spark [5] using Scala's compile-time reflection and macros. The source code of our system is available at https://github.com/fegaras/diablo. It includes the directory benchmarks, which contains all the source files and scripts for running the experiments reported in this section.

We have evaluated the performance of SAC relative to the Spark MLlib [6], which uses the linear algebra library Breeze, and Tensorflow [1]. In our experiments, we used the pure JVM implementation of the Breeze library used in MLlib, instead of a native implementation, such as OpenBLAS. Although there are other linear algebra libraries that support distributed block arrays, MLlib is the closest to our work since it is built on top of Spark and uses a Scala API. The purpose of our evaluations is to show that the performance of our system is on par with a high-performance linear algebra library, thus providing implementation independence and extensibility without sacrificing performance. Of course, we would not expect to beat the performance of a linear algebra library for individual array operations, such as matrix addition and multiplication, since these libraries are hand-written, tuned, and optimized by expert library programmers. Nevertheless, we show that our system can give better performance for complex array programs.

The platform used in our experiments is the XSEDE Expanse cloud computing infrastructure at SDSC (San Diego Supercomputer Center) [35]. Each program was run on a cluster of 5 nodes where each node is equipped with a 128-core AMD EPYC 7742 processor with 2.5 GHz clock speed, 256 GB RAM and 1 TB SSD. The programs were run on Apache Spark 3.1.2 on Apache Hadoop 3.2.2. Each Spark executor was configured to have 12 cores and 24 GB memory. Therefore, there were 10 executors per node, giving a total of 50 executors, from which 1 was reserved.

TABLE I: Real datasets.

Dataset Name	Size	# of Ratings	Sparsity
Netflix_1	$5K \times 50K$	$7M$	97.20%
Netflix_2	$5K \times 480K$	$24M$	99%
Amazon_1	$30K \times 30K$	$100K$	99.99%
Amazon_2	$20K \times 100K$	$160K$	99.99%
Amazon_3	$20K \times 30K$	$100K$	99.98%
Amazon_4	$25K \times 100K$	$250K$	99.99%
Movielens_1	$20K \times 50K$	$7.6M$	99.24%
Movielens_2	$5K \times 20K$	$2.7M$	97.3%
Life_Expectancy	2937×18	N/A	0%
Boston	506×13	N/A	0%
Diabetes	442×10	N/A	0%
California	20640×8	N/A	0%
Cancer	569×30	N/A	0%
Titanic	1309×7	N/A	0%
Credit_card	9708×13	N/A	0%

We evaluated the performance of SAC on five different benchmark programs: matrix addition, matrix multiplication, matrix factorization, linear regression, and neural network. The benchmark programs were implemented using SAC distributed tensors, the MLlib BlockMatrix linear algebra, and the Tensorflow distributed API using the Keras library. In all our experiments, we used blocks of size 1000 for vectors and 1000×1000 for matrices. The datasets used for our evaluations were both real-world and randomly generated datasets. The real datasets used are shown in Table I. We used two different subsets of Netflix Prize Data [21], four different subsets of Amazon Books Reviews [2], and two different subsets of Movielens 25M dataset [26] for Multiplication and Factorization evaluations. For linear regression, we used the Boston housing dataset [32], the California housing dataset [32], the Life expectancy dataset [21], and the Diabetes progression dataset [32]. We used the Titanic Survival dataset [21], the Breast cancer detection dataset [32], and the Credit card approval dataset [21] for binary classification tasks using Neural Networks. Each evaluation was repeated 4 times, from

which the first was ignored to avoid the overhead due to the JIT warm-up. Hence, each data point in the plots in this section represents the mean value of three evaluations. We compared the performance of our system with hand-written programs that call array operations from the MLlib linear algebra library and distributed Tensorflow. As Tensorflow does not support individual distributed array operations, we only compared SAC with Tensorflow for Linear Regression and Neural Network programs.

