Translation of Array-Based Loops to Spark SQL

Md Hasanuzzaman Noor
University of Texas at Arlington, CSE
Arlington, Texas, USA
mdhasanuzzaman.noor@mavs.uta.edu

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

***Abstract*—Many programs written to analyze data are expressed in terms of array operations in an imperative programming language with loops. However, for data analysts who need to analyze vast volumes of data, large-scale data-intensive processing is becoming a necessity. Hence, they want to convert their programs, originally written to run on a single computer, to work on current Big Data systems, such as Map-Reduce and Spark, so that they can process larger amounts of data. We present a novel framework, called SQLgen, that automatically translates imperative programs with loops and array operations to distributed data-parallel programs. Unlike related work, SQLgen translates these programs to SQL, which can be translated to more efficient code since it can be optimized using a relational database optimizer. SQLgen has been implemented on Spark SQL. We compare the performance of SQLgen with DIABLO, hand-written RDD based, and Spark SQL programs on real-world problems. SQLgen is up to 78× faster than DIABLO and up to 25× faster than hand-written RDD-based programs, giving performance close to that of hand-written programs in Spark SQL.**

***Index Terms*—Big Data, arrays, SQL, Spark, machine learning**

I. Introduction

Many organizations are shifting towards data-driven decision making where the key step is performing statistical analysis of data. During the statistical analysis, the data analysts investigate the data to discover patterns, spot anomalies, and test hypotheses. For example, they run different clustering and dimensionality reduction algorithms to gain insight into the datasets. These algorithms are usually expressed in terms of array operations because much of the data used in data analysis, scientific computing, and machine learning come in the form of arrays, such as vectors, matrices, and tensors. They operate on these array data using loops, which are inherently sequential since they access and update the array elements incrementally, one at a time. Furthermore, many of these algorithms exhibit better performance when expressed with mutable arrays, compared to other immutable data structures. More importantly, scientists and data analysts are mostly familiar with imperative programming languages, such as C and Python, and they often use numerical analysis tools that are based on arrays, such as MATLAB and NumPy, and use algorithms from linear algebra and data analysis textbooks that are expressed using loops and arrays.

Scientific organizations, such as NASA and CERN, generate and process massive amounts of data to make interesting discoveries and solve complex research problems. Big Data provides scientists with greater statistical and predictive power for data analysis. Moreover, many companies across many industries are also collecting massive amounts of user data to make business decisions through user behavior analytics using machine learning (ML) tools, such as Deep Neural Networks (DNN), that give more accurate results with large amounts of data. The most popular machine learning frameworks today, TensorFlow [1] and PyTorch [2], utilize specialized hardware, such as GPUs, TPUs, and SIMD accelerators, to parallelize algorithms and accelerate computations. These frameworks utilize the resources better when these resources are scaled up to a single high-end computer, rather than scaled out to multiple commodity computers. Recent research work has tried to close the gap of resource utilization when resources are either scaled up or scaled out (see, for example, Horovod, BigDL [3], and TensorFlowOnSpark). There has also been some recent work on combining linear algebra with the relational algebra of relational database systems [4]–[6] to let programmers write ML algorithms on conventional database systems. There are also new frameworks, such as Map-Reduce [7], Spark [8], and Flink [9], commonly known as Data-Intensive Scalable Computing Systems (DISC), that are designed for processing data on a larger scale and utilize resources better than current ML frameworks when these resources are scaled out to computer clusters. These DISC systems are distributed data-parallel systems on clusters of shared-nothing computers connected through a high-speed network. Apache Spark [8] improves the Hadoop performance by maintaining intermediate results in the memory of the compute nodes, instead of writing to the disk. Spark is also more expressive by supporting more operations in the Spark API, such as flatMap, filter, and join. These operations allow programmers to build rich pipelines of computation to do complex mathematical data processing in a concise way.

Since analyzing large amounts of data play an important role in data analysis and in the accuracy of machine learning models, many organizations want to convert their programs, originally written to run on a single computer, to work on current DISC systems so that they can process larger amounts of data. Data analysts and scientists, who are mostly familiar with imperative programming languages and numerical analysis tools that are based on arrays, have to learn new programming paradigms to rewrite their programs to run on DISC systems. This rewriting process slows down the development and deployment process and is non-trivial

since the programmers need to address the intricacies and avoid the pitfalls inherent to these frameworks to get optimal performance. The lack of expertise in a particular framework may result in erroneous or suboptimal programs. Furthermore, to achieve flexibility and better performance, instead of using libraries, such as MLlib [10], programmers may often write ad-hoc array-based programs that are specific to their needs. Consequently, instead of rewriting these ad-hoc programs by hand to run on a particular platform, one solution can be to use an automatic translation system that will translate sequential programs with loops to distributed data-parallel programs.

Because of the prevalence of array-based loop programs and the rise of Big Data, there have been significant efforts to automatically parallelize loops with array operations in the area of High Performance Computing (HPC). The key challenge here is to address loop carried dependencies, also known as *recurrences*. A recurrence occurs when there is a dependency between the iterations of a loop. For example, the update $V[i] := V[i-1] + V[i+1]$ on array V inside a loop over i is a recurrence since the values of V read in one iteration of the loop depend on the updated values of V in the previous iterations. Researchers in HPC proposed various solutions to handle these loops. Among them, the most notable techniques are distributed loops, DOACROSS, and DOPIPE. In most parallelization frameworks, the loops without recurrences are simply those that are "embarrassingly parallel" (DOALL). Even though there have been significant efforts to parallelize the loop-based programs in HPC, there has not been much work to automatically parallelize loops on DISC systems, with the notable exceptions of DIABLO, MOLD, and CASPER. CASPER [11] translates sequential java programs to Hadoop programs while MOLD [12] translates sequential java programs to Spark programs expressed in the Core API. DIABLO [13] translates loop-based programs to comprehensions and then to distributed data-parallel programs in the Spark Core API. Both MOLD and DIABLO translate the loops to RDD operations based on the Spark Core API. A Resilient Distributed Dataset (RDD) is an immutable distributed collection of elements of data, partitioned across a cluster of nodes. Even though RDDs can be operated on unstructured data in parallel using transformations and actions, working with RDDs has some performance pitfalls. One such pitfall is that the RDD operations, such as map and flatMap, take functions as arguments that are compiled to bytecode and are not optimized. Spark has addressed these shortcomings by providing two additional APIs, called DataFrames and Datasets [14].

