CONTENTS

Scalable Linear Algebra Programming for Big Data Analysis

Leonidas Fegaras
University of Texas at Arlington
Arlington, Texas
fegaras@cse.uta.edu

ABSTRACT

Arrays are very important data structures for many data-centric and scientific applications. One of the most effective representations of large dense arrays in a distributed setting is a block array, such as a tiled matrix, which is a distributed collection of non-overlapping dense array blocks. Although there are many linear algebra libraries for machine learning that support distributed block arrays and provide an optimal implementation for many array operations, these libraries do not support ad-hoc array programming and customized storage structures. Imperative programs with loops and array indexing, on the other hand, are more powerful as they allow arbitrary array computations but are hard to parallelize and convert to distributed programs.

Our goal is to provide an SQL-like abstraction for data-parallel distributed array computations that is expressive enough to capture a large class of array computations and can be compiled to efficient data-parallel distributed code. Our abstraction is a monolithic array construction in the form of an array comprehension that is as expressive as SQL by supporting a group-by syntax that allows us to capture many array computations in declarative form. We present rules for translating array comprehensions on block arrays to data-parallel distributed code that can run on Apache Spark. We describe a comprehensive set of effective optimizations that can produce very efficient translations, such as the optimal block matrix multiplication algorithm, even though they are oblivious to linear algebra operations. Finally, we justify our claims by evaluating the performance of our generated code on Apache Spark relative to Spark MLlib.

1 INTRODUCTION

Much of the data used in data-centric applications come in the form of arrays, such as vectors, matrices, and tensors. In the early days of numerical computing, most of the array programming was done in an imperative loop-based language, such as Fortran or C, using array indexing to access and update array elements incrementally, one at a time. Although loop-based programs are efficient when they run on a single processor, they are hard to parallelize and reason about. Currently, most array programming is done using vectorization languages, such as MATLAB, R, and NumPy, that allow programmers to write high-level array code that closely resembles mathematical formulas. These languages provide highly tuned array operations that are applied to whole arrays instead of individual elements, thus making loops inessential. Moreover, they hide the implementation details and optimize performance by choosing an implementation (a kernel) for an array operation from a variety of build-in array storages and algorithms. Internally, these languages rely on numerical libraries, such as BLAS [9], for efficient linear algebra computations. These libraries, which are also an integral part of many machine learning (ML) sytems, such as TensorFlow [1], PyTorch, and MLlib [5], implement basic array operations efficiently using multicore parallelism and GPU acceleration. Many array operations provided by the vectorization languages are overloaded to work on a variety of array storage structures, thus offering an implementation-independent view to the programmer. Given that there are numerous storage structures for arrays, such as dense, tiled, and compressed sparse matrices, each library operation must have numerous implementations, especially those operations that operate on multiple arrays, such as matrix multiplication. As a result, this code specialization based on array implementation is hard to extend with user-defined storage structures and algorithms. This is particularly true for distributed arrays, which have to be partitioned into blocks and distributed across compute nodes. In that case, not only there are numerous ways to implement these blocks as arrays, but there are also numerous ways to partition the arrays into blocks. A better solution would have been to express array computations in a high-level declarative language for array computations that is expressive enough to capture most array programs and is supported by a translation scheme that separates the specification from the implementation and generates high-quality code.

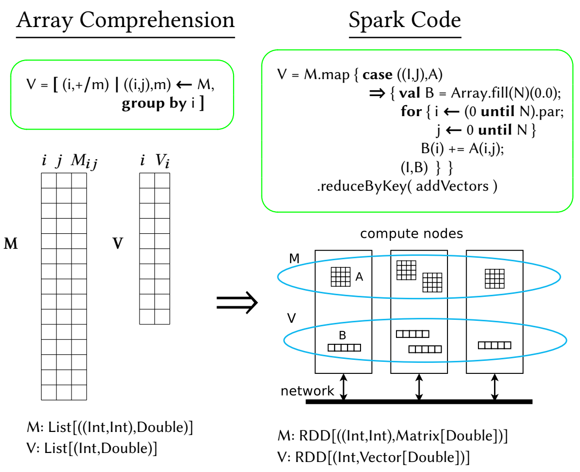
This problem of sacrificing expressiveness and extensibility for efficiency incurred from the library approach is exacerbated by the need to process large arrays that do not fit in memory. Given that the accuracy of the data analysis and ML models depends on the data size, current data-centric applications must analyze enormous amounts of array data using complex mathematical data processing methods. In recent years, new frameworks in distributed Big Data analytics have become essential tools for large-scale machine learning and scientific discoveries. These systems, which are also known as Data-Intensive Scalable Computing (DISC) systems, have revolutionized our ability to analyze Big Data. Unlike High-Performance Computing (HPC) systems, which are mainly designed for shared-memory architectures, DISC systems are distributed data-parallel systems on clusters of shared-nothing computers connected through a high-speed network. Compared to low-level distributed-memory communication paradigms, such as MPI, DISC systems automate many aspects of distributed computing, such as fault tolerance, which is important for long-running Big Data analysis on thousands of computers, scalability, data partitioning and distribution, and task scheduling and management. One of the earliest DISC systems is Map-Reduce [8], which was introduced by Google and later became popular as an open-source software with Apache Hadoop. Recent systems, such as Apache Spark [4] and Apache Flink [3], go beyond Map-Reduce by maintaining dataset partitions in the memory of the compute nodes. All these systems use data shuffling to exchange data among compute nodes, which takes place implicitly between the map and reduce stages in Map-Reduce and during group-bys and joins in Spark and Flink. Essentially, all data exchanges across compute nodes are done in a controlled way using special operations, which implement data shuffling by distributing data based on some key, so that data associated with the same key are processed together by the same compute node.

The goal of this paper is to provide a well-formed abstraction for data-parallel distributed array computations "without regret", that is, an abstraction that is declarative so that we can reason about it, is expressive enough to capture a large class of array computations, and can be compiled to efficient data-parallel distributed code. Our main construct is the *array comprehension*, which is a monolithic array construction in the form of a list comprehension. List comprehensions are found in many modern programming languages, such as Python, Scala, and Haskell. Unlike regular list comprehensions though, our array comprehensions are as expressive as SQL queries by supporting a group-by syntax that allows us to capture many array computations in declarative form without using array indexing which is hard to reason about. An array comprehension can access and correlate multiple arrays by traversing their elements one-by-one and can construct a new array in one shot by mapping array indices to values, which are derived from the elements of the input arrays. 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 and concatenation. More complex array operations, such as matrix inverse and LU decomposition, can be coded using array comprehensions inside loops.

1.1 Highlights of our Approach

This paper presents a generic and customizable system that translates abstract array programs to high-performance distributed code that can run on current DISC platforms. 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. Our framework uses a two-layer approach where array programs are expressed in a powerful high-level syntax on abstract arrays, while these abstract arrays are mapped to customized storage structures based on user-defined type mappings, thus separating specification from implementation.

An abstract array with dimensionality i in our framework 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 represented as an association list of key-value pairs in which the key contains 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 (i.e., some elements may be missing) if the array is sparse. A concrete implementation of an array (i.e., its storage structure) is specified by two customized functions: the *sparsifier*, which converts the storage structure to an association list, and the *builder*, which constructs the storage structure from the association list. These two functions, which are inverse of each other, are used by our translator to transform any operation $f(x_1, \ldots, x_n)$ on abstract arrays x_i to an operation on their concrete storage structures c_i by up-coercing the storages c_i using the sparsifiers s_i and down-coercing the abstract result using the builder b, that is,



```
V = [ (i,+/m) | ((i,j),m) ← M,
                group by i ]
```

```
V = M.map { case ((I,J),A)
              ⇒ { val B = Array.fill(N)(0.0);
                  for { i ← (0 until N).par;
                        j ← 0 until N }
                     B(i) += A(i,j);
                  (I,B) } }
       .reduceByKey( addVectors )
```

M: List[((Int,Int),Double)]
V: List[(Int,Double)]

M: RDD[((Int,Int),Matrix[Double])]
V: RDD[(Int,Vector[Double])]

Figure 1: Code generation for $V_i = \sum_j M_{ij}$

$b(f(s_1(c_1), \ldots, s_n(c_n)))$. This layered approach introduces levels of indirection and generates superfluous intermediate structures (the abstract arrays x_i) that need to be removed. In our framework, this is accomplished by fusing these functions into one function that represents the concrete code so that the resulting program builds the output structures directly and works on the storage structures c_i without creating the association lists x_i.

Our language for expressing abstract array programs uses a monolithic array construction in the form of an array comprehension that is as expressive as SQL by supporting a group-by syntax that allows us to capture many array computations in declarative form. Comprehensions with a group-by syntax were first introduced by Wadler and Peyton Jones [24] and have been used as the formal calculus for the DISC query languages MRQL [10] and DIQL [12]. For example, the following comprehension returns the number of employees in each department:

```
[ (d.name,count(e)) | e ← Employees, d ← Departments,
                  e.dno == d.dnumber, group by d.name ].
```

The formal semantics of comprehensions, the translation of comprehensions to an algebra, and a query optimization framework are described in our earlier work [10, 12]. Array comprehensions are actually list comprehensions in which the array storages used in a comprehension are implicitly converted to association lists by array sparsifiers and the list returned by the comprehension is converted to an array storage by an array builder.

Consider for example the following array comprehension that constructs a vector V of size n from a matrix M of size $n \times m$ such that $V_i = \sum_j M_{ij}$, where both the matrix M and the resulting vector V are stored in memory:

```
V = vector(n)[ ( i, +/m ) | ((i,j),m) ← M, group by i ],        (1)
```

which constructs the entire array V in one shot. The matrix M of type matrix[Double] is implicitly converted to an association list of type List[((Int,Int),Double)]. The generator ((i,j),m) ← M 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 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 bag of t, indicating that each such variable must now contain all the values associated with the same group-by key value. Consequently, after we group by i, the pattern variables that are not used in the group-by key, j and m, are now 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 calculates $\sum_j M_{ij}$. Finally, the array builder, vector(n)(L), converts the association list L of type List[(Int,Double)] to a vector of type vector[Double] of size n.

In Query (1), both matrix M and the query result V are stored in memory. However, in our framework, the storage of these arrays can be customized by using a different sparsifier for M and a different builder for the result. More specifically, we want to translate array comprehensions to efficient distributed programs over block arrays, which are distributed bags of dense array chunks. In Spark [4], we can implement a tiled matrix as a distributed collection (an RDD) of fix-sized square tiles of type RDD[((Int,Int),Array[Double])], where each tile ((i,j),A) has coordinates i and j and values stored in the dense matrix A, which has a fixed size N*N, for some constant N. Similarly, we can implement the query result V as a block vector of type RDD[(Int,Array[Double])], where each block (i,B) has a coordinate i and values stored in the vector B of size N. Then, Query (1) can be rewritten as:

V = tiled[(i, +/m) | ((i,j),m) ← M, **group by** i]. (2)

For this query, the implicit sparsifier that converts a tiled matrix M of type RDD[((Int,Int),Array[Double])] to an association list of type List[((Int,Int),Double)] is:

```
[  ( ( ii*N+i, jj*N+j ), A(i,j) )
|  ((ii,jj),A) ← M, i ← 0 until N, j ← 0 until N ],
```

where the index variable i in 'i ← 0 **until** N' iterates from 0 to N−1. Here, ii and jj are tile coordinates, and i and j are indices within a tile. That is, for each tile A with coordinates ii and jj read from the tiled matrix M, this list comprehension constructs $N * N$ elements from the data stored in A, so that each element has row coordinate ii*N+i, column coordinate jj*N+j, and value A(i,j). On the other hand, the builder tiled(L), which constructs a block array of type RDD[(Int,Array[Double])] from the list L of type List[(Int,Double)], is:

```
rdd[  ( i/N, vector(N)(w) )
    |  (i,v) ← L, let w = ( i%N, v ), group by i/N ],
```

where the builder rdd builds an RDD from a list. That is, this comprehension groups the elements (i,v) of L by i/N (the tile coordinate) so that all N pairs (i,v) with the same i/N go into the same tile. After the group-by, the value of w is lifted to a list that contains all the values (i%N,v) that belong to the same tile at location i/N. The builder vector(N)(w), used in Query (1), converts the list w to a vector, which is an array block of size N.

Like array queries, sparsifiers and builders are expressed as comprehensions, but unlike array queries, they use efficient array indexing. This implicit coercion from stored arrays to lists and the building of stored arrays from the results of a comprehension introduce levels of indirection and generate superfluous intermediate structures that need to be removed. This is done by unnesting nested comprehensions into flat comprehensions. As we will show in this paper, after some simple transformations, Query (2) is optimized to the Spark code shown on the right of Figure 1. That is, from every tile A in M with tile coordinates I and J, a new vector block B with a coordinate I is constructed. The tile processing, which takes place at each compute node, is parallelized using the the Scala's par method [20]. Finally, vector blocks with the same coordinate are reduced pairwise using vector addition, addVectors.

In this paper, we present a small set of generic rules for translating array comprehensions to efficient Spark RDD programs that operate on block arrays. These rules are not based on specific array operations but can apply to many array processing programs that can be expressed as array comprehensions. Matrix multiplication, for example, is translated to an optimal block matrix multiplication algorithm by one generic rule that recognizes a certain class of group-by-joins (joins between two datasets followed by a group-by and an aggregation), and translates them to a special block group-by-join algorithm. Our system, called *SAC* (Scalable Array Comprehensions), has been implemented using Scala's compile-time reflection and macros. It translates array comprehensions to Scala code that calls Spark RDD operations whose functional arguments use the Scala's Parallel Collections library for multicore parallelism [20].

In an earlier work [13], we have presented a framework, called DIABLO (a Data-Intensive Array-Based Loop Optimizer), for translating array-based loops to array comprehensions, which in turn are translated to Spark programs expressed in the Spark Core API. The DIABLO input language resembles the syntax of some loop-based imperative languages, such as C and Java. DIABLO can translate any array-based loop expressed in this loop-based language to an equivalent Spark program as long as this loop satisfies some simple syntactic restrictions, which are more permissive than the recurrence restrictions imposed by many current systems. Unlike SAC, DIABLO generates Spark programs that operate on arrays in the coordinate format. In a distributed setting, arrays stored in the coordinate format are known to be less efficient than block arrays because 1) they occupy more space and therefore require more data shuffling to evaluate complex array operations, and 2) they are less amenable to multicore parallelism at each compute node since they store the array elements in random order. Furthermore, the focus of DIABLO is in the translation of imperative programs to array comprehensions, while the focus of SAC is in the translation of array comprehensions to efficient code on customizable array storages, with an emphasis on block arrays. That is, SAC supplements DIABLO and can be used as a drop-in back-end replacement for DIABLO to make it able to work on block arrays. Finally, the work reported in this paper generalizes our earlier work on extending MRQL with array-based computations [11]. It extends the MRQL query optimizer with a GroupByJoin physical operator that generalizes the SUMMA parallel algorithm for matrix multiplication [14], an idea also used in Section 5.4 in the context of block arrays. However, unlike our current framework, this system too is based on arrays stored in the coordinate format.

The contributions of this paper are summarized as follows:

- We introduce a novel comprehension syntax for array computations that can capture many array computations in declarative form and is independent of array storage.
- We describe a translation scheme that translates array comprehensions to efficient imperative programs with memory effects (Sections 2 and 3).
- We extend this translation scheme to generate efficient Spark code from array comprehensions (Section 4).
- We introduce special type transformations to translate comprehensions on block arrays to efficient Spark code that reduces the amount of data shuffling (Section 5).

Expression:			
e	$::=$	$[\, e \mid \overline{q} \,]$	comprehension
	$\mid$	$\oplus/e$	reduction using $\oplus$
	$\mid$	$v[e_1, \dots, e_n]$	array indexing $n \geq 1$
	$\mid$	$\dots$	other expression
Qualifiers:			
$\overline{q}$	$::=$	$q_1, \dots, q_n$	$n \geq 0$
Qualifier:			
q	$::=$	$p \leftarrow e$	generator
	$\mid$	$\mathbf{let}\, p = e$	local declaration
	$\mid$	e	filtering
	$\mid$	$\mathbf{group\ by}\, p\ [\, : e\,]$	group-by
Pattern:			
p	$::=$	v	pattern variable
	$\mid$	$(p_1, \dots, p_n)$	tuple $n \geq 0$

Figure 2: Language syntax

$$[\, e_1 \mid p \leftarrow e_2, \overline{q} \,] = e_2.\text{flatMap}(\lambda p.\, [\, e_1 \mid \overline{q} \,]) \quad (4)$$

$$[\, e_1 \mid \mathbf{let}\, p = e_2, \overline{q} \,] = \mathbf{let}\ p = e_2\ \mathbf{in}\ [\, e_1 \mid \overline{q} \,] \quad (5)$$

$$[\, e_1 \mid e_2, \overline{q} \,] = \mathbf{if}\ (e_2)\ \mathbf{then}\ [\, e_1 \mid \overline{q} \,]\ \mathbf{else}\ \text{Nil} \quad (6)$$

$$[\, e \mid \,] = [\, e \,] \quad (7)$$

Figure 3: Desugaring rules

- We evaluate the performance of our system relative to Spark's MLlib.linalg library (Section 6). Based on these results, SAC is up to 6 times faster than MLlib for matrix multiplication and up to 3 times faster than MLlib for matrix factorization.

2 SYNTAX AND SEMANTICS OF ARRAY COMPREHENSIONS

Figure 2 describes the syntax of our language and Figure 3 gives the desugaring rules, which are based on standard methods for translating list comprehensions [23]. The meaning, desugaring rules, and code generation for the group-by syntax are given in Section 3.

Flattening nested comprehensions that do not have a group-by qualifier is done using the following rule:

$$[\, e_1 \mid \overline{q_1}, p \leftarrow [\, e_2 \mid \overline{q_3} \,], \overline{q_2} \,] = [\, e_1 \mid \overline{q_1}, \overline{q_3}, \mathbf{let}\, p = e_2, \overline{q_2} \,] \quad (3)$$

for any sequence of qualifiers $\overline{q_1}$, $\overline{q_2}$, and $\overline{q_3}$. It may require renaming the variables in $[\, e_2 \mid \overline{q_3} \,]$ before we apply this rule to prevent variable capture.