Matrix Addition: Matrix addition was tested for Dense-Dense, Sparse-Dense and Sparse-Sparse randomly generated matrices. The generated matrices were filled with random values between 0.0 and 10.0 with sparsities: 99% and 90%. The largest matrices used had 100000×100000 elements and size 74.51 GB each and a total size of 149.02 GB. The performance of MLlib and SAC for matrix addition for different sizes of Matrices is plotted in Fig. 4.1- 4.5. We can see from Fig. 4.1 that, for Dense-Dense matrix addition, both SAC and SAC DataFrame APIs performed similar to MLlib. From Fig. 4.3-4.4, for Sparse-Sparse matrix addition, MLlib was faster than SAC, especially for 99% sparsity. However, for Sparse-Dense matrix addition, SAC performed similar to MLlib, as we can see from Fig. 4.2.

Matrix Multiplication: Matrix multiplication was tested for Dense-Dense, Sparse-Dense, and Sparse-Sparse matrices. The synthetic matrices used for multiplication were pairs of square matrices of the same size with 99% sparsity and the real datasets were subsets of the Netflix, Amazon, and Movielens datasets. All generated matrix elements were random values between 0.0 and 10.0. The largest generated matrices used in matrix multiplication had 35000×35000 elements and size 9.13 GB each and the largest real matrices had size of 5000×480000 and $24 * 10^6$ non-zero elements. From Fig. 4.5, we can see that for Dense-Dense multiplication, SAC on Spark Core API was 3× faster compared to MLlib, but the performance of SAC on Dataframes was worse than the other two implementations. For Sparse matrix multiplication, we experimented with both SAC without Group-by-join (GBJ) and SAC with GBJ (an optimization that is described in the extended version of this paper). From Fig. 4.7, for Sparse-Dense multiplication, SAC with GBJ is much faster than SAC without GBJ, while performing similar to MLlib. However, for Sparse-Sparse multiplication, SAC with GBJ performed similar to MLlib while SAC without GBJ was the slowest as we can see from Fig. 4.6 and 4.8.

Matrix Factorization: For matrix factorization, we used the Matrix Factorization algorithm with one iteration using gradient descent. For our experiments, we used the learning rate $a = 0.002$ and the normalization factor $b = 0.02$. The generated matrix to be factorized, R, was a square matrix $n * n$ with random values between 0.0 and 10.0 with sparsity of 99%, 90% and 80%. We also tested Netflix, Amazon and Movielens datasets. The derived matrices P and Q had dimensions $n * k$ and $k * n$, respectively, and were dense matrices initialized with random values between 0.0 and 10.0. The value of k used for this experiment was 10. The largest generated matrix had 40000×40000 elements and size 11.92 GB. The performance of SAC and MLlib matrix factorization experiments is shown in Fig. 4.9-4.12. For 99% sparsity, SAC performed in par with MLlib, but for 90% and 80% sparsities and the real datasets, SAC was about 10× faster than MLlib.

Linear Regression: Linear regression is a supervised machine learning algorithm which calculates a linear model to derive relationship between the input variables (x) and the single output variable (y) and learns to predict y from x. Linear Regression finds a linear model with coefficients $weights = (weights_1, \ldots, weights_m)$ to minimize the cost function, $J = \sum_i (\hat{y}_i - y_i)^2$, between the observed targets in the dataset, and the targets predicted by the model. The cost function J is mean squared error of predictions and it is minimized by using gradient descent. The synthetic datasets were generated using the scikit-learn [32] library with five different data sizes and three different feature sizes. The largest generated dataset used had $2 * 10^5$ data points and each data point had 100 features. From Fig. 4.13-4.16 we can see that, SAC was about 1.5× faster than the hand-written RDD-based MLlib program. We also implemented linear regression using Tensorflow [1] distributed APIs and the Keras library, which performs similarly to SAC for real datasets and synthetic datasets with 10 features as shown in Fig. 4.15-4.16, but performs worse than SAC for synthetic datasets with 50 and 100 features as shown in Fig. 4.13-4.14.

