Research online Publishing
www.ronpub.com

Open Access

Open Journal of Big Data (OJBD)
Volume 6, Issue 1, 2022
http://www.ronpub.com/ojbd
ISSN 2365-029X

Translation of Array-based Loop Programs to Optimized SQL-based Distributed Programs

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

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

Abstract

Many data analysis programs are often expressed in terms of array operations in sequential loops. However, these programs do not scale very well to large amounts of data that cannot fit in the memory of a single computer and they have to be rewritten to work on Big Data analysis platforms, such as Map-Reduce and Spark. We present a novel framework, called SQLgen, that automatically translates sequential loops on arrays to distributed data-parallel programs, specifically Spark SQL programs. We further extend this framework by introducing OSQLgen, which automatically parallelizes array-based loop programs to distributed data-parallel programs on block arrays. At first, our framework translates the sequential loops on arrays to monoid comprehensions and then to Spark SQL. For SQLgen, the SQL is over coordinate arrays while for OSQLgen, it is over block arrays. As block arrays are more compact than coordinate arrays, computations on block matrices are significantly faster than on arrays in the coordinate format. Since not all array-based loops can be translated to SQL on block arrays, we focus on certain patterns of loops that match an algebraic structure known as a semiring. Many linear algebra operations, such as matrix multiplication required in many machine learning algorithms, as well as many graph programs that are equivalent to a semiring can be translated to distributed data-parallel programs on block arrays using OSQLgen, thus giving us a substantial performance gain. Finally, to evaluate our framework, we compare the performance of OSQLgen with GraphX, GraphFrames, MLlib, and hand-written Spark SQL programs on coordinate and block arrays on various real-world problems.

Type of Paper and Keywords

Regular research paper: *Big Data, arrays, SQL, Spark, graph processing, machine learning*

1 Introduction

Many industrial and scientific organizations are generating and processing large amounts of data for business and research purposes. These organizations are rapidly shifting towards data-driven decision-making and statistical data analysis involving pattern discovery, anomaly detection, and hypothesis testing, by running various clustering and dimensionality reduction algorithms to gain insights into datasets. The data used in these data analysis algorithms come in different forms of arrays, such as vectors, matrices, and tensors. Programmers often write data analysis programs that 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 operating on 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 implement algorithms from linear algebra and data analysis textbooks that are expressed using loops and arrays.

The massive amount of data generated by scientific organizations, such as NASA and CERN, are used to solve complex research problems and make important scientific discoveries, which are increasingly data-driven. Moreover, many companies are also collecting massive data to make business decisions using machine learning (ML) tools, such as Deep Neural Networks (DNN). It has been shown that these ML algorithms give more accurate results when they work on larger amounts of training data. The most popular machine learning frameworks today, TensorFlow [18] and PyTorch [36], utilize specialized hardware, such as GPUs, TPUs, and SIMD accelerators, to parallelize algorithms and accelerate computations. These frameworks utilize the computing resources better when these resources are scaled up to a single high-end computer, rather than scaled out to multiple commodity computers. Although both Tensorflow and PyTorch implement some deep learning algorithms using distributed evaluation, neither of these systems provide a distributed linear algebra library to implement customized machine learning algorithms. There are recent systems though, such as, Horovod [40], BigDL [10], and TensorFlowOnSpark [48], that implement deep learning algorithms using scalable distributed algorithms. Furthermore, there are systems that implement linear algebra operations in a relational database system using SQL or relational algebra [8, 24, 28] to let programmers write ML algorithms on conventional database systems. There are also frameworks, such as Map-Reduce [13], Spark [49], and Flink [7], commonly known as Data-Intensive Scalable Computing (DISC) Systems, that are designed for processing data on a larger scale. These systems utilize resources better than current ML frameworks when these resources are scaled out to computer clusters. Apache Spark [49] improves the Hadoop performance by maintaining intermediate results in the memory of the compute nodes instead of storing them on disk. Spark is also more expressive by supporting more operations in the Spark API, such as filter and join. These operations allow programmers to build rich pipelines of computations to do complex mathematical data processing in a concise way.