Our goal is to design a framework that will translate array-based loops to a declarative domain-specific language (DSL), more specifically, Spark SQL [14]. At first, loops are translated to equivalent monoid comprehensions [13] and then to Spark SQL. Not all loops can be translated to SQL. We provide simple rules for dependence analysis that detect loops that cannot be translated to SQL. One such case is when an array is read and updated in the same loop. For example, we reject the update $V[i] := V[i-1] + V[i+1]$ inside a loop over i because V is read and updated in the same loop. But, unlike most related work, we can translate incremental updates of the form $V[e_1] += e_2$, for some commutative operation $+$ and some terms e_1 and e_2. We chose Spark SQL as our target language since it is in general more efficient than the Spark core API because it takes advantage of existing extensive work on SQL optimization for relational database systems. In Spark SQL, datasets are expressed as DataFrames, which are distributed collection of data, organized into named columns. The schema of a DataFrame must be known, while DataFrame computations are done on columns of named and typed values. Operations from the Spark Core API, on the other hand, are higher-order with arguments that are functions coded in the host language and compiled to bytecode, which cannot be analyzed during program optimization. Hence, Spark SQL can find and apply optimizations, such as pushing a selection before a join, that is very hard to detect automatically when the same program is written in the Spark Core API since the filter predicate is in bytecode. Spark DataFrames have two specialized back-end components, Catalyst (the query optimizer) and Tungsten (the off-heap serializer), which facilitate optimized performance on other Spark components, such as MLlib, that are based on DataFrames and Spark SQL. Catalyst supports both rule-based and cost-based optimization. For example, it can optimize a query by reordering the operations, such as pushing a filter operation before a join operation. The operations in Spark SQL reduce the amount of data sent over the network by selecting only the relevant columns and partitions from the dataset necessary for the computation. Consequently, we expect that loops translated to Spark SQL, as in our framework, would perform better than loops translated to Spark RDD operations, as it was done by earlier frameworks.

Consider, for example, a product R of two square matrices M and N such that $R_{ij} = \sum_k M_{ik} * N_{kj}$. In a loop-based language, it can be expressed as:

```
for i = 0, d-1 do
    for j = 0, d-1 do {
        R[i,j] := 0;
        for k = 0, d-1 do
            R[i,j] += M[i,k]*N[k,j]
}
```

Our framework translates the previous loop-based program to a bulk assignment to matrix R that calculates all the values of R in one shot using a bag comprehension that returns new content of R. Here, the cumulative effects of all the updates to the matrix R throughout the iterations are performed in bulk by grouping the values across the iterations by the matrix indices and then by summing up these values for each group. Then the matrix R can be replaced with these new values. Hence, the above program is translated to a comprehension as follows:

$$R := \{\, (i,j,+/v) \mid (i,k,m) \leftarrow M,\ (k',j,n) \leftarrow N,\ k = k',\ \textbf{let}\ v = m*n,\ \textbf{group by}\ (i,j) \,\}.$$

Here, the comprehension retrieves the values $M_{ik} \in M$ and $N_{kj} \in N$ as triples (i,k,m) and (k',j,n) so that $k = k'$, and sets $v = m * n = M_{ik} * N_{kj}$. After we group the values by the matrix indexes i and j, the variable v is lifted to a bag of numerical values $M_{ik} * N_{kj}$, for all k. Hence, the aggregation $+/v$ will sum up all the values in the bag v, deriving $\sum_k M_{ik} * N_{kj}$ for the ij element of the resulting matrix. This comprehension is then translated to the following Spark SQL program, where matrices are represented as tables M, N, and R with schema (I, J, V):

```
select M.I, N.J, sum(M.V*N.V) as V
from M join N on M.J=N.I
group by M.I, N.J
```

Here, the tables M and N are joined on the column J of matrix M and on column I of matrix N and then grouped by the column I of matrix M and column J of matrix N. Finally the sum of the product of the values for each group is calculated, giving the entries of matrix product as the final result.

Our framework, called SQLgen, has been implemented in Scala using compile-time reflection. The source language used to expressed loops with array operations is the same proof-of-concept language defined in DIABLO [13], while the target language is Spark SQL. Our framework can be easily extended to work with other imperative programming languages, such as C or Java.

The contributions of this paper are summarized as follows:

- We present a novel framework for translating array-based loops to Spark SQL that is able to handle all array programs that satisfy some simple recurrence restrictions.
- We evaluate our system on a variety of data analysis and machine learning programs and we compare its performance relative to DIABLO and to hand-written Spark programs expressed in the Spark Core API (RDDs) and in Spark SQL. Our performance results indicate that, for these programs, SQLgen outperforms the equivalent DIABLO and the hand-written Spark Core programs, giving performance close to that of hand-written programs in Spark SQL.