Neural Network: This program computes a binary classification task using an artificial neural network algorithm. Artificial neural networks (ANNs) are comprised of layers of nodes, containing an input layer, one or more hidden layers, and an output layer. The training of ANNs consists of some iterations of the algorithm or iterating until convergence. Each iteration has two parts: forward-pass and backward-pass. In the forward-pass, the network calculates the prediction and in the backward-pass the error is back-propagated to all the layers and the weights are adjusted to minimize the cost function. We chose binary classification because it is a simple yet core application of Artificial Neural Networks. The hidden layers in ANNs actually learn the features from the data and the output layer does the classification. For our binary classification task, we used an input layer, two hidden layers and an output layer. The hidden layers used the sigmoid activation function and the output layer used the RELU activation function for classification. From Fig. 4.17-4.20, we can see that for this experiment, SAC was 3-4× faster for 100 features and 5-6× faster for 50 and 10 features compared to hand-written RDD-based MLlib program. SAC performed 2-3× faster than TensorFlow for real datasets and for synthetic datasets, they performed similarly.

From all these experiments, we can see that the programs generated by SAC have better performance than the hand-written MLlib programs using dense matrices and comparable performance using sparse matrices and SAC even performs better than distributed TensorFlow programs in some cases.

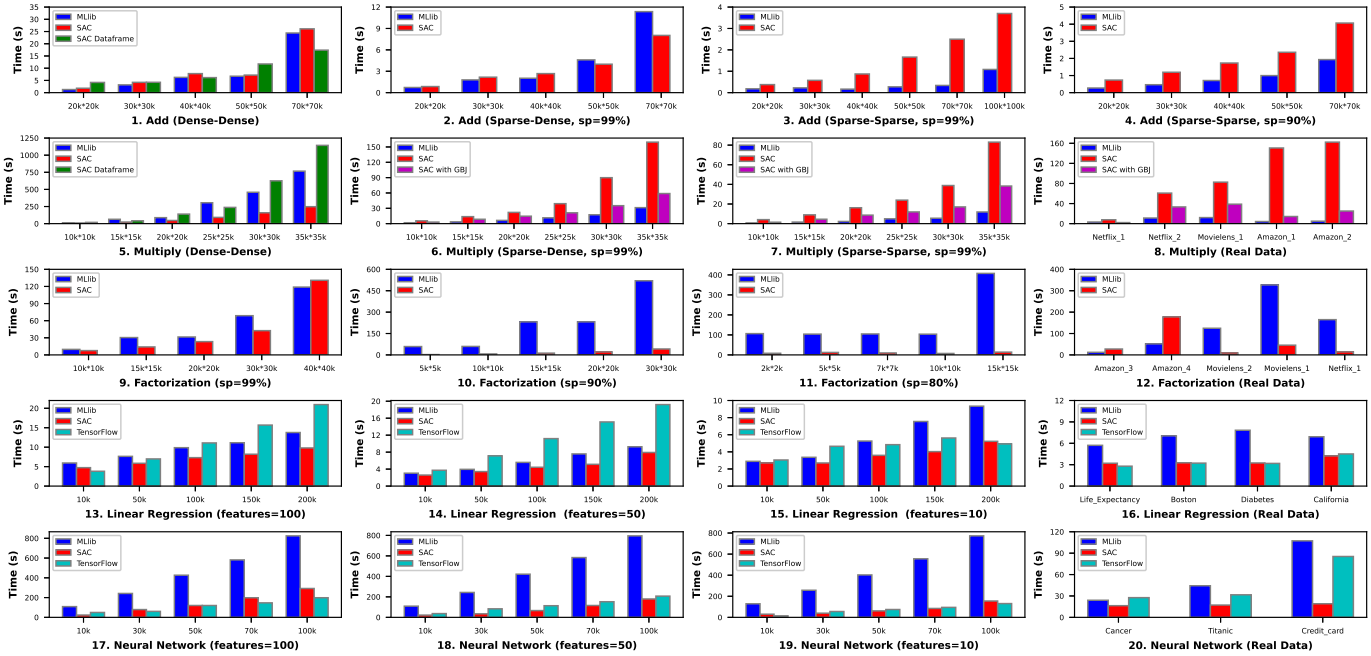

Fig. 4: Performance comparison of SAC with MLlib and Tensorflow

X. Related Work

Many array-processing systems use special storage techniques, such as array tiling, to achieve better performance on certain array computations. TileDB [29] is an array data storage management system that performs complex analytics on scientific data. It organizes array elements into ordered collections called fragments, where each fragment is dense or sparse, and groups contiguous array elements into data tiles of fixed capacity. Unlike our work, the focus of TileDB is on the I/O optimization of array operations by using small block updates to update the array stores. SciDB [34] is a large-scale data management system for scientific analysis based on an array data model with implicit ordering. The SciDB storage manager decomposes arrays into a number of equal sized and potentially overlapping chunks, in a way that allows parallel and pipeline processing of array data. Like SciDB, ArrayStore [33] stores arrays into chunks, which are typically the size of a storage block. One of their most effective storage methods is a two-level chunking strategy with regular chunks and regular tiles. SciHadoop [9] is a Hadoop plugin that allows scientists to specify logical queries over arrays stored in the NetCDF file format. Their chunking strategy, which is called the Baseline partitioning strategy, subdivides the logical input into a set of partitions (sub-arrays), one for each physical block of the input file. SciHive [20] is a scalable array-based query system that enables scientists to process raw