As the need to process and analyze large amounts of data on DISC systems is increasing, many organizations now want to convert their sequential programs into distributed programs. Therefore, the scientists and developers who are mostly familiar with array-based imperative programming languages and tools have to learn a new programming paradigm and 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 [32], programmers may often want to 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 DISC platform, one solution can be to use an automatic translation system that will translate sequential programs with loops to distributed data-parallel programs.

Our goal is to design a framework that will translate array-based loops to a declarative domain-specific language (DSL), more specifically, Spark SQL [3], to evaluate these programs in a DISC platform to analyze Big Data. At first, loops are translated to equivalent monoid comprehensions [19] and then to Spark SQL. Not all loops can 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 $V[e_1] \mathrel{+}= e_2$, for some commutative operation $+$ and some terms e_1 and e_2. This is accomplished by collecting all the e_2 values across the loops, grouping these values by e_1, aggregating each group using $+$, and replacing V with the result. That is, the effects of evaluating this incremental update multiple times across loops are captured by a single SQL query that calculates V in one shot. 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. 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 DataFrames can find and apply optimizations that are very hard to detect automatically when the same program is written in the Spark Core API. Consequently, sequential loops perform better when they are translated to Spark SQL instead of Spark RDD operations.

Our first framework, called SQLgen, translates array-based sequential loops to Spark SQL on arrays stored in

the coordinate storage format (as distributed collections of index-value pairs). Although straightforward, this storage format is space inefficient and adds a communication overhead during data shuffle. Our second framework, called OSQLgen, uses an efficient compact array storage format, namely a block matrix, which is a distributed collection of non-overlapping dense array blocks. In Spark, a block matrix is implemented as a distributed collection of fixed-sized dense square tiles. This storage format reduces not only the required storage but also the communication cost of a distributed algorithm, since the amount of data that needs to be transferred over the network is smaller. Thus, OSQLgen, which translates sequential loops to Spark SQL on block arrays, is orders of magnitude faster than SQLgen and other related approaches that are based on the coordinate format. Unfortunately, generating SQL code on block arrays from an arbitrary monoid comprehension is very hard in general because some operations may require multiple blocks from an array to construct a single block of the result. For example, matrix rotation in which every row is rotated, requires two input blocks for each output block. To alleviate the complexity of this problem, we focus on certain patterns of monoid comprehensions that can be translated to Spark SQL on block arrays that look similar to SQL on coordinate arrays, except that the arithmetic operations in the query must now work on array blocks instead of values. These patterns are based on an algebraic structure called a *semiring*, which has already been used successfully to capture general graph algorithms. Consequently, OSQLgen not only can translate array operations that match a semiring, but it is also very suitable for translating general graph algorithms expressed in loops to SQL on block arrays.

Our frameworks, SQLgen and OSQLgen, have been implemented on Spark DataFrames and the code is written in Scala using compile-time reflection. The source language used to express sequential loops with array operations is the same proof-of-concept language used in DIABLO [19], 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, and generate code for other DISC systems that support SQL, such as Apache Flink [7].

The contributions of this paper are summarized as follows:

- We present a novel framework, called SQLgen, for translating array-based loops to Spark SQL that is able to handle all array programs that satisfy some simple recurrence restrictions.
- We evaluate SQLgen 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.
- We extend SQLgen to handle block arrays, in a novel framework, called OSQLgen, for translating array-based loop programs to optimized Spark SQL programs on block arrays that is able to handle many important programs including many graph algorithms that satisfy the properties of a semiring.
- We compare the performance of OSQLgen on real-world problems relative to GraphX, GraphFrames, MLlib, and hand-written Spark SQL programs on coordinate and block arrays.