As explained in Section 1.1, abstract arrays in our framework are represented as association lists that uniquely map array indices to values. These abstract representations are mapped to concrete storage structures with the help of a pair of customized functions, a sparsifier and a builder. Then, array comprehensions on abstract arrays are translated to efficient concrete programs on storage structures based on these type mappings. In this and the following section, we describe this program translation in more detail using one specific type mapping that stores a matrix in a flat vector in row-major order. Although this storage structure is not a distributed tiled array, which is the focus of this paper, this example is important for two reasons: First, it illustrates our program translation process in detail using a simpler storage. Second, it is useful for translating comprehensions on block arrays to Spark code (described in Sections 4 and 5) because it shows how the code for tile operations is generated, given that a tile is a matrix.

Consider a matrix M of type Matrix[T] stored in row-major order as a triple (n,m,V) of type (Int,Int,Array[T]), where n and m are the matrix dimensions and V is the vector that contains the matrix elements in row-major order. The following sparsifier converts the storage S of type (Int,Int,Array[T]) to the abstract representation of the matrix M, which is of type List[((Int,Int),T)]:

```
def sparsify[T] ( S: (Int,Int,Array[T]) ): List[((Int,Int),T)]
= [ ((i,j),A(i*n+j)) | let (n,m,A) = S, i ← 0 until n, j ← 0 until m ],
```

where A(i) is array indexing in Scala. The builder matrix(n,m) L takes two groups of parameters. The n and m parameters specify the matrix dimensions, while L is the association list to be converted to a flat vector that contains the matrix values in row-major order:

```
def matrix[T] ( n: Int, m: Int )
              ( L: List[((Int,Int),T)] ): (Int,Int,Array[T])
  = {  val V = Array.ofDim[T](n*m);
       [ V(i*n+j) = v | ((i,j),v) ← L, i≥0, i<n, j≥0, j<m ];
       (n,m,V)  },
```

where Array.ofDim[T](n) creates a new array of size n and V(i) = a is an assignment that updates V(i). The sparsifier function is always named 'sparsify' because, as we will see next, it is implicitly embedded in the code by the compiler by looking at the code type, while the builder must have a unique name or type signature since all builders transform association lists. Nevertheless, the builder too can be inferred by the compiler in certain assignments, such as in the following example. In the following declaration:

```
var M: matrix[Double]
  = matrix(n,m)[ ((i,j),random()) | i ← 0 until n, j ← 0 until m ];
```

the builder matrix(n,m) is required because the M declaration specifies the abstract type, matrix[Double], but not the storage. However, in the following assignment:

```
M = [ ((j,i),m+1) | ((i,j),m) ← M, m > 10 ];
```

the builder can be inferred to be matrix(n,m), since the compiler can infer the storage type of M.

As a running example to illustrate code generation, consider the addition of two matrices M and N of size $n \times m$, expressed as follows using array comprehensions:

```
matrix(n,m)[ ((i,j),a+b) | ((i,j),a) ← M, ((ii,jj),b) ← N,          (8)
                          ii == i, jj == j ].
```

This query can also be expressed as:

```
matrix(n,m)[ ((i,j),a+N[i,j]) | ((i,j),a) ← M ],
```

which is translated to Query (8). Basically, an array indexing $V[e_1, \dots, e_n]$ in a comprehension is transformed by adding the qualifiers $((k_1, \dots, k_n), k_0) \leftarrow V, k_1 == e_1, \dots, k_n == e_n$ to the comprehension, where $k_0, k_1, \dots, k_n$ are fresh variables, and by replacing $V[e_1, \dots, e_n]$ with k_0. Without such translation, array indexing would not be able to map to operations on the underlying array storage.

Given a generator $p \leftarrow e$ in a comprehension, the compiler will infer the type of e using standard type inference. Then, it will search all defined sparsifiers to find one, if exists, that applies to the type of e, and will embed this sparsifier by replacing e with sparsify(e). For Query (8), the compiler will infer the storage type of M and N to be (Int,Int,Array[Float]) and then will embed the right sparsifiers to convert them to association lists, which for these matrices is the sparsify function defined earlier. Then, it will inline the code of the sparsifiers and builder and will optimize the resulting program. This can be done effectively when these functions are expressed as comprehensions. By expressing these functions as comprehensions, the optimizer can fuse them with the array comprehension of the query, resulting to a comprehension that traverses the array storage directly, without creating the intermediate lists. Furthermore, unlike array comprehensions, these functions can and must use array indexing so that the fused comprehension results to efficient array operations.

In addition to flattening nested comprehensions using Rule (3), the only optimizations needed are those related to index traversals. More specifically, if two index generators i ← 0 **until** n and j ← 0 **until** m are related with i==j, then they are fused to one generator and a let-binding: i ← 0 **until** min(n,m), **let** j = i.

For Query (8), if for simplicity, the inequalities in the matrix(n,m) builder are ignored, we have:

```
matrix(n,m)[ ((i,j),a+b) | ((i,j),a) ← sparsify(M),
                           ((ii,jj),b) ← sparsify(N),
                           ii == i, jj == j ]
      (if we inline the array builder without the inequalities)
= {  val V = Array.ofDim[T](n*m);
     [  V(i*n+j) = v
     |  ((i,j),v) ← [ ((i,j),a+b) | ((i,j),a) ← sparsify(M),
                                    ((ii,jj),b) ← sparsify(N),
                                    ii == i, jj == j ] ];
     (n,m,V)  }
      (if we unnest the comprehension using Rule (3))
= {  val V = Array.ofDim[T](n*m);
     [  V(i*n+j) = v
     |  ((i,j),a+b) | ((i,j),a) ← sparsify(M),
                      ((ii,jj),b) ← sparsify(N),
                      ii == i, jj == j ];
     (n,m,V)  }
      (if we inline the sparsifiers and rename their variables)
= {  val V = Array.ofDim[T](n*m);
     [  V(i*n+j) = v
     |  ((i,j),a+b) | ((i,j),a) ← [ ((i1,j1),A(i1*n1+j1))
                                  | let (n1,m1,A) = M,
                                    i1 ← 0 until n1,
                                    j1 ← 0 until m1 ],
                      ((ii,jj),b) ← [ ((i2,j2),B(i2*n2+j2))
                                    | let (n2,m2,B) = N,
                                      i2 ← 0 until n2,
                                      j2 ← 0 until m2 ],
                      ii == i, jj == j ];
     (n,m,V)  }
      (if we unnest the comprehension using Rule (3))
= {  val V = Array.ofDim[T](n*m);
     [  V(i*n+j) = v
     |  let (n1,m1,A) = M,
        i1 ← 0 until n1, j1 ← 0 until m1,
        let (n2,m2,B) = N,
        i2 ← 0 until n2, j2 ← 0 until m2,
        i2 == i1, j2 == j1,
        let ((i,j),v) = ((i1,j1),A(i1*n1+j1)+B(i2*n2+j2))  ];
     (n,m,V)  }
      (if we merge the array index bounds)
= {  val V = Array.ofDim[T](n*m);
     [  V(i*n+j) = A(i1*n1+j1)+B(i1*n2+j1))
     |  let (n1,m1,A) = M, let (n2,m2,B) = N,
        i1 ← 0 until min(n1,n2), j1 ← 0 until min(m1,m2)  ];
     (n,m,V)  }
```

Given that comprehensions over arrays are translated to array index traversals of type scala.collection.immutable.Range in Scala, to parallelize the code, the only transformation needed is to convert the outer index traversal to a parallel traversal of type scala.collection.parallel.immutable.ParRange. This is done by applying par, such as i1 ← (0 **until** min(n1,n2)).par in our example.

Comprehensions can also be used along with total aggregations, such as for checking whether a vector V is sorted:

```
&&/[ v <= w | (i,v) ← V, (j,w) ← V, j == i+1 ],
```

which checks if all consecutive elements of V (i.e., V_i and V_{i+1}) are ordered. The builder of $\oplus/e$ is:

```
{ var b = 1⊕; [ b = (b ⊕ v) | v ← e ]; b },
```

where $\mathbf{1}_\oplus$ is the zero value of the monoid $\oplus$. Similar to the matrix sparsifier, the vector sparsifier is:

```
def sparsify[T] ( V: Array[T] ): List[(Int,T)]
  = [ (i,V(i)) | i ← 0 until V.length ].
```

If we embed the sparsifiers, unfold the builder code, and unnest the list comprehensions, the array comprehension becomes:

```
{  var b = true;
   [ b = (b && (V(i) <= V(j))) | i ← 0 until V.length,
                                 j ← 0 until V.length, j == i+1 ];
   b },
```

which is further optimized to:

```
{  var b = true;
   [ b = (b && (V(i) <= V(i+1))) | i ← 0 until V.length−1 ];
   b },
```

given that min(V.length,V.length−1) = V.length−1.

3 COMPREHENSIONS WITH A GROUP-BY

Consider the product of two matrices M and N with dimensions $n \times l$ and $l \times m$, respectively, which is equal to a matrix C so that $C_{ij} = \sum_k M_{ik} * N_{kj}$. Using our array comprehensions enhanced with a group-by syntax, matrix multiplication can be expressed as follows:

```
matrix(n,m)[ ((i,j),+/v) | ((i,k),a) ← M, ((kk,j),b) ← N,
                           kk == k, let v = a*b,              (9)
                           group by (i,j) ].
```

This comprehension retrieves the values M_{ik} and N_{kj} and sets $v = M_{ik} * N_{kj}$. After we group the values by the matrix indices i and j, the variable v is lifted to a bag of numerical values $M_{ik}*N_{kj}$, for all k. Hence, the aggregation $+/v$ sums up all the values in the bag v, deriving $\sum_k M_{ik} * N_{kj}$ for the ij element of the resulting matrix.

Another example of a group-by comprehension is matrix smoothing for a matrix M, which is a matrix C such that C_{ij} =

$\frac{1}{9}\sum_{i-1\leq I\leq i+1}\sum_{j-1\leq J\leq j+1} M_{IJ}$. That is, C_{ij} is the average value in the neighborhood of M_{ij}:

```
matrix(n,m)[ ((ii,jj),(+/a)/a.length)
             | ((i,j),a) ← M,
               ii ← (i−1) to (i+1), jj ← (j−1) to (j+1),
               ii >= 0, ii < n, jj >= 0, jj < m,
               group by (ii,jj)  ],
```

which also takes care of the boundary cases.

Given a pattern p that consists of bound pattern variables, the qualifier **group by** p in $[\, e \mid \overline{q_1},\ \textbf{group by}\, p,\ \overline{q_2}\,]$ groups every pattern variable in $\overline{q_1}$ (except the variables in p) by the group-by key p into a list that contains all the values of this variable associated with this group-by key. Furthermore, the qualifier **group by** $p : e$ is syntactic sugar for the qualifiers **let** $p = e$, **group by** p. Group-by qualifiers may appear in multiple places in a comprehension. For all these cases, only the pattern variables that precede the group-by qualifier in the same comprehension must be lifted to lists, and if multiple group-by qualifiers exist, these variables will have to be lifted multiple times to nested lists.

Group-by qualifiers are translated to groupBy operations before a comprehension is translated by the rules in Figure 2. Given a list s of type $\text{List}[(K,V)]$, the operation groupBy(s) groups the elements of s by their first component (the group-by key) and returns an association list of type $\text{Map}[K,\text{List}[V]]$, which is implemented as a hash table:

```
def groupBy[K,V] ( s: List[(K,V)] ): Map[K,List[V]]
  = {  val m = Map[K,List[V]]();
       [ m(k) = if (m.contains(k)) m(k):+v else List(v)          (10)
       | (k,v) ← s ];
       m  }.
```

Let $\overline{v} = (v_1,\ldots,v_n)$ be the pattern variables in the sequence of qualifiers $\overline{q_1}$ that are used in the rest of the comprehension $[\, e \mid \overline{q_2}\,]$ but do not appear in the group-by pattern p. Then, the group-by syntax is translated as follows:

$$\begin{aligned}&[\, e \mid \overline{q_1},\ \textbf{group by}\, p,\ \overline{q_2}\,]\\ &\quad = [\, e \mid (p,s) \leftarrow \text{groupBy}([\,(p,\overline{v}) \mid \overline{q_1}\,]),\\ &\qquad\qquad \textbf{let}\ \overline{v} = \text{unzip}(s),\ \overline{q_2}\,],\end{aligned} \tag{11}$$

where $\text{unzip}(s) = ([\, v_1 \mid \overline{v} \leftarrow s\,],\ldots,[\, v_n \mid \overline{v} \leftarrow s\,])$. That is, each pattern variable v_i in $\overline{v}$ is lifted to a list that contains all the values of v_i in the current group.

For example, the matrix multiplication in Query (9) has the following meaning:

```
matrix(n,m)[ ((i,j),(+/v)) | ((i,j),s) ← groupBy(S), let v = s ],
```

where S is:

```
[ ((i,j),v) | ((i,k),a) ← M, ((kk,j),b) ← N, kk == k, let v = a*b ]
```

since the only variable lifted is v with v = [v | v ← s], which is equal to s.

Array comprehensions with a group-by syntax allow more array operations to be expressed declaratively without having to use array indexing and loops. They also make comprehensions equivalent to basic SQL queries. As we will show, although it has pure semantics, the group-by syntax makes it easier to recognize certain code patterns in a comprehension and translate them to efficient code. For example, we will see next that the matrix multiplication in Query (9) is translated to the following code, which is as efficient as a program hand-coded in an imperative language:

```
{  val V = Array.ofDim[T](n*m)(0.0);
   [ V(i*n+j) += A(i*n+k)*B(k*l+j)
   | let (n,l,A) = M, let (ll,m,B) = N,
     i ← 0 until n, k ← 0 until l, j ← 0 until m ];
   (n,m,V)  },
```

which is equivalent to a triple loop with body V_{ij} += $A_{ik} \times B_{kj}$.

Consider the general comprehension $[\, e \mid \overline{q_1},\ \textbf{group by}\ p,\ \overline{q_2}\,]$. To simplify our translation rules, we rewrite this term to $[\, z \mid \overline{q_1},\ \textbf{group by}\ p,\ z \leftarrow [\, e \mid \overline{q_2}\,]\,]$. Let $\mathcal{V} = \{v_1,\ldots,v_n\}$ be the set of variables that are lifted by the group-by but are not lifted or redefined in $\overline{q_2}$ and let $\overline{v} = (v_1,\ldots,v_n)$. Recall that these variables are lifted to lists that contain all the values of these variables associated with the group-by key p. A lifted variable may occur any number of times in the rest of the comprehension, $[\, e \mid \overline{q_2}\,]$. Let $w_1,\ldots,w_m$ be the occurrences of the lifted variables in $[\, e \mid \overline{q_2}\,]$. A lifted variable $w_i \in \mathcal{V}$ may occur in $[\, e \mid \overline{q_2}\,]$ 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 first two do not match. All these cases can be generalized to $\oplus_i/w_i.\text{map}(g_i)$, for some monoid $\oplus_i$ and function g_i. Hence, we can represent the term after group-by as follows:

$$\begin{aligned}[\, e \mid \overline{q_2}\,] &= f(\oplus_1/w_1.\text{map}(g_1),\ldots,\oplus_n/w_m.\text{map}(g_m))\\ &= f(\otimes/\text{zip}(\overline{w}).\text{map}(g)),\end{aligned}$$

for some term function f, where $\overline{w} = (w_1,\ldots,w_m)$, $g = g_1 \times \cdots \times g_m$, and $\otimes = \oplus_1 \times \cdots \times \oplus_m$ (a product of monoids[1]). Then, based on the Rule (11) and the implementation of groupBy in definition (10), we have:

$$\begin{aligned}&[\, e \mid \overline{q_1},\ \textbf{group by}\ p,\ \overline{q_2}\,]\\ &= [\, z \mid \overline{q_1},\ \textbf{group by}\ p,\ z \leftarrow f(\otimes/\text{zip}(\overline{w}).\text{map}(g))\,]\\ &= \{\ \textbf{val}\ M = \text{Map}();\\ &\qquad [\, M(p) = \textbf{if}\ (M.\text{contains}(p))\ M(p) \otimes g(\overline{w})\ \textbf{else}\ g(\overline{w}) \mid \overline{q_1}\,];\\ &\qquad [\, z \mid p \leftarrow M.\text{keys},\ z \leftarrow f(M(p))\,]\ \ \}\end{aligned} \tag{12}$$

Consider now an array comprehension of the form:

$$\text{matrix}(n,m)[\,((i,j),e) \mid \overline{q_1},\ \textbf{group by}\ (i,j),\ \overline{q_2}\,],$$

Notice that, here, the matrix index (i,j) in the comprehension head is the group-by key. In the implementation (12) of an array comprehension with group-by, we can now use arrays of size n*m, one array for each aggregation, instead of a Map:

$$\begin{aligned}&\text{matrix}(n,m)[\,((i,j),e) \mid \overline{q_1},\ \textbf{group by}\ (i,j),\ \overline{q_2}\,]\\ &= \text{matrix}(n,m)[\, z \mid \overline{q_1},\ \textbf{group by}\ (i,j),\\ &\qquad\qquad z \leftarrow f(\oplus_1/w_1.\text{map}(g_1),\ldots,\oplus_n/w_n.\text{map}(g_n))\,]\\ &= \{\ \textbf{val}\ V_1 = \text{Array.fill}(n*m)(\mathbf{1}_{\oplus_1});\\ &\qquad \ldots\\ &\qquad \textbf{val}\ V_n = \text{Array.fill}(n*m)(\mathbf{1}_{\oplus_n});\\ &\qquad [\ \{\ V_1(i*n+j) = V_1(i*n+j) \oplus_1 g_1(v_1);\\ &\qquad\qquad \ldots\\ &\qquad\qquad V_n(i*n+j) = V_n(i*n+j) \oplus_n g_n(v_n)\ \}\ \mid \overline{q_1}\,];\\ &\qquad \text{matrix}(n,m)[\, z \mid ((i,j),\_) \leftarrow V_1,\end{aligned}$$

[1] That is, the monoid $\otimes$ with identity $\mathbf{1}_\otimes = (\mathbf{1}_{\oplus_1},\ldots,\mathbf{1}_{\oplus_m})$ and $(x_1,\ldots,x_m) \otimes (y_1,\ldots,y_m) = (x_1 \oplus_1 y_1,\ldots,x_m \oplus_m y_m)$.

$$z \leftarrow f(V_1(i*n+j), \ldots, V_n(i*n+j))\ ]\ \}.$$

Notice that, the final result is a matrix constructed using the matrix(n, m) builder, and is made out of the arrays that hold the aggregation results. This matrix though does not have a group-by and can be translated using the methods given in Section 2.