array datasets in parallel with a SQL-like query language. SciHive maps array datasets in NetCDF files to Hive tables and executes queries via Map-Reduce. Based on the mapping of array variables to Hive tables, SQL-like queries on arrays are translated to HiveQL queries on tables and then optimized by the Hive query optimizer. SciMATE [38] extends the Map-Reduce API to support the processing of the NetCDF and HDF5 scientific formats, in addition to flat-files. SciMATE supports various optimizations specific to scientific applications by selecting a small number of attributes used by an application and perform data partition based on these attributes. TensorFlow [1] is a dataflow language for machine learning that supports data parallelism on multi-core machines and GPUs but has limited support for distributed computing for certain ML algorithms. Linalg [40] (now part of Spark's MLlib library) is a distributed linear algebra and optimization library that runs on Spark. It consists of fast and scalable implementations of standard matrix computations for common linear algebra operations, such as matrix multiplication and factorization. One of its distributed matrix representations, BlockMatrix, treats the matrix as dense blocks of data, where each block is small enough to fit in memory on a single machine. Linalg allows matrix computations to be pushed from the JVM down to hardware via the Basic Linear Algebra Subprograms (BLAS) interface. SystemML [8] is a machine learning (ML) library built on top of Spark. It supports a high-level specification of ML algorithms that simplifies the development and deployment of ML algorithms by separating algorithm semantics from underlying data representations and runtime execution plans. Distributed matrices in SystemML are partitioned into fixed size blocks, called Binary Block Matrices.

Tensor Comprehensions (TCs) [36] borrow from the Einstein notation to express computations on multi-dimensional arrays without using loops. Basically, a TC is the body of a loop whose control flow is inferred from context. Their system uses a polyhedral compiler as the main optimization

engine that maps tensor comprehensions to high-performance accelerated GPU kernels. Despite their name, TCs are not related to our tensor comprehensions. They are more related to our earlier work on translating of array-based loops to DISC programs [18]. The Tensor Relational Algebra (TRA) [39] is a set of higher-order operations over tensor relations. Each TRA operation takes as input a kernel function defined over multi-dimensional arrays and returns a function over distributed tensor relations. Unlike most distributed data analysis systems on tensors that focus on ML, such as TensorFlow and PyTorch, Tensor Computation Runtimes (TCRs) [22] offer a rich set of operators over tensors that can be used in many other domains, such as graph processing and relational operators. TCRs implement tensor operations using a generic compiler and a runtime system for hardware accelerator. Finally, TVM [11] uses a compiler that exposes graph-level and operator-level optimizations to provide performance for deep learning programs across diverse hardware back-ends. Unlike our work, TVM does not support distributed tensor operations.