2 Related Work

With the rise of Big Data and the prevalence of array-based loop programs, there have been significant efforts in the area of High-Performance Computing (HPC) to automatically parallelize loops with array operations. The key challenge in achieving this is the existence of 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. In most parallelization frameworks, the loops without recurrences are simply those that are "embarrassingly parallel" (DOALL). DOALL [23] translates sequential loops when there is no recurrence. To handle some recurrences, DOACROSS [9] rewrites a loop with a recurrence to separate the dependent and independent parts of an iteration. The independent part is run in parallel and merged with the dependent part which is run sequentially. DOPIPE [15], distributes an iteration with a loop-carried dependency over multiple synchronized loops where each step starts when there is sufficient data available from the previous step.

Even though there have been significant efforts to parallelize the loop-based programs in HPC, there has been very little work to automatically parallelize loops in DISC systems, with the notable exceptions of MOLD, CASPER, and DIABLO. MOLD [37] is a translator of sequential Java code to parallel/distributed Scala code that can be executed either on a single computer using parallel Scala collections or on a cluster of computers using Spark. Like SQLgen and DIABLO, it uses group-by operations to parallelize certain loops with recurrences. However, MOLD uses a template-based rewriting system to match specific templates of Java loops. It uses a heuristic search to find the matching

templates for each program fragment and to generate the output Spark program. CASPER [2] 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, and 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 requiring to search for rules to apply. DIABLO [19] 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. Furthermore, both MOLD and DIABLO translate sequential 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. Unfortunately, working with RDDs has some performance pitfalls. One such pitfall is that RDD operations, such as map and flatMap, take functions as arguments that are compiled to bytecode, which cannot be optimized at run-time. Spark has addressed these shortcomings by providing two additional APIs, called DataFrames and Datasets [3]. Our work improves DIABLO by replacing its back-end engine with Spark SQL which utilizes database query optimization techniques. The Spark SQL generated by SQLgen is often faster than the Spark RDD programs generated by DIABLO because Spark SQL is optimized by a relational-style cost-based query optimizer, called Catalyst, that can generate very efficient physical plans that are evaluated in a very efficient execution engine, called Tungsten, that substantially improves memory management and pushes performance closer to the limits of modern hardware. Furthermore, all these systems, including SQLgen, generate Spark code that works on arrays in coordinate format, that is, as distributed collections of index-value pairs, while OSQLgen generates Spark code for block arrays, which are distributed collections of dense array blocks. Block arrays have shown to be far more compact and faster to process in a distributed environment than arrays in coordinate format.

Many array-processing systems use special storage techniques, such as array tiling, to achieve better performance on certain array computations. TileDB [35] 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. SciDB [41] 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. Similar to SciDB, ArrayStore [42] stores arrays into chunks, which are typically the size of a storage block. One of their most effective storage methods is a two-level chunking strategy with regular chunks and regular tiles. SciHive [20] is a scalable array-based query system that enables scientists to process raw array datasets in parallel with a SQL-like query language. SciHive maps array datasets in NetCDF files to Hive tables and executes queries via Map-Reduce. Based on the mapping of array variables to Hive tables, SQL-like queries on arrays are translated to HiveQL queries on tables and then optimized by the Hive query optimizer. SciMATE [47] extends the Map-Reduce API to support the processing of the NetCDF and HDF5 scientific formats, in addition to flat-files. TensorFlow [18] is a dataflow language for machine learning that supports data parallelism on multi-core machines and GPUs but has limited support for distributed computing for certain ML algorithms. Linalg [6] (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 [4] 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.

There has also been some recent work on combining linear algebra with relational algebra to let programmers implement ML algorithms on relational database systems [8, 24]. The work by Luo *et al.* [28] 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. This makes it hard to specify some matrix operations, such as matrix inversion. The PArADISE [30, 31] framework evaluates a set of ML algorithms and linear algebra operations using SQL in a parallel relational database system. It transforms ML algorithms, such as the Hidden Markov Model written in R, to a sequence of SQL queries that can be executed in parallel in a parallel DBMS. However, the translation from R to SQL is not completely automated while the language restrictions in R make the system more complex. The system described in [17] derives SQL queries from imperative code in a non-distributed setting. Unlike our work, this work addresses aggregates, inserts, and appends to lists but does not address array updates.