For example, the matrix multiplication in Query (9) is translated as follows:

```
matrix(n,m)[ ((i,j),+/v)
               | ((i,k),a) ← sparsify(M), ((kk,j),a) ← sparsify(N),
                 kk == k, let v = a*b, group by (i,j) ],
      (after unfolding the sparsifiers for M and N)
= matrix(n,m)[ ((i,j),+/v)
               | let (n,l,A) = M, i ← 0 until n, k ← 0 until l,
                 let (ll,m,B) = N, kk ← 0 until ll, j ← 0 until m,
                 kk == k, let v = A(i*n+k)*B(kk*ll+j), group by (i,j) ]
      (after merging the array index kk with k)
= matrix(n,m)[ ((i,j),+/v)
               | let (n,l,A) = M, i ← 0 until n, k ← 0 until l,
                 let (ll,m,B) = N, j ← 0 until m,
                 let v = A(i*n+k)*B(k*ll+j), group by (i,j) ]
      (by translating the group-by qualifier)
= { val V = Array.fill(n,m)(0.0);
    [ V(i*n+j) = V(i*n+j) + A(i*n+k)*B(k*ll+j)
    | let (n,l,A) = M, i ← 0 until n, k ← 0 until l,
      let (ll,m,B) = N, j ← 0 until m ];
    matrix(n,m)[ v | ((i,j),_) ← V, v ← [ ((i,j),V(i*n+j)) |] ] }
= { val V = Array.fill(n,m)(0.0);
    [ V(i*n+j) = V(i*n+j) + A(i*n+k)*B(k*ll+j)
    | let (n,l,A) = M, i ← 0 until n, k ← 0 until l,
      let (ll,m,B) = N, j ← 0 until m ];
    matrix(n,m)[ ((i,j),V(i*n+j)) | ((i,j),_) ← V ] }
    (since the last term is equal to (n,m,V))
= { val V = Array.fill(n,m)(0.0);
    [ V(i*n+j) = V(i*n+j) + A(i*n+k)*B(k*ll+j)
    | let (n,l,A) = M, i ← 0 until n, k ← 0 until l,
      let (ll,m,B) = N, j ← 0 until m ];
    (n,m,V) }
```

which is equivalent to the desired efficient loop-based program.

4 TRANSLATING QUERIES ON SPARK

In this section, we translate array comprehensions to distributed programs that can run on Apache Spark [27]. Distributed datasets in Spark are represented as Resilient Distributed Datasets (RDDs), which support a functional API that is very similar to that for Scala collections. Most RDD operations are second-order, in which the functional argument is evaluated sequentially while the operation itself is evaluated in parallel, in a distributed mode. Unlike Scala collections, Spark does not allow nested RDDs and will raise a run-time error if the functional parameter of an RDD operation accesses another RDD. That is, Spark does not support nested parallelism because it is hard to implement efficiently in a distributed setting. However, instead of using nested RDD operations, one may use joins, cogroups, and cross products to correlate RDDs. Consequently, RDD comprehensions require special translation rules to derive joins, instead of nested flatMaps.

The RDD builder, rdd, that converts a List[T] to an RDD[T] can be implemented by applying the Spark method 'parallelize' on this list. However, RDD comprehensions must be translated to RDD operations in a special way to avoid generating nested operations. A group-by qualifier can be translated to the Spark groupByKey operation of type RDD[(K,V)]⇒RDD[(K,List[V])] using Equation (11). However, groupByKey is an expensive operation because it collects the grouped values into a list, shuffles these lists to the reducers, and finally reduces them by some aggregation. Instead, we want to generate calls to the more efficient reduceByKey(⊗), for some monoid ⊕, that reduces the values of type V using the monoid ⊗, instead of placing them into a list. That way, grouped values are partially reduced before they are shuffled. To generate these reduceByKey calls, we consider the group-by qualifier in combination with the aggregations in the comprehension. Recall that, based on the discussion in Section 3 and on Equation (12), any comprehension with a group-by can be put into the following form:

$$\begin{aligned}
&\text{rdd}[\, e \mid \overline{q_1},\ \textbf{group by}\ p,\ \overline{q_2}\,] \\
&\quad = \text{rdd}[\, z \mid \overline{q_1},\ \textbf{group by}\ p,\ z \leftarrow [\, e \mid \overline{q_2}\,]\,] \\
&\quad = \text{rdd}[\, z \mid \overline{q_1},\ \textbf{group by}\ p, \\
&\qquad\qquad z \leftarrow f(\oplus_1/w_1.\text{map}(g_1), \ldots, \oplus_m/w_m.\text{map}(g_m))\,],
\end{aligned}$$

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 can be translated to a reduceByKey operation:

$$\begin{aligned}
&\text{rdd}[\, e \mid \overline{q_1},\ \textbf{group by}\ p,\ \overline{q_2}\,] \\
&\quad = \text{rdd}[\, (p, (g_1(w_1), \ldots, g_m(w_m))) \mid \overline{q_1}\,] \\
&\qquad .\text{reduceByKey}(\otimes) \qquad\qquad\qquad\qquad (13) \\
&\qquad .\text{map}\{\ \textbf{case}\ (p, (a_1, \ldots, a_m)) \Rightarrow f(a_1, \ldots, a_m)\ \},
\end{aligned}$$

where $\otimes = \oplus_1 \times \cdots \times \oplus_m$.

The following rule identifies and generates joins between the RDDs X and Y, instead of nested flatMaps, 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:

$$\begin{aligned}
&\text{rdd}[\, e \mid \overline{q_1},\ p_1 \leftarrow X,\ \overline{q_2},\ p_2 \leftarrow Y,\ \overline{q_3},\ e_1 == e_2,\ \overline{q_4}\,] \\
&\quad = \text{rdd}[\, e \mid \overline{q_1},\ (\_, (p_1, p_2)) \leftarrow Z, \overline{q_2},\ \overline{q_3},\ \overline{q_4}\,], \qquad (14)
\end{aligned}$$

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

One way to represent arrays in a distributed setting is to store them as coordinate arrays, similar to the array representation used in Section 2. For instance, a matrix can be defined in Spark as an RDD of type RDD[((Long,Long),Double)], while matrix multiplication of two RDD matrices A and B, which was defined in (9), will be translated to the following program using Rules (14) and (13):

```
A.map{ case ((i,k),a) ⇒ (k,((i,k),a)) }
  .join( B.map{ case ((kk,j),b) ⇒ (kk,((kk,j),b)) } )
  .map{ case (_,(((i,k),a),((kk,j),b))) ⇒ ((i,j),a*b) }
  .reduceByKey(_+_).
```

Although correct, this Spark program has a high cost: it shuffles the matrices A and B across the compute nodes to perform the join, and then it shuffles all the products $A_{ik} * B_{kj}$ to perform the reduceByKey. Since data shuffling is the main cost factor for a distributed program, instead of fully sparse matrices in the coordinate format, we want to use a more compact representation for matrices by partitioning a matrix into tiles, which are unboxed arrays of type Array[Double] in which indices are calculated, not stored. The sparse matrix representation, on the other hand, is preferable when both dimensions of the matrix are large and the matrix is very sparse.

5 TRANSLATING BLOCK ARRAY QUERIES

A more effective way of representing an array in a distributed setting is to encode it as a distributed bag of non-overlapping blocks, where each block is a fix-sized chunk of the distributed array. A block is the unit of data distribution. Our goal in this section is to translate RDD comprehensions over block arrays to Spark's distributed data-parallel programs whose functional parameters will process the blocks very efficiently using multi-core parallelism and array indexing. Given that data partitioning is necessary for data-parallel distributed processing, blocks are a natural way to partition large arrays and at the same time to minimize space overhead, compared to fully sparse arrays in coordinate format, in which the indices are stored along with a matrix element. This small space footprint translates to less data to shuffle across nodes and faster time to process each partition. Spark actually uses thread-level parallelism at each compute node to process the elements of each RDD partition in parallel using multicore parallelism, but the unit of parallelism for a block array is the entire block, which is likely to be one block for each compute node. Consequently, in addition to generating distributed operations from array comprehensions on block arrays, our goal is to process the data inside blocks using multicore parallelism.

In this paper, we focus on tiled matrices but our work can be easily extended to handle other block arrays too. We represent a tiled matrix using the following Scala class:

```
case class Tiled[T] ( rows: Long, cols: Long,
                      tiles: RDD[((Long,Long),Array[T])] ),
```

where rows is the number of rows, cols is the number of columns, and tiles is an RDD of fix-sized square tiles, where each tile ((i,j),A) has coordinates i and j and values stored in the array A. The array A has a fixed size N*N, for some constant N which is the same for all tiles. The coordinates i and j of a tile are unique, that is, tiles is an association list. A matrix element with indices ij is stored in the tile that has coordinates $(i/N, j/N)$ at the location $(i\%N) * N + (j\%N)$ inside the tile. The tile sparsifier is as follows:

```
def sparsify[T] ( S: Tiled[T] ): List[((Long,Long),T)]
  = [ ( ( ii*N+i, jj*N+j ), a(i*N+j) )
    | ((ii,jj),a) ← S.tiles,
      i ← 0 until N, j ← 0 until N ],
```

where ii and jj are the tile coordinates, and i and j are indices within a tile. The tiled builder uses the rdd and the array builders:

```
def tiled[T] ( n: Long, m: Long )
             ( L: List[((Long,Long),T)] ): Tiled[T]
  = Tiled( n, m, rdd[ ( (ii, jj), array(N*N)(w) )
                    | ((i,j),v) ← L, let ii = i/N, let jj = j/N,
                      let w = ( (i%N)*N+(j%N), v ),
                      group by (ii,jj) ] ),
```

where the group-by collects all tile elements into an array. The group-by comprehension is in an RDD comprehension, which means that it will be translated to a groupByKey operation in Spark, which requires data shuffling across the compute nodes. However, in some cases, this group-by qualifier can be eliminated, as in the case of a map over a matrix. Such an optimization is actually a general optimization over comprehensions with a group-by. A group-by qualifier in a comprehension can be eliminated if the group-by key is unique, that is, when the group-by function is injective. Although it is in general undecidable to prove whether a group-by key is unique, it is easy to do so for special cases, such as when the group-by key consists of array indices that are bound through an array traversal. For an array or map A, and the pattern variables v_i in $\overline{q_1}$ or $\overline{q_2}$, we have:

$$\begin{aligned}&[\, e \mid \overline{q_1},\ (k,v) \leftarrow A,\ \overline{q_2},\ \textbf{group by}\ k\,] \qquad (15)\\ &\quad = [\, e \mid \overline{q_1},\ (k,v) \leftarrow A,\ \textbf{let}\ \overline{v} = \text{unzip}([\, \overline{v} \mid \overline{q_2}\,])\,],\end{aligned}$$

since the generator $(k,v) \leftarrow A$ for an array or map A indicates that k is unique. That is, this group-by is removed and every pattern variable v_i in $\overline{q_1}$ or $\overline{q_2}$ is lifted to a bag that contains all its values in the group. A similar rule exists for a generator $k \leftarrow e_1$ **until** e_2, since every value of k is unique.

Although correct, unfolding and normalizing tiled array comprehensions based on the tiled sparsifier and builder do not always result to optimal translations. In the rest of this section, we present special rules to translate tiled array comprehensions to efficient Spark programs.

5.1 Queries that Preserve Tiling

Consider the block comprehension without a group-by:

$$\text{tiled}(\overline{d})[\, (key, e) \mid \overline{q}\,] \qquad (16)$$

where $\overline{d}$ is the tile dimensions (e.g., (m,n) for a matrix), key is the tile indices (e.g., (i,j) for a matrix) and e is the associated value. Let $(\overline{k_i}, v_i) \leftarrow X_i$ be a generator over a tiled array X_i in $\overline{q}$, where $\overline{k_i}$ is a tuple of index variables, such as $((i,j),v) \leftarrow X$ for a tiled matrix X. We say that this tiled comprehension *preserves tiling* if key is a tuple $\overline{w}$ that consists of variables that are defined in the tuples $\overline{k_i}$. The rest of the variables in the tuples $\overline{k_i}$ (not in $\overline{w}$) must be related to the variables in $\overline{w}$ with equality predicates in $\overline{q}$, so that the index of the constructed array is unique. For example, matrix addition and matrix diagonal preserve tiling:

```
tiled(n,m)[ ((i,j),a+b) | ((i,j),a) ← A, ((ii,jj),b) ← B,
                          ii == i, jj == j ],
tiled(n)[ (i,a) | ((i,j),a) ← A, i == j ].
```

A comprehension that preserves tiling is translated to an RDD comprehension that does not need a group-by to shuffle tiles. More specifically, the comprehension (16) is translated to:

$$\text{Tiled}(\overline{d}, \text{rdd}[\, (\overline{w}, \text{array}(N*N)[\, (\overline{w}, e) \mid \overline{q_2}\,]) \mid \overline{q_1}\,]), \qquad (17)$$

where the qualifiers in $\overline{q_1}$ are those from $\overline{q}$ that do not refer to the tile values and each tiled generator $(\overline{k_i}, v_i) \leftarrow X_i$ has been modified to be $(\overline{k_i}, \_v_i) \leftarrow X_i$.tiles (given the pattern variable v_i bound to an array value, $\_v_i$ is bound to the entire tile in $\overline{q_1}$). The qualifier list $\overline{q_2}$ is equal to $\overline{q}$ but each tiled generator $(\overline{k_i}, v_i) \leftarrow X_i$ in $\overline{q}$ has been modified to be $(\overline{k_i}, v_i) \leftarrow \_v_i$. For example, matrix addition:

```
tiled(n,m)[ ((i,j),a+b) | ((i,j),a) ← A, ((ii,jj),b) ← B,
                          ii == i, jj == j ]
```

is translated to:

```
Tiled( n, m, rdd[ ( (i,j), V(_a,_b) )
                | ((i,j),_a) ← A.tiles, ((ii,jj),_b) ← B.tiles,
                  ii == i, jj == j ] ),
```

where V(_a,_b) is:

```
array(N*N)[ ((i,j),a+b) | ((i,j),a) ← _a, ((ii,jj),b) ← _b,
                          ii == i, jj == j ],
```

which is a regular array comprehension in which the tiles _a and _b will be lifted using the following array sparsifier for a tile A:

```
[ ((i,j),A(i*N+j)) | i ← 0 until N, j ← 0 until N ].
```

Based on the translation of RDD and array comprehensions, matrix addition is translated to the following Spark code:

```
Tiled( n, m, A.tiles.join(B.tiles)
          .map{ case ((ii,jj),(_a,_b)) ⇒ ((ii,jj),V(_a,_b)) } ).
```

After optimizations similar to those for matrix addition for regular matrices, we get the following code for V(_a,_b):

```
{ val V = Array.ofDim[Double](N*N);
  [ V((i%N)*N+(j%N)) = _a( (i%N)*N+(j%N) )
                        + _b( (i%N)*N+(j%N) )
  | i ← (0 until N).par,  /* multicore parallelism */
    j ← 0 until N ];
  V }.
```

5.2 Queries that do not Preserve Tiling

For those array comprehensions that do not have a group-by and do not preserve tiling, we need to shuffle only the relevant tiles to the appropriate reducers. The array indices $\overline{w}$ in the result of the tiled comprehension

$$\text{tiled}(\overline{d})[\,(\overline{w}, e) \mid \overline{q}\,] \qquad (18)$$

may now be arbitrary expressions that depend on the indices of the tiled generators in $\overline{q}$.

Let $f(\overline{k})$ be a term that depends on the array indices $\overline{k} = k_1, \ldots, k_m$ from the tiled generators in $\overline{q}$. The tiles accessed from the tiled generators $\overline{q}$ will have tile coordinates $\overline{K}$, where $K_i = k_i/N$. Given the tile coordinates $\overline{K}$ of the input tiles, the tile coordinate returned by $f(\overline{k})$ would be equal to $f(K_1 * N + j_1, \ldots, K_m * N + j_m)/N$, for $j_i \in [0, N)$ (the tile dimensions). We define, $\mathcal{I}_f(\overline{K})$ to be the set of all such tile coordinates:

$$\text{set}[\,f(K_1 * N + j_1, \ldots, K_m * N + j_m)/N \mid j_1 \leftarrow 0\,\textbf{until}\,N,\ \ldots,\ j_m \leftarrow 0\,\textbf{until}\,N\,],$$

where set(s) returns the distinct values of s. For example, if $f(k) = k+1$, then $\mathcal{I}_f(K) = \text{set}[\,f(K*N+j)/N \mid j \leftarrow 0\,\textbf{until}\,N\,]$, which is equal to the set $\{K, K+1\}$ since $f(K*N+j)/N = (K*N+j+1)/N = K+(j+1)/N$. On the other hand, if $f(k) = k$, then $\mathcal{I}_f(K) = \{K\}$.