Although many of these systems support block matrices, their runtime systems are based on a library of build-in, hand-optimized linear algebra operations, which is hard to extend with new storage structures and algorithms. Furthermore, many of these systems lack a comprehensive framework for automatic inter-operator optimization, such as finding the best way to form the product of several matrices. Like these systems, our framework separates specification from implementation, but, unlike these systems, our system supports ad-hoc operations on array collections, rather than a library of build-in array operations, is extensible with customized storage structures, and uses relational-style optimizations to optimize array programs with multiple operations.

There has also been some recent work on combining linear algebra with relational algebra to let programmers implement ML algorithms on relational database systems [10], [23], [24]. The work by [24], [25] adds a new attribute type to relational schemas to capture arrays that can fit in memory and extends SQL with array operators. Although their system evaluates SQL queries in Map-Reduce, the arrays are not fully distributed. Instead, large matrices must be split into multiple rows as indexed tiles while the programmer is expected to write SQL code to implement matrix operations by correlating these tiles using array operators in SQL. That is, SQL queries on distributed arrays are customizable but the array operators used in correlating tiles are build-in from a library. However, even if these tile operations were customizable, this system does not separate specification from implementation, thus making hard to change the array storage since it would require programmers to write explicit code to correlate tiles for the new storage.

XI. Conclusion

We have presented a flexible and customizable framework for translating array programs to high-performance distributed code. Our performance evaluation results show that SAC is often nearly as efficient as linear algebra and ML libraries, and occasionally outperforms them in complex ML programs. The effectiveness of our framework can be attributed to its effective formal basis, the array comprehensions, which can capture most array operations and are easy to normalize and optimize. Due to the generality of array comprehensions, we were able to solve hard optimization problems using a small number of powerful rules, which are independent of array storage. We are currently extending our framework to generate CUDA code that can run on NVidia GPUs. More specifically, we are translating the functional arguments of the generated Spark Core API methods, which work on array blocks, to GPU kernel code.

Acknowledgments: Our evaluations were performed at the XSEDE Expanse cloud computing infrastructure at SDSC, www.xsede.org, supported by NSF.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

[2] Amazon Books Reviews. https://jmcauley.ucsd.edu/data/amazon, 2022.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A Portable Linear Algebra Library for High-Performance Computers. In *Supercomputing'90*, pages 2-11, 1990.

[4] Apache Flink. http://flink.apache.org/, 2022.

[5] Apache Spark. http://spark.apache.org/, 2022.

[6] Apache Spark MLlib. https://spark.apache.org/mllib/, 2022.

[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi. Spark SQL: Relational Data Processing in Spark. In *ACM SIGMOD International Conference on Management of Data*, pages 1383–1394, 2015.

[8] M. Boehm, M. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. Reiss, P. Sen, A. Surve, and S. Tatikonda. SystemML: Declarative Machine Learning on Spark. *PVLDB*, 9(13):1425–1436, 2016.

[9] J. Buck, N. Watkins, J. Lefevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. A. Brandt. SciHadoop: Array-based Query Processing in Hadoop. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[10] L. Chen, A. Kumar, J. Naughton, and J. M. Patel. Towards linear algebra over normalized data. *Proceedings of the VLDB Endowment (PVLDB)*, 10(11):1214—1225, 2017.

[11] T Chen, T Moreau, Z Jiang, L Zheng, E Yan, M Cowan, H Shen, L Wang, Y Hu, L Ceze, C Guestrin, and A Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[13] J. J. Dongarra, J. D. Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. In *ACM Transactions on Mathematical Software*, 16(1):1—17, 1990.