II. Related Work

MOLD [12] is a translator of Java code to Scala code that can be executed either on a single computing node via parallel Scala collections or on a cluster of computers in a distributed manner using Spark. Like SQLgen and DIABLO, it uses a group-by operation to parallelize loops with recurrences. That is, the cumulative effects of these recurrences are brought together by grouping the new array values by their destination location. For incremental updates on arrays, the source values of these updates are grouped by the array index of the incremented array and are aggregated in parallel. However, the authors use a template-based rewriting system to match specific templates of Java loops. They use a heuristic search to find the matching templates for each program fragment and to generate the Map-Reduce output program. Another translator is CASPER [11], which translates sequential Java code into semantically equivalent Map-Reduce programs. It uses a program synthesizer to search over the space of sequential program summaries expressed as IRs. Unlike MOLD, CASPER uses a theorem prover based on Hoare logic to prove that the derived Map-Reduce programs are equivalent to the original sequential programs. Our system differs from both MOLD and CASPER as it translates loops directly to parallel programs using simple meaning preserving transformations, without having to search for rules to apply. The actual rule-based optimization of our translations is done at a second stage using a small set of rewrite rules, thus separating meaning-preserving translation from optimization.

DIABLO [13] translates array-based loops to parallel programs using simple meaning preserving transformations. Unlike MOLD and CASPER, it does not use any search mechanisms, which makes the translation process fast. The transformation stage is separated from the optimization stage and optimization is done using a small set of rewrite rules. However, DIABLO lacks a comprehensive cost-based query optimizer. Our work improves DIABLO by replacing its back-end engine with Spark SQL which utilizes optimization techniques developed by database researchers. The translated Spark SQL can be faster than the programs translated by DIABLO when the schema information of the input dataset is available. Spark SQL can also be faster than a hand-optimized RDD based Scala program because of the effectiveness of Catalyst and Tungsten in Spark.

Another area related to automated parallelization for DISC systems is deriving SQL queries from imperative code in a non-distributed setting [15]. Unlike our work, this work addresses aggregates, inserts, and appends to lists but does not address array updates. The work by Luo *et al.* [6] adds a new attribute type to relational schemas to capture arrays that can fit in memory and extends SQL with array operators. Although these SQL queries are evaluated in Map-Reduce, the arrays are not fully distributed. Instead, large matrices must be split into multiple rows as indexed tiles and the programmer must use SQL to implement a matrix operation by correlating these tiles. This makes it hard to specify some matrix operations, such as matrix inversion.

III. Background

In our earlier work, we have presented a novel framework, called DIABLO (a Data-Intensive Array-Based Loop Optimizer) [13], for translating array-based loops to monoid comprehensions, which in turn are translated to Spark programs expressed in the Spark Core API. It 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.

Sparse arrays in DIABLO are represented as distributed collections of tuples that contain the indexes and value of a single array element. For example, a sparse matrix M is represented as a bag of tuples (i,j,v) such that $v = M_{ij}$.

When two arrays are used together in the body of a loop, such as in $A[i]*B[i+1]$, this term is translated to a join between A and B so that the index of A is equal to the index of B plus 1. Incremental updates, on the other hand, are translated to group-bys. For example, the cumulative effects of an update $A[e]+=v$ throughout a loop are done in bulk by grouping the values v across all loop iterations by the array index e (that is, by the different destination locations) and by summing up these values for each group. Then the entire vector A is replaced with these sums.

Loop-based programs are translated to monoid comprehensions, which have the following syntax:

Expression:

e	::=	$\ldots$	scala expression
	$\mid$	$\{\, e \mid q_1,\ \ldots,\ q_n \,\}$	comprehension
	$\mid$	$\oplus/e$	reduction

Qualifier:

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

Pattern:

p	::=	$v \mid (p_1, \ldots, p_n)$	scala pattern

In the comprehension, the expression e is the comprehension head and q_i is a qualifier. The $\oplus/e$ syntax is a shorthand for reduce$(\oplus, e)$ which is a total aggregation over a comprehension e. The domain e of a generator $p \leftarrow e$ must be a bag. This generator draws elements from this bag and, each time, it binds the pattern p to an element. A condition qualifier e is an expression of type boolean. It is used for filtering out elements drawn by the generators. A let-binding **let** $p = e$ binds the pattern p to the result of e. A group-by qualifier uses a pattern p and an optional expression e. If e is missing, it is taken to be p. The group-by operation groups all the pattern variables in the same comprehension that are defined before the group-by (except the variables in p) by the value of e (the group-by key), so that all variable bindings that result to the same key value are grouped together. After the group-by, p is bound to a group-by key and each one of these pattern variables is lifted to a bag of values. The result of a comprehension $\{\, e \mid q_1,\ \ldots,\ q_n \,\}$ is a bag that contains all values of e derived from the variable bindings in the qualifiers. Comprehensions are translated to algebraic operations that resemble the bulk operations supported by many DISC systems, such as groupBy, join, map, and flatMap. These operations are then translated to calls to the underlying DISC platform, such as calls to the Spark core API.

For example, the product R of two square matrices M and N, such that $R_{ij} = \sum_k M_{ik} * N_{kj}$, can be expressed as follows in a loop-based language:

```
for i = 0, d-1 do
    for j = 0, d-1 do {
        R[i,j] := 0;
        for k = 0, d-1 do
            R[i,j] += M[i,k]*N[k,j]   }
```

This program is translated to a single assignment that replaces the entire content of the matrix R with a new content, which is calculated using DISC operations. More specifically, it is translated to the following assignment:

$$R \;:=\; \{\, (i,j,+/v) \mid (i,k,m) \leftarrow M,\ (k',j,n) \leftarrow N,\ k = k',\ \textbf{let}\ v = m*n,\ \textbf{group by}\ (i,j) \,\}.$$

This comprehension retrieves the values $M_{ik} \in M$ and $N_{kj} \in N$ as triples (i,k,m) and (k',j,n) so that $k = k'$, and sets $v = m*n = M_{ik} * N_{kj}$. After we group the values by the matrix indexes i and j, the variable v is lifted to a bag of numerical values $M_{ik} * N_{kj}$, for all k. Hence, the aggregation $+/v$ will sum up all the values in the bag v, deriving $\sum_k M_{ik} * N_{kj}$ for the ij element of the resulting matrix. This comprehension is translated to a join between M and N followed by a reduceByKey operation in Spark.