Based on this definition, comprehension (18) can be transformed to:

$$\text{Tiled}(\overline{d}, \text{rdd}[\,(\overline{K}, V) \mid \overline{q_1},\ K_1 \leftarrow \mathcal{I}_{w_1}(\overline{k}),\ \ldots,\ K_m \leftarrow \mathcal{I}_{w_m}(\overline{k}),\ \textbf{group by}\,\overline{K}\,]), \qquad (19)$$

where V is:

$$\text{array}[\,(\overline{w}, e) \mid \overline{q_2},\ \textstyle\bigwedge_i \overline{K_i} == w_i(\overline{\_k} * N + \overline{k})/N\,].$$

The qualifiers in $\overline{q_1}$ are those from $\overline{q}$ that do not refer to the tile values and each tiled generator $(\overline{k_i}, v_i) \leftarrow X_i$ has been transformed to the two qualifiers $(\overline{k_i}, \_v_i) \leftarrow X_i$.tiles and **let** $\_\_v_i = (\overline{k_i}, \_v_i)$ (given the pattern variable v_i bound to an array value, $\_v_i$ is bound to the entire tile in $\overline{q_1}$, while $\_\_v_i$ is bound to the index-tile pair). The group-by operation shuffles all the required tiles to the reducers to compute the resulting tiles. The qualifier list $\overline{q_2}$ in V is equal to $\overline{q}$ but each tiled generator $(\overline{k_i}, v_i) \leftarrow X_i$ in $\overline{q}$ has been transformed to the two qualifiers $(\overline{\_k_i}, \_v_i) \leftarrow \_\_v_i$ and $(\overline{k_i}, v_i) \leftarrow \_v_i$. The guards $\overline{K_i} == w_i(\overline{\_k} * N + \overline{k})/N$ selects only the proper tiles from all those shuffled by group-by.

For example, the following comprehension rotates the rows so that the first row is moved to the second, the second to third, etc, and the last to the first:

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

The shuffled tiles for a tile in X with coordinates (i,j) have row coordinates from set[(i*N+_i+1)%m/N | _i ← 0 **until** N], which will be evaluated to List(i,i+1) for i < n/N and to List(i,0) for i = n/N, and column coordinates from List(j). That is, for each tile with coordinates (i,j) such that i < n/N, we require two tiles: the tile itself and its row successor with coordinates (i+1,j). Hence, the row rotation is translated to:

```
Tiled(n,m,rdd[ ( (K1,K2), V )
              | ((i,j),_a) ← X.tiles, let __a = ((i,j),_a),
                K1 ← set[ (i*N+_i+1)%m/N | _i ← 0 until N ],
                K2 ← List(j),
                group by (K1,K2)  ] ),
```

which is translated to a Spark groupByKey operation. V is:

```
array(N*N)[ (((i+1)%m,j),a) | ((_i,_j),_a) ← __a, ((i,j),a) ← _a,
              K1 == (_i*N+i+1)%m/N, K2 == (_j*N+j)/N ] ),
```

which is translated to an efficient array-based program over the list of tiles __a.

5.3 Queries with a Group-By

Tile comprehensions with a group-by do not preserve tiling. They can be translated in a way similar to that for comprehension (18), but we can do better by using the RDD operation reduceByKey instead of the RDD operation groupByKey, because a group-by in a group-by comprehension is often followed by aggregation. A reduceByKey(⊕) operation, for some monoid ⊕, is equivalent to a groupByKey followed by reduction of each group using ⊕. Although functionally equivalent, reduceByKey is far more efficient than groupByKey in a distributed setting because it partially reduces the groups locally before they are shuffled to the reducers for the final reduction, resulting to less data shuffling.

Before we describe the general translation scheme, we explain the key idea using the example of matrix multiplication of the tiled matrices A and B:

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

We want to translate it to the reduceByKey operation:

```
Tiled( n, m, rdd[ ((i,j),V(_a,_b))
               | ((i,k),_a) ← A.tiles, ((kk,j),_a) ← B.tiles,
                 kk == k ].reduceByKey(⊕) ),
```

where the tile V(_a,_b) is the tile _a multiplied by _b:

```
array(N*N)[ ((i,j),+/v)) | ((i,k),a) ← _a, ((kk,j),a) ← _b,
                           kk == k, let v = a*b,
                           group by (i,j) ]
```

and the monoid ⊕ over the tiles _x and _y adds the tiles pairwise:

```
_x⊕_y = array(N*N)[ ((i,j),x+y) | ((i,j),x) ← _x, ((ii,jj),y) ← _y,
                                  ii == i, jj == j ].
```

That is, the rdd comprehension calculates the partial sum of products of all matching tiles and the reduceByKey calculates the final sums by adding the tiles pairwise. Then, after translating the rdd and array comprehensions, we will get:

```
Tiled(n,m,A.tiles.map{ case ((i,k),_a) ⇒ (k,((i,k),_a)) }
    .join( B.tiles.map{ case ((kk,j),_a) ⇒ (kk,((kk,j),_a)) } )
    .map{ case (_,(((i,k),_a),((kk,j),_a))) ⇒ ((i,j),V(_a,_b)) }
    .reduceByKey(⊕))
```

where V(_a,_b) is translated to efficient code that multiplies the tiles _a and _b:

```
{ val V = Array.ofDim[Double](N*N);
  for { i ← 0 until N; j ← 0 until N; k ← 0 until N }
     V(i*N+j) += _a(i*N+k)*_b(k*N+j);
  V },
```

and _x⊕_y is pairwise addition:

```
{ val V = Array.ofDim[Double](N*N);
  for { i ← 0 until N; j ← 0 until N }
     V(i*N+j) = _x(i*N+j)+_y(i*N+j);
  V }.
```

To generate these reduceByKey calls, we consider the group-by qualifier in combination with the aggregations in the comprehension. Recall that, based on the discussion in Section 3 and on Equation (12), any tiled comprehension with a group-by can be put into the following form:

$$\begin{aligned}
&\text{tiled}(n,m)[\, (p,e) \mid \overline{q},\ \textbf{group by}\ p,\ \overline{q'}\,] \\
&\quad = \text{tiled}(n,m)[\, (p,z) \mid \overline{q},\ \textbf{group by}\ p,\ z \leftarrow [\, e \mid \overline{q'}\,]\,] \\
&\quad = \text{tiled}(n,m)[\, (p,z) \mid \overline{q},\ \textbf{group by}\ p, \\
&\qquad\qquad z \leftarrow f(\oplus_1/w_1.\text{map}(g_1), \ldots, \oplus_m/w_m.\text{map}(g_m))\,],
\end{aligned}$$

for some variables w_i lifted by group-by, some monoids $\oplus_i$, and some functions g_i and f. That is, we abstract all reductions $\oplus_i/w_i.\text{map}(g_i)$ from the term $[\, e \mid \overline{q'}\,]$. Notice that, here, the key of the tiled comprehension is equal to the group-by key, p. Then, the group-by comprehension can be translated to the following reduceByKey operation:

$$\begin{aligned}
\text{Tiled}(n,m,\text{rdd}[\, (p, (&\text{array}(N*N)[\, g_1(w_1) \mid \overline{q_2}\,], \ldots, \\
&\text{array}(N*N)[\, g_m(w_m) \mid \overline{q_2}\,])) \\
&\mid \overline{q_1}\,] \\
&.\text{reduceByKey}(\otimes').\text{mapValues}(f'))
\end{aligned}$$

Like in (17), the qualifiers in $\overline{q_1}$ are those from $\overline{q}$ that do not refer to the tile values and each tiled generator $(\overline{k_i}, v_i) \leftarrow X_i$ has been modified to be $(\overline{k_i}, \_v_i) \leftarrow X_i$.tiles. The qualifier list $\overline{q_2}$ is equal to $\overline{q}$ but each tiled generator $(\overline{k_i}, v_i) \leftarrow X_i$ in $\overline{q}$ has been modified to be $(\overline{k_i}, v_i) \leftarrow \_v_i$. The monoid $\otimes'$ is:

$$(x_1, \ldots, x_m) \otimes' (y_1, \ldots, y_m) \;=\; (x_1 \oplus_1' y_1, \ldots, x_m \oplus_m' y_m)$$

where $x_k \oplus_k' y_k$ applies $\oplus_k$ to the tile elements pairwise:

$$\begin{aligned}
\text{array}(N*N)[\, a \oplus_k b \mid ((i,j),a) \leftarrow x_k,\ ((ii,jj),b) \leftarrow y_k, \\
ii == i,\ jj == j\,]
\end{aligned}$$

Finally, $f'(x_1, \ldots, x_m)$ is:

$$\begin{aligned}
\text{array}(N*N)[\, &f(a_1, \ldots, a_m) \\
&\mid ((i_1,j_1),a_1) \leftarrow x_1, \ldots,\ ((i_m,j_m),a_m) \leftarrow x_m, \\
&\quad i_1 == \cdots == i_m,\ j_1 == \cdots == j_m\,]
\end{aligned}$$

For the matrix multiplication example, there is only one reduction with $\oplus_1 = +$ and f is identity, which means that mapValues(f') is identity too.

5.4 Using a Group-By-Join

There is a special class of tiled comprehensions with a group-by that can be translated to more efficient code. Consider the following comprehension:

```
tiled(n,m)[ (k,⊕/c) | ((i,j),a) ← A, ((ii,jj),b) ← B,
                      kx(i,j) == ky(ii,jj), let c = h(a,b),
                      group by k: ( gx(i,j), gy(ii,jj) ) ],
```

for some arbitrary term functions kx, ky, gx, gy, and h, and some aggregation ⊕. Notice that the key k of the output matrix is the group-by key, which must be a pair where one component depends on i,j and the other on ii,jj only. This is called a group-by-join because it is a join between A and B, followed by a group-by with aggregation. Matrix multiplication, defined in (9), is an example of a group-by-join. Many group-by comprehensions can be put in the group-by-join form by combining generators in pairs forming joins until we are left with two generators and a group-by. A group-by-join can be evaluated very efficiently by joining each row of tiles from A with each column of tiles from B, which requires that we replicate every tile from A and B. When applied to the matrix multiplication, this algorithm is equivalent to the block matrix multiplication implemented using the SUMMA algorithm [14]. The group-by-join is translated to the following Spark RDD code:

```
Tiled( n, m, rdd[ (k,V) | (k,(__a,__b)) ← As.cogroup(Bs) ] ),
```

where As, Bs, and V are

```
As = A.tiles.flatMap{ case ((i,j),a)
     ⇒ (0 until B.cols/N).map(k ⇒ ((gx(i,j),k),(kx(i,j),a))) }
Bs = B.tiles.flatMap{ case ((ii,jj),b)
     ⇒ (0 until A.rows/N).map(k ⇒ ((k,gy(ii,jj)),(ky(ii,jj),b))) }
V = array(N*N)[ (k,⊕/c)
         | (k1,_a) ← __a, (k2,_b) ← __b, k2 == k1,
           ((i,j),a) ←_ a, ((ii,jj),b) ← _b, kx(i,j) == ky(ii,jj),
           let c = h(a,b), group by k: (gx(i,j),gy(ii,jj)) ].
```

That is, the tiles in A are replicated B.cols/N times and the tiles in B are replicated A.rows/N times.

6 PERFORMANCE EVALUATION

Our system, SAC, has been implemented using Scala's compile-time reflection and macros. Our code generator uses the Scala typechecker to infer the types of the generator domains to select the appropriate sparsifiers based on these types. It translates array comprehensions to Scala code that calls Spark RDD operations whose functional arguments use the Scala's Parallel Collections library [20] for multicore parallelism. The produced Scala code is embedded in the rest of the Scala code generated at compile-time. The source code of our system is available on GitHub at https://github.com/fegaras/array.

We have evaluated the performance of our system relative to the Spark MLlib.linalg library [5]. MLlib uses the linear algebra package Breeze, but in our experiments, instead of using a native Breeze library implementation, such as OpenBLAS, we used the pure JVM implementation of this library. Although there are many linear algebra libraries that support distributed tiled arrays, MLlib is the closest to our work since it is built on top of Spark and has a Scala API.

The platform used for these evaluations was a small cluster of 4 nodes, where each node has one Xeon E5-2680v3 at 2.5GHz, with 24 cores, 128GB RAM, and 320GB SSD. For our experiments,

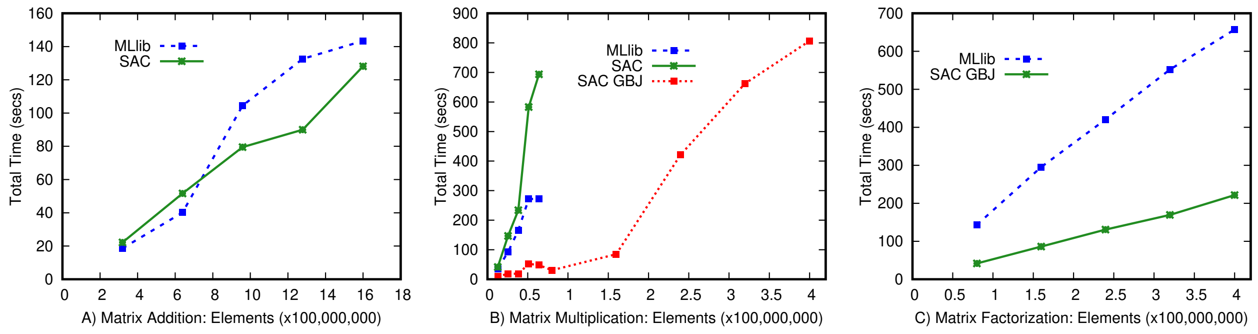


Figure 4: Performance evaluation of matrix addition, multiplication, and factorization of tiled matrices on Apache Spark

we used Apache Spark 3.0.0 running on Apache Hadoop 2.7.0. Each Spark executor was configured to have 11 cores and 60GB RAM. Consequently, there were 2 executors per node, giving a total of 8 executors. Each program was evaluated over 5 datasets and each evaluation was repeated 4 times, so each data point in the plots in Figure 4 represents the mean value of these 4 evaluations.

The matrices used in our experiments were tiled matrices where each tile had size 1000*1000. We implemented these distributed tiled matrices in MLlib.linalg as instances of BlockMatrix, where each tile was an instance of DenseMatrix. The matrices used for addition and multiplication were pairs of square matrices of the same size filled with random values between 0.0 and 10.0. The largest matrices used in matrix addition had 40000×40000 elements and size 12GB each, while those used in matrix multiplication had 20000×20000 elements and size 3GB each. Matrix multiplication was translated in two different ways in SAC: as a join followed by a group-by, and using a special group-by join (see Section 5.4 for a discussion). The results are shown in Figure 4.A and B. We can see that, for matrix addition, SAC performs a bit faster than MLlib. For matrix multiplication though, SAC (that uses a join followed by a group-by) is up to 3 times slower than MLlib, while MLib is up to 6 times slower than SAC GBJ (that uses the special group-by join).

The third program to evaluate was one iteration of matrix factorization using gradient descent [16]. The goal of this computation is to split a matrix R of dimension $n \times m$ into two low-rank matrices P and Q of dimensions $n \times k$ and $m \times k$, for small k, such that the error between the predicted and the original rating matrix $R - P \times Q^T$ is below some threshold, where $P \times Q^T$ is the matrix multiplication of P with the transpose of Q and '$-$' is cell-wise matrix subtraction. Matrix factorization can be implemented by repeatedly applying the following operations:

$$\begin{array}{lcl} E & \leftarrow & R - P \times Q^T \\ P & \leftarrow & P + \gamma(2E \times Q - \lambda P) \\ Q & \leftarrow & Q + \gamma(2E^T \times P - \lambda Q) \end{array}$$

where γ is the learning rate and λ is the normalization factor used in avoiding overfitting. For our experiments, we used $\gamma = 0.002$ and $\lambda = 0.02$. The matrix to be factorized, R, was a square sparse matrix $n*n$ with random integer values between 0 and 5, in which only the 10% of the elements were non-zero. The dimension k was set to 1000. The derived matrices P and Q had dimension $n * 1000$ and were initialized with random values between 0.0 and 1.0. The largest matrix R used had 20000×20000 elements and size 3GB. The results are shown in Figure 4.C. We can see that SAC (using GBJ) is up to three times faster than MLlib.

7 RELATED WORK

Many array-processing systems use special storage techniques, such as regular tiling, to achieve better performance on certain array computations. TileDB [19] 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 [22] 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 [21] stores arrays into chunks, which are typically the size of a storage block. One of their most effective storage method is a two-level chunking strategy with regular chunks and regular tiles. SciHadoop [7] 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 [15] 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 [25] 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. Linalg [26] (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 [6] 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. 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 [2, 17, 18]. The work by Luo *et al.* [18] 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 would differ from ours since it does not separate specification from implementation, thus making hard to change the array storage, and requires programmers to write explicit code to correlate tiles.

8 CONCLUSION AND FUTURE WORK

Our performance results show that SAC can be as efficient as a highly optimized array library when applied to certain distributed operations on tiled arrays. The same layered approach can also be used for translating comprehensions on other types of array storage, such as on tiled arrays where each tile is stored in the compressed sparse column format. As future work, we plan to look at operations that are hard to express using comprehensions, such as inverting a matrix, which requires a special LU decomposition algorithm. We believe that such operations should be coded as black-box library functions in a high-performance array library, such as BLAS or LAPACK. Such operations would require special optimizations to fuse them with general array comprehensions and with each other. Furthermore, our framework cannot directly generate calls to a high-performance array library, such as BLAS or LAPACK, or to GPU libraries, such as CUDA or OpenGL, but it can be improved to recognize certain patterns in a comprehension that are translatable to such calls. Finally, we would like to investigate general methods to optimize storage, such as unboxing arrays where vectors of tuples are mapped to tuples of vectors, and to parallelize irregular structures using nested parallelism.

Acknowledgments: Our evaluations were performed at the XSEDE Comet 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] 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.

[3] Apache Flink. http://flink.apache.org/, 2020.

[4] Apache Spark. http://spark.apache.org/, 2020.

[5] Apache Spark MLlib. https://spark.apache.org/mllib/, 2020.

[6] 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.

[7] 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.

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

[9] 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.

[10] L. Fegaras. An Algebra for Distributed Big Data Analytics. *JFP*, special issue on Programming Languages for Big Data, volume 27, 2017.

[11] 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.

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

[13] 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.

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

[15] 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.

[16] Y. Koren, R. Bell, and C. Volinsky. Matrix Factorization Techniques for Recommender Systems *IEEE Computer*, 42(8):30–37, August 2009.

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] P. Wadler. List Comprehensions. Chapter 7 in *The Implementation of Functional Programming Languages*, by S. Peyton Jones, Prentice Hall, 1987.

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

[25] 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.

[26] 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.

[27] 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.