[14] L. Fegaras. An Algebra for Distributed Big Data Analytics. *Journal of Functional Programming (JFP)*, special issue on Programming Languages for Big Data, volume 27, 2017.

[15] L. Fegaras. A Query Processing Framework for Large-Scale Scientific Data Analysis. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems (TLDKS)*, Springer, July 2018.

[16] L. Fegaras. Scalable Linear Algebra Programming for Big Data Analysis. In *24th International Conference on Extending Database Technology (EDBT)*, 313–324, 2021.

[17] L. Fegaras and M. H. Noor. Compile-Time Code Generation for Embedded Data-Intensive Query Languages. In *IEEE BigData Congress*, 2018.

[18] L. Fegaras and M. H. Noor. Translation of Array-Based Loops to Distributed Data-Parallel Programs. *Proceedings of the VLDB Endowment (PVLDB)*, 13(8): 1248-1260, 2020.

[19] R. A. Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. In *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[20] Y. Geng, X. Huang, M. Zhu, H. Ruan, and G. Yang. SciHive: Array-based query processing with HiveQL. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications (Trustcom)*, 2013.

[21] Kaggle. https://www.kaggle.com, 2022.

[22] D. Koutsoukos, S. Nakandala, K. Karanasos, K. Saur, G. Alonso, and M. Interlandi. Tensors: An abstraction for general data processing. *PVLDB*, 14(10): 1797–1804, 2021.

[23] A. Kunft, A. Alexandrov, A. Katsifodimos, and V. Markl. Bridging the gap: towards optimization across linear and relational algebra. In *ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, pages 1–4, 2016.

[24] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. Scalable linear algebra on a relational database system. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 31(7):1224—1238, 2018.

[25] S. Luo, D. Jankov, B. Yuan, and C. Jermaine. Automatic Optimization of Matrix Implementations for Distributed Machine Learning and Linear Algebra. In *ACM SIGMOD International Conference on Management of Data*, pages 1222—1234, 2021.

[26] Movielens dataset. https://grouplens.org/datasets/movielens/25m, 2022.

[27] M. H. Noor and L. Fegaras. Translation of Array-Based Loops to Spark SQL. *IEEE International Conference on Big Data (BigData'20)*, December 2020.

[28] M. H. Noor and L. Fegaras. Translation of Array-Based Graph Programs to Spark SQL on Block Arrays. *IEEE International Conference on Big Data (BigData'21)*, December 2021.

[29] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson. The TileDB Array Data Storage Manager. *PVLDB*, 10(4):349–360, 2016.

[30] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 8024–8035, 2019.

[31] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A Generic Parallel Collection Framework. In *Euro-Par Parallel Processing*, 2011.

[32] Scikit-learn Datasets. https://scikit-learn.org/stable/datasets/, 2022.

[33] E. Soroush, M. Balazinska, and D. Wang. ArrayStore: A Storage Manager for Complex Parallel Array Processing. In *ACM SIGMOD International Conference on Management of Data*, pages 253–264, 2011.

[34] E. Soroush, M. Balazinska, S. Krughoff, and A. Connolly. Efficient Iterative Processing in the SciDB Parallel Array Engine. In *27th International Conference on Scientific and Statistical Database Management (SSDBM)*, 2015.

[35] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, et al. XSEDE: Accelerating Scientific Discovery. In *Computing in Science & Engineering*, 16(5): 62–74, 2014.

[36] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. ACM Transactions on Architecture and Code Optimization. 16(4), article 38, 2019.

[37] P. Wadler and S. Peyton Jones. Comprehensive Comprehensions (Comprehensions with 'Order by' and 'Group by'). In *Haskell Symposium*, pages 61–72, 2007.

[38] Y. Wang, W. Jiang, and G. Agrawal. SciMATE: A Novel MapReduce-like Framework for Multiple Scientific Data Formats. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2012.