IV. Our Framework

As we discussed in Section III, in our earlier work, loop-based programs are first translated to monoid comprehensions, then to the monoid algebra, and finally to Java byte code that calls the Spark Core API. The goal of this paper is to improve the performance of these translations by translating the generated monoid comprehensions directly to Spark SQL queries, thus taking advantage of the Catalyst optimizer used by Spark SQL, which is more powerful than the DIABLO optimizer used for optimizing the monoid algebra.

Recall from Section III that the syntax of a monoid comprehension is as follows:

e	::=	$\{\, e \mid q_1,\ \ldots,\ q_n \,\}$	comprehension
	$\mid$	$\oplus/e$	reduction

where in the comprehension, the comprehension head e is an expression and q_i is a qualifier, and $\oplus/e$ is a total aggregation over a comprehension e. The output of our translations is a list of statements c that have the following syntax: $c ::= v := s \mid \{c_1; \ldots; c_n\}$, where s is the Spark SQL generated from a comprehension, which is assigned to a variable v. Multiple assignments can be grouped in a code block c.

In DIABLO, a sparse array, such as a sparse vector or a matrix, is represented by a key-value map (also known as an indexed set), which is a bag of type $\{(K,T)\}$, where K is the array index type and T is the array value type. An array can be traversed using a generator in a comprehension. For example, we can traverse the elements of a sparse vector V using the generator $(i,v) \leftarrow V$, where the pattern variable i is the index of type Long, and v is the value. Similarly, we can traverse a sparse matrix M using a generator $((i,j),v) \leftarrow M$, where i and j are row and column indices of type Long and v is the value.

Vectors and matrices in DIABLO are translated to DataFrames in Spark SQL. Basically, a sparse array is translated to a relational table with two columns: the first column is a tuple that contains the index elements and the second column is the element value, which can be a primitive type or

a composite type, such as StructType, which is represented by a case class in Scala. For example, a vector V of type $\{(\text{Long}, \text{Double})\}$ in DIABLO is mapped to a table V of schema $(\_1 : \text{Long}, \_2 : \text{Double})$, while a matrix M of type $\{((\text{Long}, \text{Long}), \text{Double})\}$ in DIABLO is mapped to a table M of schema $(\_1 : \text{Struct}\,(\_1 : \text{Long}, \_2 : \text{Long}), \_2 : \text{Double})$, where the index column is nested with the row index column referred to as _1._1 and the column index referred to as _1._2.

A. Program Translation

We translate the comprehension in two steps: pattern compilation and comprehension translation. Pattern variables in a comprehension are defined in the generators and used in the rest of the comprehension. However, SQL does not support patterns. To address this problem, we eliminate patterns by substituting each pattern with a fresh variable and by creating an environment ρ that binds the variables in the pattern to terms that depend on the fresh variable. The fresh variable is also used as the alias for the SQL table. For example, if there is a generator $((i,j),v) \leftarrow e$ in a comprehension, we replace it with $x \leftarrow e$, where x is a fresh variable, and we create an environment $\rho = [i \to x.\_1.\_1,\ j \to x.\_1.\_2,\ v \to x.\_2]$, which expresses i, j, and v in terms of x. (The term $x.\_n$ returns the nth element of the tuple x.) Given a term x and a pattern p, the semantic function $\mathcal{C}[\![p]\!]_x$ returns a binding list that binds the pattern variables in p in terms of x such that $p = x$:

$$\mathcal{C}[\![(p_1,\ldots,p_n)]\!]_x = \mathcal{C}[\![p_1]\!]_{x.\_1} +\!\!+ \cdots +\!\!+ \mathcal{C}[\![p_n]\!]_{x.\_n} \quad (1)$$

$$\mathcal{C}[\![v]\!]_x = [v \to x] \quad (2)$$

where $+\!\!+$ merges bindings. For our example, after applying (1) on $((i,j),v)$ we get the bindings:

$$\begin{aligned}\mathcal{C}[\![((i,j),v)]\!]_x &= \mathcal{C}[\![(i,j)]\!]_{x.\_1} +\!\!+ \mathcal{C}[\![v]\!]_{x.\_2} \\ &= \mathcal{C}[\![i]\!]_{x.\_1.\_1} +\!\!+ \mathcal{C}[\![j]\!]_{x.\_1.\_2} +\!\!+ \mathcal{C}[\![v]\!]_{x.\_2}\end{aligned}$$

Then, applying (2) we get, $\mathcal{C}[\![((i,j),v)]\!]_x = [i \to x.\_1.\_1,\ j \to x.\_1.\_2,\ v \to x.\_2]$. Before the translation to SQL, we eliminate the patterns from a comprehension as follows. For each generator $p \leftarrow e'$, and any sequences of qualifiers $\overline{q_1}$ and $\overline{q_2}$ in a comprehension, we do:

$$\{\, e \mid \overline{q_1},\ p \leftarrow e', \overline{q_2} \,\} = \{\, \rho(e) \mid \overline{q_1},\ x \leftarrow e', \rho(\overline{q_2}) \,\} \quad (3)$$

where x is a fresh variable and $\rho = \mathcal{C}[\![p]\!]_x$, which expresses the variables in p in terms of the fresh variable x. The $\rho(e)$ and $\rho(\overline{q_2})$ replace all occurrences of the variables in e and $\overline{q_2}$ using the binding ρ. For example, the comprehension

$$\{\, (i, a+b) \mid (i,a) \leftarrow A, (j,b) \leftarrow B, i = j \,\}$$

is translated to:

$$\{\, (x.\_1, x.\_2 + y.\_2) \mid x \leftarrow A,\ y \leftarrow B,\ x.\_1 = y.\_1 \,\}$$