CONTENTS

Scalable Linear Algebra Programming for Big Data Analysis

Leonidas Fegaras
University of Texas at Arlington
Arlington, Texas
fegaras@cse.uta.edu

ABSTRACT

Arrays are very important data structures for many data-centric and scientific applications. One of the most effective representations of large dense arrays in a distributed setting is a block array, such as a tiled matrix, which is a distributed collection of non-overlapping dense array blocks. Although there are many linear algebra libraries for machine learning that support distributed block arrays and provide an optimal implementation for many array operations, these libraries do not support ad-hoc array programming and customized storage structures. Imperative programs with loops and array indexing, on the other hand, are more powerful as they allow arbitrary array computations but are hard to parallelize and convert to distributed programs.

Our goal is to provide an SQL-like abstraction for data-parallel distributed array computations that is expressive enough to capture a large class of array computations and can be compiled to efficient data-parallel distributed code. Our abstraction is a monolithic array construction in the form of an array comprehension that is as expressive as SQL by supporting a group-by syntax that allows us to capture many array computations in declarative form. We present rules for translating array comprehensions on block arrays to data-parallel distributed code that can run on Apache Spark. We describe a comprehensive set of effective optimizations that can produce very efficient translations, such as the optimal block matrix multiplication algorithm, even though they are oblivious to linear algebra operations. Finally, we justify our claims by evaluating the performance of our generated code on Apache Spark relative to Spark MLlib.

1 INTRODUCTION

Much of the data used in data-centric applications come in the form of arrays, such as vectors, matrices, and tensors. In the early days of numerical computing, most of the array programming was done in an imperative loop-based language, such as Fortran or C, using array indexing to access and update array elements incrementally, one at a time. Although loop-based programs are efficient when they run on a single processor, they are hard to parallelize and reason about. Currently, most array programming is done using vectorization languages, such as MATLAB, R, and NumPy, that allow programmers to write high-level array code that closely resembles mathematical formulas. These languages provide highly tuned array operations that are applied to whole arrays instead of individual elements, thus making loops inessential. Moreover, they hide the implementation details and optimize performance by choosing an implementation (a kernel) for an array operation from a variety of build-in array storages and algorithms. Internally, these languages rely on numerical libraries, such as BLAS [9], for efficient linear algebra computations. These libraries, which are also an integral part of many machine learning (ML) sytems, such as TensorFlow [1], PyTorch, and MLlib [5], implement basic array operations efficiently using multicore parallelism and GPU acceleration. Many array operations provided by the vectorization languages are overloaded to work on a variety of array storage structures, thus offering an implementation-independent view to the programmer. Given that there are numerous storage structures for arrays, such as dense, tiled, and compressed sparse matrices, each library operation must have numerous implementations, especially those operations that operate on multiple arrays, such as matrix multiplication. As a result, this code specialization based on array implementation is hard to extend with user-defined storage structures and algorithms. This is particularly true for distributed arrays, which have to be partitioned into blocks and distributed across compute nodes. In that case, not only there are numerous ways to implement these blocks as arrays, but there are also numerous ways to partition the arrays into blocks. A better solution would have been to express array computations in a high-level declarative language for array computations that is expressive enough to capture most array programs and is supported by a translation scheme that separates the specification from the implementation and generates high-quality code.

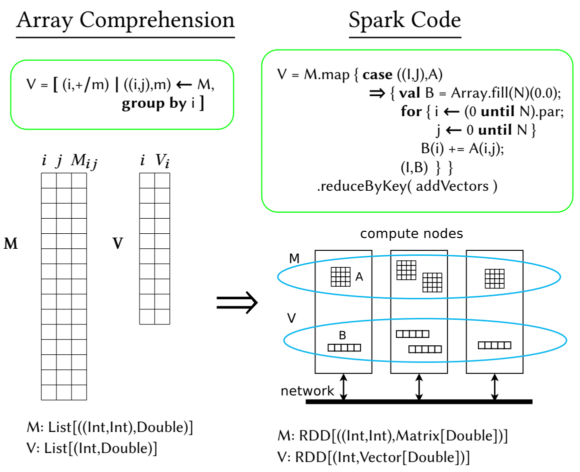
This problem of sacrificing expressiveness and extensibility for efficiency incurred from the library approach is exacerbated by the need to process large arrays that do not fit in memory. Given that the accuracy of the data analysis and ML models depends on the data size, current data-centric applications must analyze enormous amounts of array data using complex mathematical data processing methods. In recent years, new frameworks in distributed Big Data analytics have become essential tools for large-scale machine learning and scientific discoveries. These systems, which are also known as Data-Intensive Scalable Computing (DISC) systems, have revolutionized our ability to analyze Big Data. Unlike High-Performance Computing (HPC) systems, which are mainly designed for shared-memory architectures, DISC systems are distributed data-parallel systems on clusters of shared-nothing computers connected through a high-speed network. Compared to low-level distributed-memory communication paradigms, such as MPI, DISC systems automate many aspects of distributed computing, such as fault tolerance, which is important for long-running Big Data analysis on thousands of computers, scalability, data partitioning and distribution, and task scheduling and management. One of the earliest DISC systems is Map-Reduce [8], which was introduced by Google and later became popular as an open-source software with Apache Hadoop. Recent systems, such as Apache Spark [4] and Apache Flink [3], go beyond Map-Reduce by maintaining dataset partitions in the memory of the compute nodes. All these systems use data shuffling to exchange data among compute nodes, which takes place implicitly between the map and reduce stages in Map-Reduce and during group-bys and joins in Spark and Flink. Essentially, all data exchanges across compute nodes are done in a controlled way using special operations, which implement data shuffling by distributing data based on some key, so that data associated with the same key are processed together by the same compute node.

The goal of this paper is to provide a well-formed abstraction for data-parallel distributed array computations "without regret", that is, an abstraction that is declarative so that we can reason about it, is expressive enough to capture a large class of array computations, and can be compiled to efficient data-parallel distributed code. Our main construct is the *array comprehension*, which is a monolithic array construction in the form of a list comprehension. List comprehensions are found in many modern programming languages, such as Python, Scala, and Haskell. Unlike regular list comprehensions though, our array comprehensions are as expressive as SQL queries by supporting a group-by syntax that allows us to capture many array computations in declarative form without using array indexing which is hard to reason about. An array comprehension can access and correlate multiple arrays by traversing their elements one-by-one and can construct a new array in one shot by mapping array indices to values, which are derived from the elements of the input arrays. 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 and concatenation. More complex array operations, such as matrix inverse and LU decomposition, can be coded using array comprehensions inside loops.

1.1 Highlights of our Approach

This paper presents a generic and customizable system that translates abstract array programs to high-performance distributed code that can run on current DISC platforms. 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. Our framework uses a two-layer approach where array programs are expressed in a powerful high-level syntax on abstract arrays, while these abstract arrays are mapped to customized storage structures based on user-defined type mappings, thus separating specification from implementation.

An abstract array with dimensionality i in our framework 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 represented as an association list of key-value pairs in which the key contains 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 (i.e., some elements may be missing) if the array is sparse. A concrete implementation of an array (i.e., its storage structure) is specified by two customized functions: the *sparsifier*, which converts the storage structure to an association list, and the *builder*, which constructs the storage structure from the association list. These two functions, which are inverse of each other, are used by our translator to transform any operation $f(x_1,\ldots,x_n)$ on abstract arrays x_i to an operation on their concrete storage structures c_i by up-coercing the storages c_i using the sparsifiers s_i and down-coercing the abstract result using the builder b, that is,



```
V = [ (i,+/m) | ((i,j),m) ← M,
                  group by i ]
```

```
V = M.map { case ((I,J),A)
              ⇒ { val B = Array.fill(N)(0.0);
                  for { i ← (0 until N).par;
                        j ← 0 until N }
                     B(i) += A(i,j);
                  (I,B) } }
      .reduceByKey( addVectors )
```

M: List[((Int,Int),Double)]
V: List[(Int,Double)]

M: RDD[((Int,Int),Matrix[Double])]
V: RDD[(Int,Vector[Double])]

Figure 1: Code generation for $V_i = \sum_j M_{ij}$

$b(f(s_1(c_1),\ldots,s_n(c_n)))$. This layered approach introduces levels of indirection and generates superfluous intermediate structures (the abstract arrays x_i) that need to be removed. In our framework, this is accomplished by fusing these functions into one function that represents the concrete code so that the resulting program builds the output structures directly and works on the storage structures c_i without creating the association lists x_i.

Our language for expressing abstract array programs uses a monolithic array construction in the form of an array comprehension that is as expressive as SQL by supporting a group-by syntax that allows us to capture many array computations in declarative form. Comprehensions with a group-by syntax were first introduced by Wadler and Peyton Jones [24] and have been used as the formal calculus for the DISC query languages MRQL [10] and DIQL [12]. For example, the following comprehension returns the number of employees in each department:

[(d.name,count(e)) | e ← Employees, d ← Departments,
e.dno == d.dnumber, **group by** d.name].

The formal semantics of comprehensions, the translation of comprehensions to an algebra, and a query optimization framework are described in our earlier work [10, 12]. Array comprehensions are actually list comprehensions in which the array storages used in a comprehension are implicitly converted to association lists by array sparsifiers and the list returned by the comprehension is converted to an array storage by an array builder.

Consider for example the following array comprehension that constructs a vector V of size n from a matrix M of size $n \times m$ such that $V_i = \sum_j M_{ij}$, where both the matrix M and the resulting vector V are stored in memory:

V = vector(n)[(i, +/m) | ((i,j),m) ← M, **group by** i], (1)

which constructs the entire array V in one shot. The matrix M of type matrix[Double] is implicitly converted to an association list of type List[((Int,Int),Double)]. The generator ((i,j),m) ← M 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 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 bag of t, indicating that each such variable must now contain all the values associated with the same group-by key value. Consequently, after we group by i, the pattern variables that are not used in the group-by key, j and m, are now 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 calculates $\sum_j M_{ij}$. Finally, the array builder, vector(n)(L), converts the association list L of type List[(Int,Double)] to a vector of type vector[Double] of size n.

In Query (1), both matrix M and the query result V are stored in memory. However, in our framework, the storage of these arrays can be customized by using a different sparsifier for M and a different builder for the result. More specifically, we want to translate array comprehensions to efficient distributed programs over block arrays, which are distributed bags of dense array chunks. In Spark [4], we can implement a tiled matrix as a distributed collection (an RDD) of fix-sized square tiles of type RDD[((Int,Int),Array[Double])], where each tile ((i,j),A) has coordinates i and j and values stored in the dense matrix A, which has a fixed size N*N, for some constant N. Similarly, we can implement the query result V as a block vector of type RDD[(Int,Array[Double])], where each block (i,B) has a coordinate i and values stored in the vector B of size N. Then, Query (1) can be rewritten as:

V = tiled[(i, +/m) | ((i,j),m) ← M, **group by** i]. (2)

For this query, the implicit sparsifier that converts a tiled matrix M of type RDD[((Int,Int),Array[Double])] to an association list of type List[((Int,Int),Double)] is:

[((ii*N+i, jj*N+j), A(i,j))
| ((ii,jj),A) ← M, i ← 0 **until** N, j ← 0 **until** N],

where the index variable i in 'i ← 0 **until** N' iterates from 0 to N−1. Here, ii and jj are tile coordinates, and i and j are indices within a tile. That is, for each tile A with coordinates ii and jj read from the tiled matrix M, this list comprehension constructs $N * N$ elements from the data stored in A, so that each element has row coordinate ii*N+i, column coordinate jj*N+j, and value A(i,j). On the other hand, the builder tiled(L), which constructs a block array of type RDD[(Int,Array[Double])] from the list L of type List[(Int,Double)], is:

rdd[(i/N, vector(N)(w))
  | (i,v) ← L, **let** w = (i%N, v), **group by** i/N],

where the builder rdd builds an RDD from a list. That is, this comprehension groups the elements (i,v) of L by i/N (the tile coordinate) so that all N pairs (i,v) with the same i/N go into the same tile. After the group-by, the value of w is lifted to a list that contains all the values (i%N,v) that belong to the same tile at location i/N. The builder vector(N)(w), used in Query (1), converts the list w to a vector, which is an array block of size N.

Like array queries, sparsifiers and builders are expressed as comprehensions, but unlike array queries, they use efficient array indexing. This implicit coercion from stored arrays to lists and the building of stored arrays from the results of a comprehension introduce levels of indirection and generate superfluous intermediate structures that need to be removed. This is done by unnesting nested comprehensions into flat comprehensions. As we will show in this paper, after some simple transformations, Query (2) is optimized to the Spark code shown on the right of Figure 1. That is, from every tile A in M with tile coordinates I and J, a new vector block B with a coordinate I is constructed. The tile processing, which takes place at each compute node, is parallelized using the the Scala's par method [20]. Finally, vector blocks with the same coordinate are reduced pairwise using vector addition, addVectors.

In this paper, we present a small set of generic rules for translating array comprehensions to efficient Spark RDD programs that operate on block arrays. These rules are not based on specific array operations but can apply to many array processing programs that can be expressed as array comprehensions. Matrix multiplication, for example, is translated to an optimal block matrix multiplication algorithm by one generic rule that recognizes a certain class of group-by-joins (joins between two datasets followed by a group-by and an aggregation), and translates them to a special block group-by-join algorithm. Our system, called *SAC* (Scalable Array Comprehensions), has been implemented using Scala's compile-time reflection and macros. It translates array comprehensions to Scala code that calls Spark RDD operations whose functional arguments use the Scala's Parallel Collections library for multicore parallelism [20].

In an earlier work [13], we have presented a framework, called DIABLO (a Data-Intensive Array-Based Loop Optimizer), for translating array-based loops to array comprehensions, which in turn are translated to Spark programs expressed in the Spark Core API. The DIABLO input language resembles the syntax of some loop-based imperative languages, such as C and Java. DIABLO can translate any array-based loop expressed in this loop-based language to an equivalent Spark program as long as this loop satisfies some simple syntactic restrictions, which are more permissive than the recurrence restrictions imposed by many current systems. Unlike SAC, DIABLO generates Spark programs that operate on arrays in the coordinate format. In a distributed setting, arrays stored in the coordinate format are known to be less efficient than block arrays because 1) they occupy more space and therefore require more data shuffling to evaluate complex array operations, and 2) they are less amenable to multicore parallelism at each compute node since they store the array elements in random order. Furthermore, the focus of DIABLO is in the translation of imperative programs to array comprehensions, while the focus of SAC is in the translation of array comprehensions to efficient code on customizable array storages, with an emphasis on block arrays. That is, SAC supplements DIABLO and can be used as a drop-in back-end replacement for DIABLO to make it able to work on block arrays. Finally, the work reported in this paper generalizes our earlier work on extending MRQL with array-based computations [11]. It extends the MRQL query optimizer with a GroupByJoin physical operator that generalizes the SUMMA parallel algorithm for matrix multiplication [14], an idea also used in Section 5.4 in the context of block arrays. However, unlike our current framework, this system too is based on arrays stored in the coordinate format.

The contributions of this paper are summarized as follows:

- We introduce a novel comprehension syntax for array computations that can capture many array computations in declarative form and is independent of array storage.
- We describe a translation scheme that translates array comprehensions to efficient imperative programs with memory effects (Sections 2 and 3).
- We extend this translation scheme to generate efficient Spark code from array comprehensions (Section 4).
- We introduce special type transformations to translate comprehensions on block arrays to efficient Spark code that reduces the amount of data shuffling (Section 5).

Expression:

e	$::=$	$[\, e \mid \overline{q} \,]$	comprehension
	$\mid$	$\oplus/e$	reduction using $\oplus$
	$\mid$	$v[e_1,\ldots,e_n]$	array indexing $n \geq 1$
	$\mid$	$\ldots$	other expression

Qualifiers:

$\overline{q}$	$::=$	$q_1,\ldots,q_n$	$n \geq 0$

Qualifier:

q	$::=$	$p \leftarrow e$	generator
	$\mid$	$\mathbf{let}\, p = e$	local declaration
	$\mid$	e	filtering
	$\mid$	$\mathbf{group\ by}\, p\ [\,:\, e\,]$	group-by

Pattern:

p	$::=$	v	pattern variable
	$\mid$	$(p_1,\ldots,p_n)$	tuple $n \geq 0$

Figure 2: Language syntax

$$[\, e_1 \mid p \leftarrow e_2,\, \overline{q} \,] = e_2.\mathrm{flatMap}(\lambda p.\, [\, e_1 \mid \overline{q} \,]) \tag{4}$$

$$[\, e_1 \mid \mathbf{let}\, p = e_2,\, \overline{q} \,] = \mathbf{let}\ p = e_2\ \mathbf{in}\ [\, e_1 \mid \overline{q} \,] \tag{5}$$

$$[\, e_1 \mid e_2,\, \overline{q} \,] = \mathbf{if}\ (e_2)\ \mathbf{then}\ [\, e_1 \mid \overline{q} \,]\ \mathbf{else}\ \mathrm{Nil} \tag{6}$$

$$[\, e \mid \,] = [\, e \,] \tag{7}$$

Figure 3: Desugaring rules

- We evaluate the performance of our system relative to Spark's MLlib.linalg library (Section 6). Based on these results, SAC is up to 6 times faster than MLlib for matrix multiplication and up to 3 times faster than MLlib for matrix factorization.

2 SYNTAX AND SEMANTICS OF ARRAY COMPREHENSIONS

Figure 2 describes the syntax of our language and Figure 3 gives the desugaring rules, which are based on standard methods for translating list comprehensions [23]. The meaning, desugaring rules, and code generation for the group-by syntax are given in Section 3.

Flattening nested comprehensions that do not have a group-by qualifier is done using the following rule:

$$\begin{aligned}&[\, e_1 \mid \overline{q_1},\, p \leftarrow [\, e_2 \mid \overline{q_3} \,],\, \overline{q_2} \,]\\ &\quad = [\, e_1 \mid \overline{q_1},\, \overline{q_3},\, \mathbf{let}\, p = e_2,\, \overline{q_2} \,]\end{aligned} \tag{3}$$

for any sequence of qualifiers $\overline{q_1}$, $\overline{q_2}$, and $\overline{q_3}$. It may require renaming the variables in $[\, e_2 \mid \overline{q_3} \,]$ before we apply this rule to prevent variable capture.