[39] B. Yuan, D. Jankov, J. Zou, Y. Tang, D. Bourgeois, and C. Jermaine. Tensor Relational Algebra for Distributed Machine Learning System Design. *PVLDB*, 14(8): 1338–1350, 2021.

[40] R. B. Zadeh, X. Meng, A. Ulanov, B. Yavuz, L. Pu, S. Venkataraman, E. Sparks, A. Staple, and M. Zaharia. Matrix Computations and Optimization in Apache Spark. In *International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 31–38, 2016.

[41] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

APPENDIX

Proof of Theorem 8.1: Based on Section V, the view function of a distributed tensor X is:

$$V(X) = [\,(\overline{c_i * N + t_i}, v) \mid (\overline{c_i}, a) \leftarrow X.\_3,\ (\overline{t_i}, v) \leftarrow a\,]$$

where $\overline{c_i}$ are tensor coordinates, a is an in-memory tensor, and $\overline{t_i}$ are tensor indices. That is, it traverses the RDD $X.\_3$, deriving the in-memory tensors a with tensor coordinates $\overline{c_i}$, and then it traverses the tensor a as index-value pairs. Hence, we have:

$$\begin{aligned}
&V((\overline{d}, \overline{s}, \mathrm{rdd}\,[\,(\overline{c_k}, \mathcal{B}[\,(\overline{k}, e) \mid \overline{q''}\,])\mid \overline{q'}\,]))\\
&= [\,(\overline{c_i * N + t_i}, v) \mid (\overline{c_i}, a) \leftarrow \mathrm{rdd}\,[\,(\overline{c_k}, \mathcal{B}[\,(\overline{k}, e) \mid \overline{q''}\,]) \mid \overline{q'}\,],\\
&\qquad\qquad (\overline{t_i}, v) \leftarrow a\,]\\
&= [\,(\overline{c_i * N + t_i}, v) \mid \overline{q'}, \mathbf{let}\ (\overline{c_i}, a) = (\overline{c_k}, \mathcal{B}[\,(\overline{k}, e) \mid \overline{q''}\,]),\\
&\qquad\qquad (\overline{t_i}, v) \leftarrow a\,]\\
&= [\,(\overline{c_k * N + t_k}, v) \mid \overline{q'}, (\overline{k}, v) \leftarrow \mathcal{B}[\,(\overline{k}, e) \mid \overline{q''}\,]\,]\\
&= [\,(\overline{c_k * N + t_k}, e) \mid \overline{q'}, \overline{q''}\,]\\
&= [\,(\overline{k}, e) \mid \overline{q'}, \overline{q''}, \overline{\mathbf{let}\ k = c_k * N + t_k}\,].
\end{aligned}$$

We can prove that $\overline{Q} = \overline{q'}, \overline{q''}, \overline{\mathbf{let}\ k = c_k * N + t_k}$ for different qualifier types in $\overline{q}$, as they translated in Section VIII-A. For instance, a generator $(\overline{i}, v) \leftarrow e$ over a distributed tensor e is translated to the generator $(\overline{c_i}, b_v) \leftarrow e.\_3$ in $\overline{q'}$ and to $(\overline{t_i}, v) \leftarrow b_v$, $\overline{\mathbf{let}\ i = c_i * N + t_i}$ in $\overline{q''}$. The same generator is translated to the following generator in $\overline{Q}$:

$$\begin{aligned}
&(\overline{i}, v) \leftarrow V(e)\\
&= (\overline{i}, v) \leftarrow [\,(\overline{c_i * N + t_i}, v) \mid (\overline{c_i}, a) \leftarrow e._3,\ (\overline{t_i}, v) \leftarrow a\,]\\
&= (\overline{c_i}, a) \leftarrow e._3,\ (\overline{t_i}, v) \leftarrow a, \mathbf{let}\ i = c_i * N + t_i,
\end{aligned}$$

which proves the theorem for this qualifier. The proof for the other qualifier types is similar. ■