The next step is the translation of a comprehension to SQL using the semantic function $\mathcal{SQL}$, which takes the comprehension as input and translates it to a Spark SQL query:

$$\begin{aligned}\mathcal{SQL}[\![\{\, h \mid \overline{q} \,\}]\!] = {} & \textbf{select}\, h \ \ \textbf{from}\, \mathcal{Q}[\![\overline{q}]\!] \\ & \textbf{where}\, \mathcal{P}[\![\overline{q}]\!] \ \ \textbf{group by}\, \mathcal{G}[\![\overline{q}]\!]\end{aligned} \quad (4)$$

where h refers to comprehension head and the semantic functions $\mathcal{Q}$, $\mathcal{P}$, and $\mathcal{G}$ translate a list of qualifiers to SQL tables and joins, predicates, and group-by expression respectively. They are described next in this section.

The comprehension head h is translated to a *select* clause. For a total aggregation over a comprehension, such as $\oplus/\{\, h \mid \ldots \,\}$, the monoid $\oplus$ is applied to the translation of the header h in the *select* clause. For example, the header of $+/(\{\, v \mid (i,v) \leftarrow V \,\})$ is translated to $\textbf{select}\, sum(v)$.

We use the semantic function $\mathcal{Q}$ to translate the generators in a comprehension to SQL *from* clauses. If there are pairs of generators in the qualifiers correlated with a join condition, we translate each such pair to a *join* clause along with a join condition. In the following rules, semantic function $\mathcal{Q}$ takes a list of qualifiers as input, identifies joins, and creates a SQL join clause with a join condition:

$$\begin{aligned}&\mathcal{Q}[\![\overline{q_1},\ v_1 \leftarrow e_1,\ \overline{q_2},\ v_2 \leftarrow e_2,\ \overline{q_3},\ e_3 = e_4,\ \overline{q_4}]\!] = \\ &\quad \mathcal{Q}[\![\overline{q_1},\ (v_1,v_2) \leftarrow (e_1\, \textbf{join}\, e_2\, \textbf{on}\, e_3 = e_4),\ \overline{q_2},\ \overline{q_3},\ \overline{q_4}]\!]\end{aligned} \quad (5)$$

where $e_3 = e_4$ must correlate the variables v_1 and v_2, that is, e_3 must depend on v_1 only and e_4 must depend on v_2 only, or vice versa. The remaining generators are translated to table traversals:

$$\mathcal{Q}[\![v \leftarrow e,\ \overline{q}]\!] = e\, v,\ \mathcal{Q}[\![\overline{q}]\!] \quad (6)$$

$$\mathcal{Q}[\![e,\ \overline{q}]\!] = \mathcal{Q}[\![\overline{q}]\!] \quad (7)$$

$$\mathcal{Q}[\![\ ]\!] = \emptyset \quad (8)$$

where (6) collects the generators that are not joined with any other table as simple table traversals. If there is more than one such table, this corresponds to a cross product, which is not supported by Spark SQL.

The semantic function $\mathcal{P}$ is used to collect condition qualifiers. It takes a list of qualifiers and translates them to a list of SQL conditions:

$$\mathcal{P}[\![e,\ \overline{q}]\!] = e\, \textbf{and}\, \mathcal{P}[\![\overline{q}]\!] \quad (9)$$

$$\mathcal{P}[\![p \leftarrow e,\ \overline{q}]\!] = \mathcal{P}[\![\overline{q}]\!] \quad (10)$$

$$\mathcal{P}[\![\ ]\!] = \emptyset \quad (11)$$

The semantic function $\mathcal{G}$ collects the group-by keys. Currently, our translation algorithm accepts at most one group-by qualifier. $\mathcal{G}$ takes a list of qualifiers as input and returns an optional group-by key:

$$\mathcal{G}[\![\textbf{group by}\, p,\ \overline{q}]\!] = p \quad (12)$$

$$\mathcal{G}[\![p \leftarrow e,\ \overline{q}]\!] = \mathcal{G}[\![\overline{q}]\!] \quad (13)$$

$$\mathcal{G}[\![e,\ \overline{q}]\!] = \mathcal{G}[\![\overline{q}]\!] \quad (14)$$

$$\mathcal{G}[\![\ ]\!] = \emptyset \quad (15)$$

B. Examples of Program Translation

Consider a loop-based program that sums up the values of an array, written as follows:

$$sum := 0$$
$$\textbf{for } i = 1, 10 \textbf{ do } sum \mathrel{+}= V[i]$$

Here, the values of an array V are summed and assigned to a variable sum. The comprehension of this program is:

$$sum := +/(\{\, v \mid (i, v) \leftarrow V, \text{inRange}(i, 1, 10) \,\})$$

where the predicate $\text{inRange}(x.\_1, 1, 10)$ returns true if $1 \leq x.\_1 \leq 10$. Before the translation to SQL, we eliminate the patterns from the comprehension. The only pattern in the comprehension is (i, v), which is replaced with a fresh variable x. Then, using (1) and (2) we get $\mathcal{C}[\![(i, v)]\!]_x = [i \rightarrow x.\_1, v \rightarrow x.\_2]$. Therefore, using (3), the comprehension is transformed to:

$$sum := +/(\{\, x.\_2 \mid x \leftarrow V, \text{inRange}(x.\_1, 1, 10) \,\})$$

To generate the equivalent SQL query, we use (4) to translate the transformed comprehension to SQL where the semantic functions take the qualifiers of the transformed comprehension as their input.