As explained in Section 1.1, abstract arrays in our framework are represented as association lists that uniquely map array indices to values. These abstract representations are mapped to concrete storage structures with the help of a pair of customized functions, a sparsifier and a builder. Then, array comprehensions on abstract arrays are translated to efficient concrete programs on storage structures based on these type mappings. In this and the following section, we describe this program translation in more detail using one specific type mapping that stores a matrix in a flat vector in row-major order. Although this storage structure is not a distributed tiled array, which is the focus of this paper, this example is important for two reasons: First, it illustrates our program translation process in detail using a simpler storage. Second, it is useful for translating comprehensions on block arrays to Spark code (described in Sections 4 and 5) because it shows how the code for tile operations is generated, given that a tile is a matrix.

Consider a matrix M of type Matrix[T] stored in row-major order as a triple (n,m,V) of type (Int,Int,Array[T]), where n and m are the matrix dimensions and V is the vector that contains the matrix elements in row-major order. The following sparsifier converts the storage S of type (Int,Int,Array[T]) to the abstract representation of the matrix M, which is of type List[((Int,Int),T)]:

```
def sparsify[T] ( S: (Int,Int,Array[T]) ): List[((Int,Int),T)]
= [ ((i,j),A(i*n+j)) | let (n,m,A) = S, i ← 0 until n, j ← 0 until m ],
```

where A(i) is array indexing in Scala. The builder matrix(n,m) L takes two groups of parameters. The n and m parameters specify the matrix dimensions, while L is the association list to be converted to a flat vector that contains the matrix values in row-major order:

```
def matrix[T] ( n: Int, m: Int )
                ( L: List[((Int,Int),T)] ): (Int,Int,Array[T])
  = {  val V = Array.ofDim[T](n*m);
       [ V(i*n+j) = v | ((i,j),v) ← L, i≥0, i<n, j≥0, j<m ];
       (n,m,V)  },
```

where Array.ofDim[T](n) creates a new array of size n and V(i) = a is an assignment that updates V(i). The sparsifier function is always named 'sparsify' because, as we will see next, it is implicitly embedded in the code by the compiler by looking at the code type, while the builder must have a unique name or type signature since all builders transform association lists. Nevertheless, the builder too can be inferred by the compiler in certain assignments, such as in the following example. In the following declaration:

```
var M: matrix[Double]
  = matrix(n,m)[ ((i,j),random()) | i ← 0 until n, j ← 0 until m ];
```

the builder matrix(n,m) is required because the M declaration specifies the abstract type, matrix[Double], but not the storage. However, in the following assignment:

```
M = [ ((j,i),m+1) | ((i,j),m) ← M, m > 10 ];
```

the builder can be inferred to be matrix(n,m), since the compiler can infer the storage type of M.

As a running example to illustrate code generation, consider the addition of two matrices M and N of size $n \times m$, expressed as follows using array comprehensions:

```
matrix(n,m)[ ((i,j),a+b) | ((i,j),a) ← M, ((ii,jj),b) ← N,          (8)
                              ii == i, jj == j ].
```

This query can also be expressed as:

```
matrix(n,m)[ ((i,j),a+N[i,j]) | ((i,j),a) ← M ],
```

which is translated to Query (8). Basically, an array indexing $V[e_1,\ldots,e_n]$ in a comprehension is transformed by adding the qualifiers $((k_1,\ldots,k_n),k_0) \leftarrow V,\, k_1 == e_1,\ldots,k_n == e_n$ to the comprehension, where $k_0,k_1,\ldots,k_n$ are fresh variables, and by replacing $V[e_1,\ldots,e_n]$ with k_0. Without such translation, array indexing would not be able to map to operations on the underlying array storage.

Given a generator $p \leftarrow e$ in a comprehension, the compiler will infer the type of e using standard type inference. Then, it will search all defined sparsifiers to find one, if exists, that applies to the type of e, and will embed this sparsifier by replacing e with sparsify(e). For Query (8), the compiler will infer the storage type of M and N to be (Int,Int,Array[Float]) and then will embed the right sparsifiers to convert them to association lists, which for these matrices is the sparsify function defined earlier. Then, it will inline the code of the sparsifiers and builder and will optimize the resulting program. This can be done effectively when these functions are expressed as comprehensions. By expressing these functions as comprehensions, the optimizer can fuse them with the array comprehension of the query, resulting to a comprehension that traverses the array storage directly, without creating the intermediate lists. Furthermore, unlike array comprehensions, these functions can and must use array indexing so that the fused comprehension results to efficient array operations.

In addition to flattening nested comprehensions using Rule (3), the only optimizations needed are those related to index traversals. More specifically, if two index generators i ← 0 **until** n and j ← 0 **until** m are related with i==j, then they are fused to one generator and a let-binding: i ← 0 **until** min(n,m), **let** j = i.

For Query (8), if for simplicity, the inequalities in the matrix(n,m) builder are ignored, we have:

```
matrix(n,m)[ ((i,j),a+b) | ((i,j),a) ← sparsify(M),
                           ((ii,jj),b) ← sparsify(N),
                           ii == i, jj == j ]
      (if we inline the array builder without the inequalities)
= { val V = Array.ofDim[T](n*m);
    [ V(i*n+j) = v
    | ((i,j),v) ← [ ((i,j),a+b) | ((i,j),a) ← sparsify(M),
                                  ((ii,jj),b) ← sparsify(N),
                                  ii == i, jj == j ] ];
    (n,m,V) }
      (if we unnest the comprehension using Rule (3))
= { val V = Array.ofDim[T](n*m);
    [ V(i*n+j) = v
    | ((i,j),a+b) | ((i,j),a) ← sparsify(M),
                    ((ii,jj),b) ← sparsify(N),
                    ii == i, jj == j ];
    (n,m,V) }
      (if we inline the sparsifiers and rename their variables)
= { val V = Array.ofDim[T](n*m);
    [ V(i*n+j) = v
    | ((i,j),a+b) | ((i,j),a) ← [ ((i1,j1),A(i1*n1+j1))
                                 | let (n1,m1,A) = M,
                                   i1 ← 0 until n1,
                                   j1 ← 0 until m1 ],
                    ((ii,jj),b) ← [ ((i2,j2),B(i2*n2+j2))
                                  | let (n2,m2,B) = N,
                                    i2 ← 0 until n2,
                                    j2 ← 0 until m2 ],
                    ii == i, jj == j ];
    (n,m,V) }
      (if we unnest the comprehension using Rule (3))
= { val V = Array.ofDim[T](n*m);
    [ V(i*n+j) = v
    | let (n1,m1,A) = M,
      i1 ← 0 until n1, j1 ← 0 until m1,
      let (n2,m2,B) = N,
      i2 ← 0 until n2, j2 ← 0 until m2,
      i2 == i1, j2 == j1,
      let ((i,j),v) = ((i1,j1),A(i1*n1+j1)+B(i2*n2+j2)) ];
    (n,m,V) }
      (if we merge the array index bounds)
= { val V = Array.ofDim[T](n*m);
    [ V(i*n+j) = A(i1*n1+j1)+B(i1*n2+j1))
    | let (n1,m1,A) = M, let (n2,m2,B) = N,
      i1 ← 0 until min(n1,n2), j1 ← 0 until min(m1,m2) ];
    (n,m,V) }
```

Given that comprehensions over arrays are translated to array index traversals of type scala.collection.immutable.Range in Scala, to parallelize the code, the only transformation needed is to convert the outer index traversal to a parallel traversal of type scala.collection.parallel.immutable.ParRange. This is done by applying par, such as i1 ← (0 **until** min(n1,n2)).par in our example.

Comprehensions can also be used along with total aggregations, such as for checking whether a vector V is sorted:

```
&&/[ v <= w | (i,v) ← V, (j,w) ← V, j == i+1 ],
```

which checks if all consecutive elements of V (i.e., V_i and V_{i+1}) are ordered. The builder of $\oplus/e$ is:

{ **var** b = $\mathbf{1}_\oplus$; [b = (b ⊕ v) | v ← e]; b },

where $\mathbf{1}_\oplus$ is the zero value of the monoid ⊕. Similar to the matrix sparsifier, the vector sparsifier is:

```
def sparsify[T] ( V: Array[T] ): List[(Int,T)]
  = [ (i,V(i)) | i ← 0 until V.length ].
```

If we embed the sparsifiers, unfold the builder code, and unnest the list comprehensions, the array comprehension becomes:

```
{ var b = true;
  [ b = (b && (V(i) <= V(j))) | i ← 0 until V.length,
                                j ← 0 until V.length, j == i+1 ];
  b },
```

which is further optimized to:

```
{ var b = true;
  [ b = (b && (V(i) <= V(i+1))) | i ← 0 until V.length−1 ];
  b },
```

given that min(V.length,V.length−1) = V.length−1.

3 COMPREHENSIONS WITH A GROUP-BY

Consider the product of two matrices M and N with dimensions $n \times l$ and $l \times m$, respectively, which is equal to a matrix C so that $C_{ij} = \sum_k M_{ik} * N_{kj}$. Using our array comprehensions enhanced with a group-by syntax, matrix multiplication can be expressed as follows:

```
matrix(n,m)[ ((i,j),+/v) | ((i,k),a) ← M, ((kk,j),b) ← N,
                           kk == k, let v = a*b,              (9)
                           group by (i,j) ].
```

This comprehension retrieves the values M_{ik} and N_{kj} and sets $v = M_{ik} * N_{kj}$. After we group the values by the matrix indices i and j, the variable v is lifted to a bag of numerical values $M_{ik} * N_{kj}$, for all k. Hence, the aggregation $+/v$ sums up all the values in the bag v, deriving $\sum_k M_{ik} * N_{kj}$ for the ij element of the resulting matrix.

Another example of a group-by comprehension is matrix smoothing for a matrix M, which is a matrix C such that C_{ij} =

$\frac{1}{9}\sum_{i-1\leq I\leq i+1}\sum_{j-1\leq J\leq j+1} M_{IJ}$. That is, C_{ij} is the average value in the neighborhood of M_{ij}:

```
matrix(n,m)[ ((ii,jj),(+/a)/a.length)
           | ((i,j),a) ← M,
             ii ← (i−1) to (i+1), jj ← (j−1) to (j+1),
             ii >= 0, ii < n, jj >= 0, jj < m,
             group by (ii,jj) ],
```

which also takes care of the boundary cases.

Given a pattern p that consists of bound pattern variables, the qualifier **group by** p in $[\, e \mid \overline{q_1}$, **group by** $p, \overline{q_2}\,]$ groups every pattern variable in $\overline{q_1}$ (except the variables in p) by the group-by key p into a list that contains all the values of this variable associated with this group-by key. Furthermore, the qualifier **group by** $p : e$ is syntactic sugar for the qualifiers **let** $p = e$, **group by** p. Group-by qualifiers may appear in multiple places in a comprehension. For all these cases, only the pattern variables that precede the group-by qualifier in the same comprehension must be lifted to lists, and if multiple group-by qualifiers exist, these variables will have to be lifted multiple times to nested lists.

Group-by qualifiers are translated to groupBy operations before a comprehension is translated by the rules in Figure 2. Given a list s of type $\text{List}[(K,V)]$, the operation groupBy(s) groups the elements of s by their first component (the group-by key) and returns an association list of type $\text{Map}[K,\text{List}[V]]$, which is implemented as a hash table:

```
def groupBy[K,V] ( s: List[(K,V)] ): Map[K,List[V]]
  = { val m = Map[K,List[V]]();
      [ m(k) = if (m.contains(k)) m(k):+v else List(v)          (10)
      | (k,v) ← s ];
      m }.
```

Let $\overline{v} = (v_1, \ldots, v_n)$ be the pattern variables in the sequence of qualifiers $\overline{q_1}$ that are used in the rest of the comprehension $[\, e \mid \overline{q_2}\,]$ but do not appear in the group-by pattern p. Then, the group-by syntax is translated as follows:

$$\begin{aligned}
&[\, e \mid \overline{q_1},\ \textbf{group by}\, p,\ \overline{q_2}\,] \\
&\quad = [\, e \mid (p,s) \leftarrow \text{groupBy}([\,(p,\overline{v}) \mid \overline{q_1}\,]), \\
&\qquad\quad \textbf{let}\ \overline{v} = \text{unzip}(s),\ \overline{q_2}\,],
\end{aligned} \tag{11}$$

where unzip$(s) = ([\, v_1 \mid \overline{v} \leftarrow s\,], \ldots, [\, v_n \mid \overline{v} \leftarrow s\,])$. That is, each pattern variable v_i in $\overline{v}$ is lifted to a list that contains all the values of v_i in the current group.

For example, the matrix multiplication in Query (9) has the following meaning:

```
matrix(n,m)[ ((i,j),(+/v)) | ((i,j),s) ← groupBy(S), let v = s ],
```

where S is:

```
[ ((i,j),v) | ((i,k),a) ← M, ((kk,j),b) ← N, kk == k, let v = a*b ]
```

since the only variable lifted is v with v = [v | v ← s], which is equal to s.

Array comprehensions with a group-by syntax allow more array operations to be expressed declaratively without having to use array indexing and loops. They also make comprehensions equivalent to basic SQL queries. As we will show, although it has pure semantics, the group-by syntax makes it easier to recognize certain code patterns in a comprehension and translate them to efficient code. For example, we will see next that the matrix multiplication in Query (9) is translated to the following code, which is as efficient as a program hand-coded in an imperative language:

```
{ val V = Array.ofDim[T](n*m)(0.0);
  [ V(i*n+j) += A(i*n+k)*B(k*l+j)
  | let (n,l,A) = M, let (ll,m,B) = N,
    i ← 0 until n, k ← 0 until l, j ← 0 until m ];
  (n,m,V) },
```

which is equivalent to a triple loop with body V_{ij} += $A_{ik} \times B_{kj}$.

Consider the general comprehension $[\, e \mid \overline{q_1}$, **group by** $p, \overline{q_2}\,]$. To simplify our translation rules, we rewrite this term to $[\, z \mid \overline{q_1}$, **group by** $p,\ z \leftarrow [\, e \mid \overline{q_2}\,]\,]$. Let $\mathcal{V} = \{v_1, \ldots, v_n\}$ be the set of variables that are lifted by the group-by but are not lifted or redefined in $\overline{q_2}$ and let $\overline{v} = (v_1, \ldots, v_n)$. Recall that these variables are lifted to lists that contain all the values of these variables associated with the group-by key p. A lifted variable may occur any number of times in the rest of the comprehension, $[\, e \mid \overline{q_2}\,]$. Let $w_1, \ldots, w_m$ be the occurrences of the lifted variables in $[\, e \mid \overline{q_2}\,]$. A lifted variable $w_i \in \mathcal{V}$ may occur in $[\, e \mid \overline{q_2}\,]$ 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 first two do not match. All these cases can be generalized to $\oplus_i/w_i.\text{map}(g_i)$, for some monoid $\oplus_i$ and function g_i. Hence, we can represent the term after group-by as follows:

$$\begin{aligned}
[\, e \mid \overline{q_2}\,] &= f(\oplus_1/w_1.\text{map}(g_1), \ldots, \oplus_n/w_m.\text{map}(g_m)) \\
&= f(\otimes/\text{zip}(\overline{w}).\text{map}(g)),
\end{aligned}$$

for some term function f, where $\overline{w} = (w_1, \ldots, w_m)$, $g = g_1 \times \cdots \times g_m$, and $\otimes = \oplus_1 \times \cdots \times \oplus_m$ (a product of monoids[1]). Then, based on the Rule (11) and the implementation of groupBy in definition (10), we have:

$$\begin{aligned}
&[\, e \mid \overline{q_1},\ \textbf{group by}\ p,\ \overline{q_2}\,] \\
&= [\, z \mid \overline{q_1},\ \textbf{group by}\ p,\ z \leftarrow f(\otimes/\text{zip}(\overline{w}).\text{map}(g))\,] \\
&= \{\ \textbf{val}\ M = \text{Map}(); \\
&\qquad [\, M(p) = \textbf{if}\ (M.\text{contains}(p))\ M(p) \otimes g(\overline{w})\ \textbf{else}\ g(\overline{w}) \mid \overline{q_1}\,]; \\
&\qquad [\, z \mid p \leftarrow M.\text{keys},\ z \leftarrow f(M(p))\,]\ \}
\end{aligned} \tag{12}$$

Consider now an array comprehension of the form:

$$\text{matrix}(n,m)[\, ((i,j),e) \mid \overline{q_1},\ \textbf{group by}\ (i,j),\ \overline{q_2}\,],$$

Notice that, here, the matrix index (i,j) in the comprehension head is the group-by key. In the implementation (12) of an array comprehension with group-by, we can now use arrays of size n*m, one array for each aggregation, instead of a Map:

$$\begin{aligned}
&\text{matrix}(n,m)[\, ((i,j),e) \mid \overline{q_1},\ \textbf{group by}\ (i,j),\ \overline{q_2}\,] \\
&= \text{matrix}(n,m)[\, z \mid \overline{q_1},\ \textbf{group by}\ (i,j), \\
&\qquad\qquad z \leftarrow f(\oplus_1/w_1.\text{map}(g_1), \ldots, \oplus_n/w_n.\text{map}(g_n))\,] \\
&= \{\ \textbf{val}\ V_1 = \text{Array.fill}(n*m)(\mathbf{1}_{\oplus_1}); \\
&\qquad \ldots \\
&\qquad \textbf{val}\ V_n = \text{Array.fill}(n*m)(\mathbf{1}_{\oplus_n}); \\
&\qquad [\ \{\ V_1(i*n+j) = V_1(i*n+j) \oplus_1 g_1(v_1); \\
&\qquad\qquad \ldots \\
&\qquad\qquad V_n(i*n+j) = V_n(i*n+j) \oplus_n g_n(v_n)\ \}\ \mid \overline{q_1}\,]; \\
&\qquad \text{matrix}(n,m)[\, z \mid ((i,j),\_) \leftarrow V_1,
\end{aligned}$$

[1]That is, the monoid $\otimes$ with identity $\mathbf{1}_\otimes = (\mathbf{1}_{\oplus_1}, \ldots, \mathbf{1}_{\oplus_m})$ and $(x_1, \ldots, x_m) \otimes (y_1, \ldots, y_m) = (x_1 \oplus_1 y_1, \ldots, x_m \oplus_m y_m)$.

```
                    z ← $f(V_1(i*n+j),\dots,V_n(i*n+j))$ ] }.
```

Notice that, the final result is a matrix constructed using the $\text{matrix}(n,m)$ builder, and is made out of the arrays that hold the aggregation results. This matrix though does not have a group-by and can be translated using the methods given in Section 2.