$$\begin{array}{rll} & \textbf{select} & sum(x.\_2) \\ & \textbf{from} & \mathcal{Q}[\![x \leftarrow V, \text{inRange}(i, 1, 10)]\!] \\ & \textbf{where} & \mathcal{P}[\![x \leftarrow V, \text{inRange}(i, 1, 10)]\!] \\ & \textbf{group by} & \mathcal{G}[\![x \leftarrow V, \text{inRange}(i, 1, 10)]\!] \\ = & \textbf{select} & sum(x.\_2) \\ & \textbf{from} & V\, x \\ & \textbf{where} & 1 <= x.\_1 \textbf{ and } x.\_1 <= 10 \end{array}$$

As another example, consider the matrix multiplication between the matrices M and N, which is stored in the matrix R:

```
for i=0, 10 do
    for j=0, 10 do {
        R[i,j] := 0.0;
        for k=0, 10 do
            R[i,j] += M[i,k]*N[k,j];
    };
```

The comprehension of matrix multiplication is as follows:

$$R := \{\, ((i, j), (+/v)) \mid ((i, k), m) \leftarrow M, ((k', j), n) \leftarrow N, k = k', \textbf{let } v = m * n, \textbf{group by } (i, j) \,\}$$

To keep this example simple, we omit the inRange qualifiers. In the comprehension above, the patterns $((i, k), m)$ and $((k', j), n)$ are replaced with fresh variables x and y. Then, after applying (1) and (2), we get the following bindings: $\mathcal{C}[\![((i, k), m)]\!]_x = [i \rightarrow x.\_1.\_1, k \rightarrow x.\_1.\_2, m \rightarrow x.\_2]$, $\mathcal{C}[\![((k', j), n)]\!]_y = [k' \rightarrow y.\_1.\_1, j \rightarrow y.\_1.\_2, n \rightarrow y.\_2]$. Then, these patterns are eliminated using (3) and the comprehension is transformed to:

$$R := \{\, ((x.\_1.\_1, y.\_1.\_2), (+/v)) \mid (x \leftarrow M, y \leftarrow N, x.\_1.\_2 = y.\_1.\_1, \textbf{let } v = x.\_2 * y.\_2, \textbf{group by } (x.\_1.\_1, y.\_1.\_2) \,\}$$

Then, we can get the equivalent SQL from (4). In the $select$ clause, we get, $x.\_1.\_1, y.\_1.\_2, sum(x.\_2 * y.\_2)$ where v is substituted by the let-binding expression. Next, the semantic function $\mathcal{Q}$ is applied to the transformed comprehension in the $from$ clause. Using (5-8), we get:

$$\begin{array}{rll} & \textbf{select} & x.\_1.\_1,\, y.\_1.\_2,\, sum(x.\_2 * y.\_2) \\ & \textbf{from} & \mathcal{Q}[\![x \leftarrow M, y \leftarrow N, x.\_1.\_2 = y.\_1.\_1, \\ & & \textbf{group by } x.\_1.\_1,\, y.\_1.\_2]\!] \\ & \textbf{group by} & \mathcal{G}[\![q]\!] \\ = & \textbf{select} & x.\_1.\_1,\, y.\_1.\_2,\, sum(x.\_2 * y.\_2) \\ & \textbf{from} & M\, x \textbf{ join } N\, y \textbf{ on } x.\_1.\_2 = y.\_1.\_1 \\ & \textbf{group by} & \mathcal{G}[\![q]\!] \end{array}$$

Next, we apply the semantic function $\mathcal{P}$ to the transformed comprehension in the $where$ clause, which is not shown here. Then, we apply semantic function $\mathcal{G}$ to the transformed comprehension in the $group\,by$ clause. Using (12), we get:

$$\begin{array}{rll} & \textbf{select} & x.\_1.\_1,\, y.\_1.\_2,\, sum(x.\_2 * y.\_2) \\ & \textbf{from} & M\, x \textbf{ join } N\, y \textbf{ on } x.\_1.\_2 = y.\_1.\_1 \\ & \textbf{group by} & \mathcal{G}[\![x \leftarrow M, y \leftarrow N, \\ & & x.\_1.\_2 = y.\_1.\_1, \\ & & \textbf{group by } x.\_1.\_1,\, y.\_1.\_2]\!] \\ = & \textbf{select} & x.\_1.\_1,\, y.\_1.\_2,\, sum(x.\_2 * y.\_2) \\ & \textbf{from} & M\, x \textbf{ join } N\, y \textbf{ on } x.\_1.\_2 = y.\_1.\_1 \\ & \textbf{group by} & x.\_1.\_1,\, y.\_1.\_2 \end{array}$$

The final translation is an assignment that assigns the result of the generated SQL query to table R.

V. Performance Evaluation

Our translation system SQLgen is implemented on top of DIABLO [13]. At first, array-based loops are translated to monoid comprehensions, and then the monoid comprehensions are translated to Spark SQL by SQLgen.

We evaluated the performance of SQLgen on 12 different programs and compared it with equivalent DIABLO programs, hand-written RDD-based Spark programs, and Spark SQL programs. The platform used in our experiments is the XSEDE Comet cloud computing infrastructure at SDSC (San Diego Supercomputer Center) [16]. Each program was run on a cluster of 10 nodes where each node is equipped with 24 core Xeon E5-2680v3 processor with 2.5GHz clock speed, 128GB RAM and 320GB SSD. The programs were run on Apache Spark 2.2.0 on Apache Hadoop 2.6.0. Each Spark executor on Spark was configured to have 4 cores and 23 GB RAM. So, there were 24/4 = 6 executors per node, giving a total of 60 executors, from which 2 were reserved. The input data for each program were randomly generated. Each program was evaluated 4 times on each of 5 different sizes of datasets. From the 4 iterations over each dataset, the results from the first iteration were ignored to avoid the possible overhead due to the JIT warm-up time. So, each data point in the plots represents the mean time on the rest of the 3 iterations. The input dataset size was calculated by multiplying the length of the dataset by the size of each serialized dataset element. For example, the size of a serialized RDD of the key-value pair

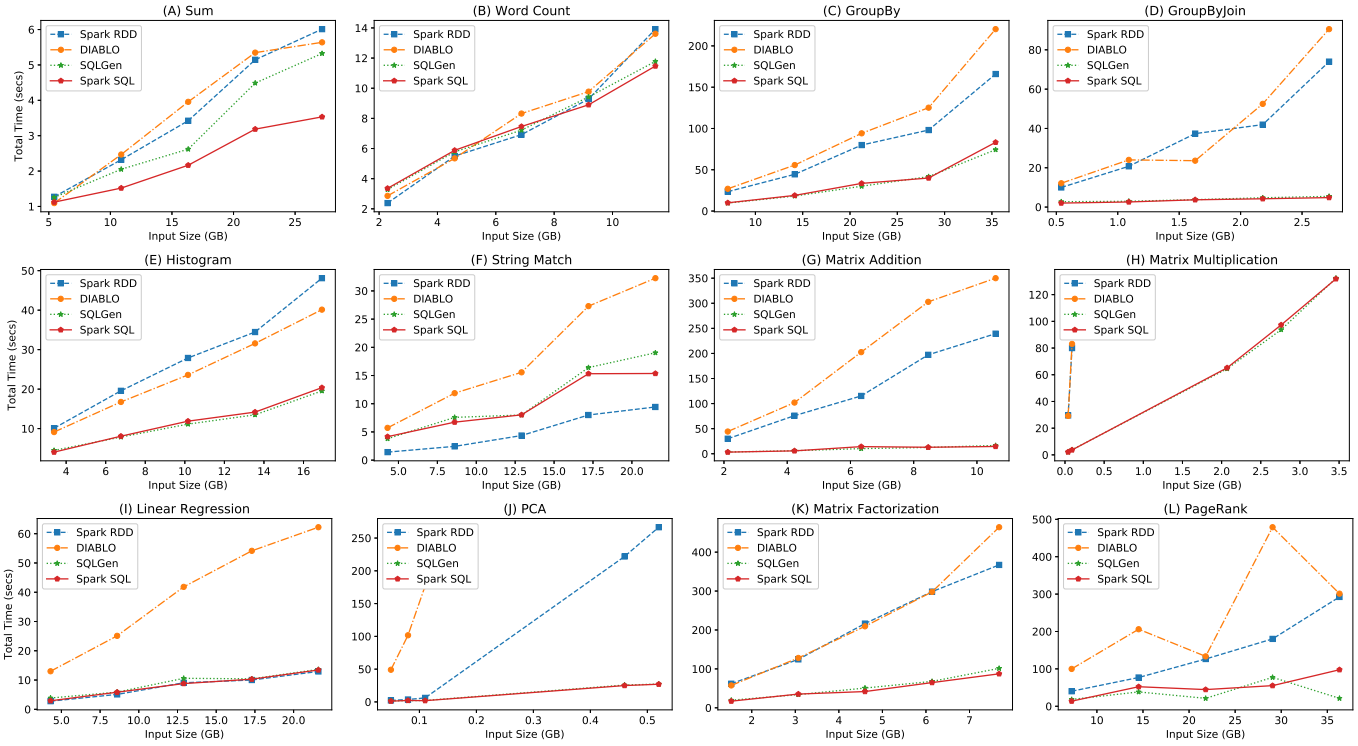

Fig. 1. Performance of SQLgen relative to DIABLO, hand-written Spark RDD code, and Spark SQL code

RDD[(Long, Double)] is 47 bytes. So, the size of 100 key-value pairs is 47*100 = 4700 bytes. The performance results are shown in Fig. 1.

Sum (A) and *Word Count (B):* Sum aggregates a dataset that contains random data. The largest dataset used had 2×10^8 elements and size 27.19 GB. Word Count counts the number of occurrences of strings with 4 characters in a dataset with 1000 different strings. The largest dataset used had 8×10^7 elements and size 11.47 GB. For these two experiments, all four modes of evaluation had similar performance.

GroupBy (C) and *GroupByJoin (D):* GroupBy groups a dataset by its first component and sums up the second component. The first components were random long integers with 10 duplicates on average. The largest dataset used had 2×10^8 elements and size 35.39 GB. The speedup range of SQLgen was $2.8\times - 3.1\times$ with an average speedup of $3\times$ compared to DIABLO. GroupByJoin joins two datasets, groups the result by some component, and returns the sum of another component in each group. The join keys of both datasets were random long integers with 10 duplicates on average. The largest datasets had 2×10^7 elements and size 2.72 GB each. For this experiment, the speedup range of SQLgen was $4.4\times - 16.7\times$ with an average speedup of $9.3\times$ compared to DIABLO.

Histogram (E): Histogram calculates the frequency of values in a dataset containing RGB values (0-255). The largest dataset used had 9×10^7 elements and size 16.93 GB. For this experiment, The speedup range of SQLgen was $2.04\times - 2.34\times$ with an average speedup of $2.1\times$ compared to DIABLO.

String Match (F): String Match matches a list of keys with a file containing strings and counts the number of occurrences of the keys in the file. The largest file containing the strings had 15×10^7 strings and size 21.51 GB and keys were broadcast to the worker nodes. In this experiment, the hand-written RDD based program was the fastest and the speedup range of SQLgen was $1.5\times - 1.9\times$ with an average speedup of $1.7\times$ compared to DIABLO.