For example, the matrix multiplication in Query (9) is translated as follows:

```
matrix(n,m)[ ((i,j),+/v)
             | ((i,k),a) ← sparsify(M), ((kk,j),a) ← sparsify(N),
               kk == k, let v = a*b, group by (i,j) ],
      (after unfolding the sparsifiers for M and N)
= matrix(n,m)[ ((i,j),+/v)
             | let (n,l,A) = M, i ← 0 until n, k ← 0 until l,
               let (ll,m,B) = N, kk ← 0 until ll, j ← 0 until m,
               kk == k, let v = A(i*n+k)*B(kk*ll+j), group by (i,j) ]
      (after merging the array index kk with k)
= matrix(n,m)[ ((i,j),+/v)
               | let (n,l,A) = M, i ← 0 until n, k ← 0 until l,
                 let (ll,m,B) = N, j ← 0 until m,
                 let v = A(i*n+k)*B(k*ll+j), group by (i,j) ]
      (by translating the group-by qualifier)
= { val V = Array.fill(n,m)(0.0);
    [ V(i*n+j) = V(i*n+j) + A(i*n+k)*B(k*ll+j)
    | let (n,l,A) = M, i ← 0 until n, k ← 0 until l,
      let (ll,m,B) = N, j ← 0 until m ];
    matrix(n,m)[ v | ((i,j),_) ← V, v ← [ ((i,j),V(i*n+j)) |] ] }
= { val V = Array.fill(n,m)(0.0);
    [ V(i*n+j) = V(i*n+j) + A(i*n+k)*B(k*ll+j)
    | let (n,l,A) = M, i ← 0 until n, k ← 0 until l,
      let (ll,m,B) = N, j ← 0 until m ];
    matrix(n,m)[ ((i,j),V(i*n+j)) | ((i,j),_) ← V ] }
    (since the last term is equal to (n,m,V))
= { val V = Array.fill(n,m)(0.0);
    [ V(i*n+j) = V(i*n+j) + A(i*n+k)*B(k*ll+j)
    | let (n,l,A) = M, i ← 0 until n, k ← 0 until l,
      let (ll,m,B) = N, j ← 0 until m ];
    (n,m,V) }
```

which is equivalent to the desired efficient loop-based program.

4 TRANSLATING QUERIES ON SPARK

In this section, we translate array comprehensions to distributed programs that can run on Apache Spark [27]. Distributed datasets in Spark are represented as Resilient Distributed Datasets (RDDs), which support a functional API that is very similar to that for Scala collections. Most RDD operations are second-order, in which the functional argument is evaluated sequentially while the operation itself is evaluated in parallel, in a distributed mode. Unlike Scala collections, Spark does not allow nested RDDs and will raise a run-time error if the functional parameter of an RDD operation accesses another RDD. That is, Spark does not support nested parallelism because it is hard to implement efficiently in a distributed setting. However, instead of using nested RDD operations, one may use joins, cogroups, and cross products to correlate RDDs. Consequently, RDD comprehensions require special translation rules to derive joins, instead of nested flatMaps.

The RDD builder, rdd, that converts a List[T] to an RDD[T] can be implemented by applying the Spark method 'parallelize' on this list. However, RDD comprehensions must be translated to RDD operations in a special way to avoid generating nested operations. A group-by qualifier can be translated to the Spark groupByKey operation of type RDD[(K,V)]⇒RDD[(K,List[V])] using Equation (11). However, groupByKey is an expensive operation because it collects the grouped values into a list, shuffles these lists to the reducers, and finally reduces them by some aggregation. Instead, we want to generate calls to the more efficient reduceByKey(⊗), for some monoid ⊕, that reduces the values of type V using the monoid ⊗, instead of placing them into a list. That way, grouped values are partially reduced before they are shuffled. To generate these reduceByKey calls, we consider the group-by qualifier in combination with the aggregations in the comprehension. Recall that, based on the discussion in Section 3 and on Equation (12), any comprehension with a group-by can be put into the following form:

$$\begin{aligned}
&\text{rdd}[\, e \mid \overline{q_1},\ \textbf{group by}\ p,\ \overline{q_2}\,] \\
&\quad = \text{rdd}[\, z \mid \overline{q_1},\ \textbf{group by}\ p,\ z \leftarrow [\, e \mid \overline{q_2}\,]\,] \\
&\quad = \text{rdd}[\, z \mid \overline{q_1},\ \textbf{group by}\ p, \\
&\qquad\qquad z \leftarrow f(\oplus_1/w_1.\text{map}(g_1),\dots,\oplus_m/w_m.\text{map}(g_m))\,],
\end{aligned}$$

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 can be translated to a reduceByKey operation:

$$\begin{aligned}
&\text{rdd}[\, e \mid \overline{q_1},\ \textbf{group by}\ p,\ \overline{q_2}\,] \\
&\quad = \text{rdd}[\, (p,(g_1(w_1),\dots,g_m(w_m))) \mid \overline{q_1}\,] \\
&\qquad .\text{reduceByKey}(\otimes) \qquad\qquad\qquad\qquad (13) \\
&\qquad .\text{map}\{\ \textbf{case}\ (p,(a_1,\dots,a_m)) \Rightarrow f(a_1,\dots,a_m)\ \},
\end{aligned}$$

where $\otimes = \oplus_1 \times \cdots \times \oplus_m$.

The following rule identifies and generates joins between the RDDs X and Y, instead of nested flatMaps, 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:

$$\begin{aligned}
&\text{rdd}[\, e \mid \overline{q_1},\ p_1 \leftarrow X,\ \overline{q_2},\ p_2 \leftarrow Y,\ \overline{q_3},\ e_1 == e_2,\ \overline{q_4}\,] \\
&\quad = \text{rdd}[\, e \mid \overline{q_1},\ (\_,(p_1,p_2)) \leftarrow Z, \overline{q_2},\ \overline{q_3},\ \overline{q_4}\,], \qquad (14)
\end{aligned}$$

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

One way to represent arrays in a distributed setting is to store them as coordinate arrays, similar to the array representation used in Section 2. For instance, a matrix can be defined in Spark as an RDD of type RDD[((Long,Long),Double)], while matrix multiplication of two RDD matrices A and B, which was defined in (9), will be translated to the following program using Rules (14) and (13):

```
A.map{ case ((i,k),a) ⇒ (k,((i,k),a)) }
  .join( B.map{ case ((kk,j),b) ⇒ (kk,((kk,j),b)) } )
  .map{ case (_,(((i,k),a),((kk,j),b))) ⇒ ((i,j),a*b) }
  .reduceByKey(_+_).
```

Although correct, this Spark program has a high cost: it shuffles the matrices A and B across the compute nodes to perform the join, and then it shuffles all the products $A_{ik} * B_{kj}$ to perform the reduceByKey. Since data shuffling is the main cost factor for a distributed program, instead of fully sparse matrices in the coordinate format, we want to use a more compact representation for matrices by partitioning a matrix into tiles, which are unboxed arrays of type Array[Double] in which indices are calculated, not stored. The sparse matrix representation, on the other hand, is preferable when both dimensions of the matrix are large and the matrix is very sparse.

5 TRANSLATING BLOCK ARRAY QUERIES

A more effective way of representing an array in a distributed setting is to encode it as a distributed bag of non-overlapping blocks, where each block is a fix-sized chunk of the distributed array. A block is the unit of data distribution. Our goal in this section is to translate RDD comprehensions over block arrays to Spark's distributed data-parallel programs whose functional parameters will process the blocks very efficiently using multi-core parallelism and array indexing. Given that data partitioning is necessary for data-parallel distributed processing, blocks are a natural way to partition large arrays and at the same time to minimize space overhead, compared to fully sparse arrays in coordinate format, in which the indices are stored along with a matrix element. This small space footprint translates to less data to shuffle across nodes and faster time to process each partition. Spark actually uses thread-level parallelism at each compute node to process the elements of each RDD partition in parallel using multicore parallelism, but the unit of parallelism for a block array is the entire block, which is likely to be one block for each compute node. Consequently, in addition to generating distributed operations from array comprehensions on block arrays, our goal is to process the data inside blocks using multicore parallelism.

In this paper, we focus on tiled matrices but our work can be easily extended to handle other block arrays too. We represent a tiled matrix using the following Scala class:

```
case class Tiled[T] ( rows: Long, cols: Long,
                      tiles: RDD[((Long,Long),Array[T])] ),
```

where rows is the number of rows, cols is the number of columns, and tiles is an RDD of fix-sized square tiles, where each tile ((i,j),A) has coordinates i and j and values stored in the array A. The array A has a fixed size N*N, for some constant N which is the same for all tiles. The coordinates i and j of a tile are unique, that is, tiles is an association list. A matrix element with indices ij is stored in the tile that has coordinates $(i/N, j/N)$ at the location $(i\%N) * N + (j\%N)$ inside the tile. The tile sparsifier is as follows:

```
def sparsify[T] ( S: Tiled[T] ): List[((Long,Long),T)]
  = [ ( ( ii*N+i, jj*N+j ), a(i*N+j) )
    | ((ii,jj),a) ← S.tiles,
      i ← 0 until N, j ← 0 until N ],
```

where ii and jj are the tile coordinates, and i and j are indices within a tile. The tiled builder uses the rdd and the array builders:

```
def tiled[T] ( n: Long, m: Long )
             ( L: List[((Long,Long),T)] ): Tiled[T]
  = Tiled( n, m, rdd[  ( (ii, jj), array(N*N)(w) )
                     | ((i,j),v) ← L, let ii = i/N, let jj = j/N,
                       let w = ( (i%N)*N+(j%N), v ),
                       group by (ii,jj) ] ),
```

where the group-by collects all tile elements into an array. The group-by comprehension is in an RDD comprehension, which means that it will be translated to a groupByKey operation in Spark, which requires data shuffling across the compute nodes. However, in some cases, this group-by qualifier can be eliminated, as in the case of a map over a matrix. Such an optimization is actually a general optimization over comprehensions with a group-by. A group-by qualifier in a comprehension can be eliminated if the group-by key is unique, that is, when the group-by function is injective. Although it is in general undecidable to prove whether a group-by key is unique, it is easy to do so for special cases, such as when the group-by key consists of array indices that are bound through an array traversal. For an array or map A, and the pattern variables v_i in $\overline{q_1}$ or $\overline{q_2}$, we have:

$$\begin{aligned}&[\, e \mid \overline{q_1},\ (k,v) \leftarrow A,\ \overline{q_2},\ \textbf{group by}\ k\,] \\ &\quad = [\, e \mid \overline{q_1},\ (k,v) \leftarrow A,\ \textbf{let}\ \overline{v} = \text{unzip}([\, \overline{v} \mid \overline{q_2}\,])\,],\end{aligned} \tag{15}$$

since the generator $(k,v) \leftarrow A$ for an array or map A indicates that k is unique. That is, this group-by is removed and every pattern variable v_i in $\overline{q_1}$ or $\overline{q_2}$ is lifted to a bag that contains all its values in the group. A similar rule exists for a generator $k \leftarrow e_1$ **until** e_2, since every value of k is unique.

Although correct, unfolding and normalizing tiled array comprehensions based on the tiled sparsifier and builder do not always result to optimal translations. In the rest of this section, we present special rules to translate tiled array comprehensions to efficient Spark programs.

5.1 Queries that Preserve Tiling

Consider the block comprehension without a group-by:

$$\text{tiled}(\overline{d})[\, (key, e) \mid \overline{q}\,] \tag{16}$$

where $\overline{d}$ is the tile dimensions (e.g., (m,n) for a matrix), key is the tile indices (e.g., (i,j) for a matrix) and e is the associated value. Let $(\overline{k_i}, v_i) \leftarrow X_i$ be a generator over a tiled array X_i in $\overline{q}$, where $\overline{k_i}$ is a tuple of index variables, such as $((i,j),v) \leftarrow X$ for a tiled matrix X. We say that this tiled comprehension *preserves tiling* if key is a tuple $\overline{w}$ that consists of variables that are defined in the tuples $\overline{k_i}$. The rest of the variables in the tuples $\overline{k_i}$ (not in $\overline{w}$) must be related to the variables in $\overline{w}$ with equality predicates in $\overline{q}$, so that the index of the constructed array is unique. For example, matrix addition and matrix diagonal preserve tiling:

```
tiled(n,m)[ ((i,j),a+b) | ((i,j),a) ← A, ((ii,jj),b) ← B,
                          ii == i, jj == j ],
tiled(n)[ (i,a) | ((i,j),a) ← A, i == j ].
```

A comprehension that preserves tiling is translated to an RDD comprehension that does not need a group-by to shuffle tiles. More specifically, the comprehension (16) is translated to:

$$\text{Tiled}(\overline{d}, \text{rdd}[\, (\overline{w}, \text{array}(N*N)[\, (\overline{w}, e) \mid \overline{q_2}\,]) \mid \overline{q_1}\,]), \tag{17}$$

where the qualifiers in $\overline{q_1}$ are those from $\overline{q}$ that do not refer to the tile values and each tiled generator $(\overline{k_i}, v_i) \leftarrow X_i$ has been modified to be $(\overline{k_i}, \_v_i) \leftarrow X_i$.tiles (given the pattern variable v_i bound to an array value, $\_v_i$ is bound to the entire tile in $\overline{q_1}$). The qualifier list $\overline{q_2}$ is equal to $\overline{q}$ but each tiled generator $(\overline{k_i}, v_i) \leftarrow X_i$ in $\overline{q}$ has been modified to be $(\overline{k_i}, v_i) \leftarrow \_v_i$. For example, matrix addition:

```
tiled(n,m)[ ((i,j),a+b) | ((i,j),a) ← A, ((ii,jj),b) ← B,
                          ii == i, jj == j ]
```

is translated to:

```
Tiled( n, m, rdd[ ( (i,j), V(_a,_b) )
                | ((i,j),_a) ← A.tiles, ((ii,jj),_b) ← B.tiles,
                  ii == i, jj == j ] ),
```

where V(_a,_b) is:

```
array(N*N)[ ((i,j),a+b) | ((i,j),a) ← _a, ((ii,jj),b) ← _b,
                          ii == i, jj == j ],
```

which is a regular array comprehension in which the tiles _a and _b will be lifted using the following array sparsifier for a tile A:

```
[ ((i,j),A(i*N+j)) | i ← 0 until N, j ← 0 until N ].
```

Based on the translation of RDD and array comprehensions, matrix addition is translated to the following Spark code:

```
Tiled( n, m, A.tiles.join(B.tiles)
        .map{ case ((ii,jj),(_a,_b)) ⇒ ((ii,jj),V(_a,_b)) } ).
```

After optimizations similar to those for matrix addition for regular matrices, we get the following code for V(_a,_b):

```
{ val V = Array.ofDim[Double](N*N);
  [ V((i%N)*N+(j%N)) = _a( (i%N)*N+(j%N) )
                      + _b( (i%N)*N+(j%N) )
  | i ← (0 until N).par, /* multicore parallelism */
    j ← 0 until N ];
  V }.
```

5.2 Queries that do not Preserve Tiling

For those array comprehensions that do not have a group-by and do not preserve tiling, we need to shuffle only the relevant tiles to the appropriate reducers. The array indices $\overline{w}$ in the result of the tiled comprehension

$$\text{tiled}(\overline{d})[\,(\overline{w},e)\mid\overline{q}\,] \qquad (18)$$

may now be arbitrary expressions that depend on the indices of the tiled generators in $\overline{q}$.

Let $f(\overline{k})$ be a term that depends on the array indices $\overline{k} = k_1,\dots,k_m$ from the tiled generators in $\overline{q}$. The tiles accessed from the tiled generators $\overline{q}$ will have tile coordinates $\overline{K}$, where $K_i = k_i/N$. Given the tile coordinates $\overline{K}$ of the input tiles, the tile coordinate returned by $f(\overline{k})$ would be equal to $f(K_1*N+j_1,\dots,K_m*N+j_m)/N$, for $j_i\in[0,N)$ (the tile dimensions). We define, $\mathcal{I}_f(\overline{K})$ to be the set of all such tile coordinates:

$$\text{set}[\,f(K_1*N+j_1,\dots,K_m*N+j_m)/N\mid j_1\leftarrow 0\,\textbf{until}\,N,\ \dots,\ j_m\leftarrow 0\,\textbf{until}\,N\,],$$

where set(s) returns the distinct values of s. For example, if $f(k)=k+1$, then $\mathcal{I}_f(K)=\text{set}[\,f(K*N+j)/N\mid j\leftarrow 0\,\textbf{until}\,N\,]$, which is equal to the set $\{K,K+1\}$ since $f(K*N+j)/N=(K*N+j+1)/N=K+(j+1)/N$. On the other hand, if $f(k)=k$, then $\mathcal{I}_f(K)=\{K\}$.

Based on this definition, comprehension (18) can be transformed to:

$$\text{Tiled}(\overline{d},\text{rdd}[\,(\overline{K},V)\mid\overline{q_1},\,K_1\leftarrow\mathcal{I}_{w_1}(\overline{k}),\,\dots,\,K_m\leftarrow\mathcal{I}_{w_m}(\overline{k}),\ \textbf{group by}\,\overline{K}\,]), \qquad (19)$$

where V is:

$$\text{array}[\,(\overline{w},e)\mid\overline{q_2},\ \textstyle\bigwedge_i\overline{K_i}==w_i(\overline{\_k}*N+\overline{k})/N\,].$$

The qualifiers in $\overline{q_1}$ are those from $\overline{q}$ that do not refer to the tile values and each tiled generator $(\overline{k_i},v_i)\leftarrow X_i$ has been transformed to the two qualifiers $(\overline{k_i},\_v_i)\leftarrow X_i$.tiles and **let** $\_\_v_i=(\overline{k_i},\_v_i)$ (given the pattern variable v_i bound to an array value, $\_v_i$ is bound to the entire tile in $\overline{q_1}$, while $\_\_v_i$ is bound to the index-tile pair). The group-by operation shuffles all the required tiles to the reducers to compute the resulting tiles. The qualifier list $\overline{q_2}$ in V is equal to $\overline{q}$ but each tiled generator $(\overline{k_i},v_i)\leftarrow X_i$ in $\overline{q}$ has been transformed to the two qualifiers $(\overline{\_k_i},\_v_i)\leftarrow\_\_v_i$ and $(\overline{k_i},v_i)\leftarrow\_v_i$. The guards $\overline{K_i}==w_i(\overline{\_k}*N+\overline{k})/N$ selects only the proper tiles from all those shuffled by group-by.

For example, the following comprehension rotates the rows so that the first row is moved to the second, the second to third, etc, and the last to the first:

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

The shuffled tiles for a tile in X with coordinates (i,j) have row coordinates from set[(i*N+_i+1)%m/N | _i ← 0 until N], which will be evaluated to List(i,i+1) for i < n/N and to List(i,0) for i = n/N, and column coordinates from List(j). That is, for each tile with coordinates (i,j) such that i < n/N, we require two tiles: the tile itself and its row successor with coordinates (i+1,j). Hence, the row rotation is translated to:

```
Tiled(n,m,rdd[ ( (K1,K2), V )
             | ((i,j),_a) ← X.tiles, let __a = ((i,j),_a),
               K1 ← set[ (i*N+_i+1)%m/N | _i ← 0 until N ],
               K2 ← List(j),
               group by (K1,K2) ] ),
```

which is translated to a Spark groupByKey operation. V is:

```
array(N*N)[ (((i+1)%m,j),a) | ((_i,_j),_a) ← __a, ((i,j),a) ← _a,
            K1 == (_i*N+i+1)%m/N, K2 == (_j*N+j)/N ] ),
```

which is translated to an efficient array-based program over the list of tiles __a.

5.3 Queries with a Group-By

Tile comprehensions with a group-by do not preserve tiling. They can be translated in a way similar to that for comprehension (18), but we can do better by using the RDD operation reduceByKey instead of the RDD operation groupByKey, because a group-by in a group-by comprehension is often followed by aggregation. A reduceByKey(⊕) operation, for some monoid ⊕, is equivalent to a groupByKey followed by reduction of each group using ⊕. Although functionally equivalent, reduceByKey is far more efficient than groupByKey in a distributed setting because it partially reduces the groups locally before they are shuffled to the reducers for the final reduction, resulting to less data shuffling.

Before we describe the general translation scheme, we explain the key idea using the example of matrix multiplication of the tiled matrices A and B:

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

We want to translate it to the reduceByKey operation:

```
Tiled( n, m, rdd[ ((i,j),V(_a,_b))
                | ((i,k),_a) ← A.tiles, ((kk,j),_a) ← B.tiles,
                  kk == k ].reduceByKey(⊕) ),
```

where the tile V(_a,_b) is the tile _a multiplied by _b:

```
array(N*N)[ ((i,j),+/v)) | ((i,k),a) ← _a, ((kk,j),a) ← _b,
                           kk == k, let v = a*b,
                           group by (i,j) ]
```

and the monoid ⊕ over the tiles _x and _y adds the tiles pairwise:

```
_x⊕_y = array(N*N)[ ((i,j),x+y) | ((i,j),x) ← _x, ((ii,jj),y) ← _y,
                                  ii == i, jj == j ].
```

That is, the rdd comprehension calculates the partial sum of products of all matching tiles and the reduceByKey calculates the final sums by adding the tiles pairwise. Then, after translating the rdd and array comprehensions, we will get:

```
Tiled(n,m,A.tiles.map{ case ((i,k),_a) ⇒ (k,((i,k),_a)) }
    .join( B.tiles.map{ case ((kk,j),_a) ⇒ (kk,((kk,j),_a)) } )
    .map{ case (_,(((i,k),_a),((kk,j),_a))) ⇒ ((i,j),V(_a,_b)) }
    .reduceByKey(⊕))
```

where V(_a,_b) is translated to efficient code that multiplies the tiles _a and _b:

```
{ val V = Array.ofDim[Double](N*N);
  for { i ← 0 until N; j ← 0 until N; k ← 0 until N }
     V(i*N+j) += _a(i*N+k)*_b(k*N+j);
  V },
```

and _x⊕_y is pairwise addition:

```
{ val V = Array.ofDim[Double](N*N);
  for { i ← 0 until N; j ← 0 until N }
     V(i*N+j) = _x(i*N+j)+_y(i*N+j);
  V }.
```

To generate these reduceByKey calls, we consider the group-by qualifier in combination with the aggregations in the comprehension. Recall that, based on the discussion in Section 3 and on Equation (12), any tiled comprehension with a group-by can be put into the following form:

$$\begin{aligned}&\text{tiled}(n,m)[\,(p,e) \mid \overline{q},\ \textbf{group by}\ p,\ \overline{q'}\,]\\ &\quad= \text{tiled}(n,m)[\,(p,z) \mid \overline{q},\ \textbf{group by}\ p,\ z \leftarrow [\,e \mid \overline{q'}\,]\,]\\ &\quad= \text{tiled}(n,m)[\,(p,z) \mid \overline{q},\ \textbf{group by}\ p,\\ &\qquad\qquad z \leftarrow f(\oplus_1/w_1.\text{map}(g_1),\dots,\oplus_m/w_m.\text{map}(g_m))\,],\end{aligned}$$

for some variables w_i lifted by group-by, some monoids $\oplus_i$, and some functions g_i and f. That is, we abstract all reductions $\oplus_i/w_i.\text{map}(g_i)$ from the term $[\,e \mid \overline{q'}\,]$. Notice that, here, the key of the tiled comprehension is equal to the group-by key, p. Then, the group-by comprehension can be translated to the following reduceByKey operation:

$$\begin{aligned}\text{Tiled}(n,m,\text{rdd}[\,(p,(&\text{array}(N*N)[\,g_1(w_1) \mid \overline{q_2}\,],\dots,\\ &\text{array}(N*N)[\,g_m(w_m) \mid \overline{q_2}\,]))\\ &\mid \overline{q_1}\,]\\ &.\text{reduceByKey}(\otimes').\text{mapValues}(f'))\end{aligned}$$

Like in (17), the qualifiers in $\overline{q_1}$ are those from $\overline{q}$ that do not refer to the tile values and each tiled generator $(\overline{k_i},v_i) \leftarrow X_i$ has been modified to be $(\overline{k_i},\_v_i) \leftarrow X_i$.tiles. The qualifier list $\overline{q_2}$ is equal to $\overline{q}$ but each tiled generator $(\overline{k_i},v_i) \leftarrow X_i$ in $\overline{q}$ has been modified to be $(\overline{k_i},v_i) \leftarrow \_v_i$. The monoid $\otimes'$ is:

$$(x_1,\dots,x_m) \otimes' (y_1,\dots,y_m) \;=\; (x_1 \oplus'_1 y_1,\dots,x_m \oplus'_m y_m)$$

where $x_k \oplus'_k y_k$ applies $\oplus_k$ to the tile elements pairwise:

$$\begin{aligned}\text{array}(N*N)[\,a \oplus_k b \mid ((i,j),a) \leftarrow x_k,\ ((ii,jj),b) \leftarrow y_k,\\ ii == i,\ jj == j\,]\end{aligned}$$

Finally, $f'(x_1,\dots,x_m)$ is:

$$\begin{aligned}\text{array}(N*N)[\,&f(a_1,\dots,a_m)\\ &\mid ((i_1,j_1),a_1) \leftarrow x_1,\dots,\ ((i_m,j_m),a_m) \leftarrow x_m,\\ &\quad i_1 == \cdots == i_m,\ j_1 == \cdots == j_m\,]\end{aligned}$$

For the matrix multiplication example, there is only one reduction with $\oplus_1 = +$ and f is identity, which means that mapValues(f') is identity too.

5.4 Using a Group-By-Join

There is a special class of tiled comprehensions with a group-by that can be translated to more efficient code. Consider the following comprehension:

```
tiled(n,m)[ (k,⊕/c) | ((i,j),a) ← A, ((ii,jj),b) ← B,
                    kx(i,j) == ky(ii,jj), let c = h(a,b),
                    group by k: ( gx(i,j), gy(ii,jj) ) ],
```

for some arbitrary term functions kx, ky, gx, gy, and h, and some aggregation ⊕. Notice that the key k of the output matrix is the group-by key, which must be a pair where one component depends on i,j and the other on ii,jj only. This is called a group-by-join because it is a join between A and B, followed by a group-by with aggregation. Matrix multiplication, defined in (9), is an example of a group-by-join. Many group-by comprehensions can be put in the group-by-join form by combining generators in pairs forming joins until we are left with two generators and a group-by. A group-by-join can be evaluated very efficiently by joining each row of tiles from A with each column of tiles from B, which requires that we replicate every tile from A and B. When applied to the matrix multiplication, this algorithm is equivalent to the block matrix multiplication implemented using the SUMMA algorithm [14]. The group-by-join is translated to the following Spark RDD code:

```
Tiled( n, m, rdd[ (k,V) | (k,(__a,__b)) ← As.cogroup(Bs) ] ),
```

where As, Bs, and V are

```
As = A.tiles.flatMap{ case ((i,j),a)
     ⇒ (0 until B.cols/N).map(k ⇒ ((gx(i,j),k),(kx(i,j),a))) }
Bs = B.tiles.flatMap{ case ((ii,jj),b)
     ⇒ (0 until A.rows/N).map(k ⇒ ((k,gy(ii,jj)),(ky(ii,jj),b))) }
V = array(N*N)[ (k,⊕/c)
          | (k1,_a) ← __a, (k2,_b) ← __b, k2 == k1,
            ((i,j),a) ←_ a, ((ii,jj),b) ← _b, kx(i,j) == ky(ii,jj),
            let c = h(a,b), group by k: (gx(i,j),gy(ii,jj)) ].
```

That is, the tiles in A are replicated B.cols/N times and the tiles in B are replicated A.rows/N times.

6 PERFORMANCE EVALUATION

Our system, SAC, has been implemented using Scala's compile-time reflection and macros. Our code generator uses the Scala typechecker to infer the types of the generator domains to select the appropriate sparsifiers based on these types. It translates array comprehensions to Scala code that calls Spark RDD operations whose functional arguments use the Scala's Parallel Collections library [20] for multicore parallelism. The produced Scala code is embedded in the rest of the Scala code generated at compile-time. The source code of our system is available on GitHub at https://github.com/fegaras/array.

We have evaluated the performance of our system relative to the Spark MLlib.linalg library [5]. MLlib uses the linear algebra package Breeze, but in our experiments, instead of using a native Breeze library implementation, such as OpenBLAS, we used the pure JVM implementation of this library. Although there are many linear algebra libraries that support distributed tiled arrays, MLlib is the closest to our work since it is built on top of Spark and has a Scala API.

The platform used for these evaluations was a small cluster of 4 nodes, where each node has one Xeon E5-2680v3 at 2.5GHz, with 24 cores, 128GB RAM, and 320GB SSD. For our experiments,

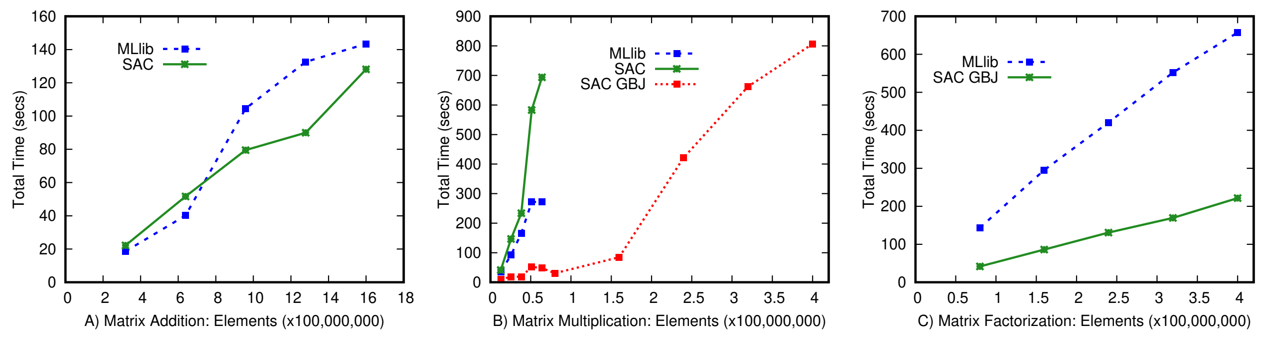


Figure 4: Performance evaluation of matrix addition, multiplication, and factorization of tiled matrices on Apache Spark

we used Apache Spark 3.0.0 running on Apache Hadoop 2.7.0. Each Spark executor was configured to have 11 cores and 60GB RAM. Consequently, there were 2 executors per node, giving a total of 8 executors. Each program was evaluated over 5 datasets and each evaluation was repeated 4 times, so each data point in the plots in Figure 4 represents the mean value of these 4 evaluations.

The matrices used in our experiments were tiled matrices where each tile had size 1000*1000. We implemented these distributed tiled matrices in MLlib.linalg as instances of BlockMatrix, where each tile was an instance of DenseMatrix. The matrices used for addition and multiplication were pairs of square matrices of the same size filled with random values between 0.0 and 10.0. The largest matrices used in matrix addition had 40000×40000 elements and size 12GB each, while those used in matrix multiplication had 20000×20000 elements and size 3GB each. Matrix multiplication was translated in two different ways in SAC: as a join followed by a group-by, and using a special group-by join (see Section 5.4 for a discussion). The results are shown in Figure 4.A and B. We can see that, for matrix addition, SAC performs a bit faster than MLlib. For matrix multiplication though, SAC (that uses a join followed by a group-by) is up to 3 times slower than MLlib, while MLib is up to 6 times slower than SAC GBJ (that uses the special group-by join).

The third program to evaluate was one iteration of matrix factorization using gradient descent [16]. The goal of this computation is to split a matrix R of dimension $n \times m$ into two low-rank matrices P and Q of dimensions $n \times k$ and $m \times k$, for small k, such that the error between the predicted and the original rating matrix $R - P \times Q^T$ is below some threshold, where $P \times Q^T$ is the matrix multiplication of P with the transpose of Q and '$-$' is cell-wise matrix subtraction. Matrix factorization can be implemented by repeatedly applying the following operations:

$$\begin{array}{rcl} E & \leftarrow & R - P \times Q^T \\ P & \leftarrow & P + \gamma(2E \times Q - \lambda P) \\ Q & \leftarrow & Q + \gamma(2E^T \times P - \lambda Q) \end{array}$$

where γ is the learning rate and λ is the normalization factor used in avoiding overfitting. For our experiments, we used $\gamma = 0.002$ and $\lambda = 0.02$. The matrix to be factorized, R, was a square sparse matrix $n{*}n$ with random integer values between 0 and 5, in which only the 10% of the elements were non-zero. The dimension k was set to 1000. The derived matrices P and Q had dimension $n * 1000$ and were initialized with random values between 0.0 and 1.0. The largest matrix R used had 20000×20000 elements and size 3GB. The results are shown in Figure 4.C. We can see that SAC (using GBJ) is up to three times faster than MLlib.

7 RELATED WORK

Many array-processing systems use special storage techniques, such as regular tiling, to achieve better performance on certain array computations. TileDB [19] 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 [22] 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 [21] stores arrays into chunks, which are typically the size of a storage block. One of their most effective storage method is a two-level chunking strategy with regular chunks and regular tiles. SciHadoop [7] 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 [15] 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 [25] 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. Linalg [26] (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 [6] 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. 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 [2, 17, 18]. The work by Luo *et al.* [18] 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 would differ from ours since it does not separate specification from implementation, thus making hard to change the array storage, and requires programmers to write explicit code to correlate tiles.

8 CONCLUSION AND FUTURE WORK

Our performance results show that SAC can be as efficient as a highly optimized array library when applied to certain distributed operations on tiled arrays. The same layered approach can also be used for translating comprehensions on other types of array storage, such as on tiled arrays where each tile is stored in the compressed sparse column format. As future work, we plan to look at operations that are hard to express using comprehensions, such as inverting a matrix, which requires a special LU decomposition algorithm. We believe that such operations should be coded as black-box library functions in a high-performance array library, such as BLAS or LAPACK. Such operations would require special optimizations to fuse them with general array comprehensions and with each other. Furthermore, our framework cannot directly generate calls to a high-performance array library, such as BLAS or LAPACK, or to GPU libraries, such as CUDA or OpenGL, but it can be improved to recognize certain patterns in a comprehension that are translatable to such calls. Finally, we would like to investigate general methods to optimize storage, such as unboxing arrays where vectors of tuples are mapped to tuples of vectors, and to parallelize irregular structures using nested parallelism.

Acknowledgments: Our evaluations were performed at the XSEDE Comet 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] 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.

[3] Apache Flink. http://flink.apache.org/, 2020.

[4] Apache Spark. http://spark.apache.org/, 2020.

[5] Apache Spark MLlib. https://spark.apache.org/mllib/, 2020.

[6] 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.

[7] 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.

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

[9] 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.

[10] L. Fegaras. An Algebra for Distributed Big Data Analytics. *JFP*, special issue on Programming Languages for Big Data, volume 27, 2017.

[11] 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.

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

[13] 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.

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

[15] 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.

[16] Y. Koren, R. Bell, and C. Volinsky. Matrix Factorization Techniques for Recommender Systems *IEEE Computer*, 42(8):30–37, August 2009.

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] P. Wadler. List Comprehensions. Chapter 7 in *The Implementation of Functional Programming Languages*, by S. Peyton Jones, Prentice Hall, 1987.

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

[25] 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.

[26] 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.

[27] 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.