Matrix Addition (G) and Matrix Multiplication(H): The matrices used for addition and multiplication were pairs of square matrices of the same size. Although sparse, all matrix elements were provided, were placed in random order, and were filled with random values between 0.0 and 10.0. The largest matrices used in matrix addition had 7000×7000 elements and size 10.59 GB each, while those in multiplication had 4000×4000 elements and size 3.46 GB each. For matrix addition, the speedup range of SQLgen was $12.8\times - 24\times$ with an average speedup of $18.9\times$ compared to DIABLO. Multiplication on DIABLO and the hand-written RDD-based program was very slow, so it was only run on 2 datasets and the speedup of SQLgen was $14.6\times$ and $21.75\times$ respectively compared to DIABLO. For the rest of the datasets, SQLgen has similar performance to the hand-written Spark SQL program.

Linear Regression (I): Linear Regression takes a dataset of 2-D points and calculates the intercept and the slope coefficient that models the dataset. The data used were points $(x + dx, x - dx)$, where x is a random double between 0

and 1000, and dx is a random double between 0 and 10. The largest dataset used had 12×10^7 elements and size 21.57 GB. For this experiment, the speedup range of SQLgen was $3.4\times-5.2\times$ with an average speedup of $4.3\times$ compared to DIABLO and has similar performance to the hand-written RDD-based program and the Spark SQL program.

PCA (J): Given a set of data points in the form of a matrix, PCA calculates the mean vector and the covariance matrix. The largest dataset had 6000×400 elements and size 0.52 GB. PCA on DIABLO was very slow, so it was only run on 3 datasets with 30, 40, and 50 columns and the speedup range of SQLgen was $27.7\times-78.9\times$ with an average speedup of $53.8\times$ compared to DIABLO. For the next two datasets, where the number of columns was increased to 400, SQLgen has an average speedup of $9.2\times$ compared to the hand-written RDD-based program.

Matrix Factorization (K): This program is one iteration of matrix factorization using gradient descent. For our experiments, we used the learning rate $a = 0.002$ and the normalization factor $b = 0.02$. The matrix to be factorized, R, was a square sparse matrix $n * n$ with random integer values between 1 and 5, in which only 10% of the elements were provided (the rest were implicitly zero). The derived matrices P and Q had dimensions $n * 2$ and $2 * n$, respectively, and were initialized with random values between 0.0 and 1.0. The largest matrix R used had 6000×6000 elements and size 7.68 GB. For this experiment, the speedup range of SQLgen was $2.9\times-4.6\times$ with an average speedup of $4\times$ compared to DIABLO.

PageRank (L): This program computes one iteration of the PageRank algorithm that assigns a rank to each vertex of a graph, which measures its importance relative to the other vertices in the graph. The graphs used in our experiments were synthetic data generated by the RMAT (Recursive MATrix) Graph Generator using the Kronecker graph generator parameters a=0.30, b=0.25, c=0.20, and d=0.25. The number of edges generated was 10 times the number of graph vertices. The largest graph used had 2×10^7 vertices, 2×10^8 edges, and had size 36.32 GB. For this experiment,the speedup range of SQLgen was $5.4\times-14.2\times$ with an average speedup of $7.5\times$ compared to DIABLO.

From all these experiments, we can see that the programs generated by SQLgen have similar performance to the hand-written Spark SQL programs. For many graphs shown in Fig. 1, the SQLgen lines coincide with that of the hand-written Spark SQL lines, which implies that the derived SQL queries from SQLgen are equivalent (although not equal) to the hand-written SQL queries. On the other hand, compared to hand-written RDD-based programs and the programs generated by DIABLO, SQLgen is significantly faster except for the simple programs Sum and Word Count, where the performance of all four programs was similar.

VI. Conclusion and Future Work

We have presented a framework that translates array-based loops to Spark SQL. In this framework, array-based loops are translated to comprehensions, which are then translated to Spark SQL. One of the most effective representations of a large dense array in a distributed setting is a blocked array, such as a tiled matrix, where an array block is the unit of data distribution. As a future work, we are planning to translate array-based loops to SQL over blocked arrays.

Acknowledgment

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562.

References

[1] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[3] J. J. Dai, Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, C. L. Zhang, Y. Wan, Z. Li *et al.*, "Bigdl: A distributed deep learning framework for big data," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 50–60.

[4] L. Chen, A. Kumar, J. Naughton, and J. M. Patel, "Towards linear algebra over normalized data," *Proc. VLDB Endow.*, vol. 10, no. 11, p. 1214–1225, Aug. 2017.

[5] A. Kunft, A. Alexandrov, A. Katsifodimos, and V. Markl, "Bridging the gap: towards optimization across linear and relational algebra," in *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, 2016, pp. 1–4.

[6] 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*, vol. 31, no. 7, pp. 1224–1238, 2018.

[7] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[8] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[9] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[10] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[11] M. B. S. Ahmad and A. Cheung, "Automatically leveraging mapreduce frameworks for data-intensive applications," in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 1205–1220.

[12] C. Radoi, S. J. Fink, R. Rabbah, and M. Sridharan, "Translating imperative code to mapreduce," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 909–927, 2014.

[13] L. Fegaras and M. H. Noor, "Translation of array-based loops to distributed data-parallel programs," *PVLDB*, vol. 13, no. 8, pp. 1248–1260, 2020.

[14] Armbrust *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. ACM, 2015, pp. 1383–1394.

[15] K. V. Emani, K. Ramachandra, S. Bhattacharya, and S. Sudarshan, "Extracting equivalent sql from imperative code in database applications," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1781–1796.

[16] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson *et al.*, "Xsede: accelerating scientific discovery," *Computing in science & engineering*, vol. 16, no. 5, pp. 62–74, 2014.