Over the past few decades, there have been various efforts to unify graph algorithms. Some of these studies found that many graph algorithms are repetitive steps that have a traversal pattern similar to matrix multiplication and can be represented using an algebraic structure called a semiring [1, 25, 44]. Based on this representation, researchers have proposed general optimized implementations for various graph algorithms. For example, Takahashi *et al.* [43] have implemented graph algorithms using block matrix multiplication implemented in a parallel system. More recently, researchers in parallel systems have standardized an interface for graph algorithms that are based on semirings under a project called GraphBLAS [12]. The goal of this project is to isolate users from low-level implementation details for optimal parallelization. Similarly, Ekanadham *et al.* [16] proposed a similar interface, called GPI, for graph algorithms that are based on semirings on Spark. Nevertheless, the vertex-centric approach, first introduced in Pregel [29], remains the most popular framework for graph processing on DISC systems and includes systems, such as Spark GraphX [21] and GraphFrames [11]. Our system OSQLgen [34] differs from both GraphBLAS and GPI as it works on graph algorithms that are expressed as sequential loop programs, thus making it unnecessary for the programmers to learn the API introduced by related work and can still get good performance.

3 BACKGROUND

Our earlier work, called DIABLO (a Data-Intensive Array-Based Loop Optimizer) [19], translates array-based loops to monoid comprehensions, which in turn are translated to Spark programs expressed in the Spark Core API. The syntax of the loop-based language used as the source of the translations is given in Figure 1. It 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.

Sparse arrays in DIABLO are represented as distributed collections of tuples that contain the indices 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 starting from small expressions to more complex expressions which result in nested comprehensions. After that, nested comprehensions are normalized using normalization transformations. The syntax of comprehension is as follows:

$$\{\, e \mid q_1,\ \ldots,\ q_n \,\},$$

which consists of comprehension head e and a list of qualifiers. Qualifiers can be a generator, a let-binding qualifier, a condition qualifier, and a group-by qualifier. In the list of qualifiers, q_i is defined as follows:

Qualifier:

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

Pattern:

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

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

Expression:

e	::=	d	a destination (L-value)
	|	$e_1 \star e_2$	any binary operation $\star$
	|	$(e_1, \ldots, e_n)$	tuple construction
	|	$\langle\, A_1{=}e_1, \ldots, A_n{=}e_n \,\rangle$	record construction
	|	$const$	constant (int, float, ...)

Destination:

d	::=	v	variable
	|	$d.A$	record projection
	|	$v[e_1, \ldots, e_n]$	array indexing

Statement:

s	::=	$d \mathrel{+}= e$	incremental update
	|	$d := e$	assignment
	|	$\mathbf{var}\ v : t = e$	declaration
	|	$\mathbf{for}\ v = e_1, e_2\ \mathbf{do}\ s$	iteration
	|	$\mathbf{for}\ v\ \mathbf{in}\ e\ \mathbf{do}\ s$	traversal
	|	$\mathbf{while}\ (e)\ s$	loop
	|	$\mathbf{if}\ (e)\ s_1\ [\,\mathbf{else}\ s_2\,]$	conditional
	|	$\{\, s_1;\ \ldots;\ s_n \}$	statement block

Figure 1: Syntax of loop-based programs

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 C of two square matrices A and B such that $C_{ij} = \sum_k A_{ik} * B_{kj}$ can be expressed as follows in our loop-based language:

$$
\begin{array}{l}
\mathbf{for}\ i = 0, d-1\ \mathbf{do} \\
\quad \mathbf{for}\ j = 0, d-1\ \mathbf{do}\ \{ \\
\quad\quad C[i,j] = 0.0; \\
\quad\quad \mathbf{for}\ k = 0, d-1\ \mathbf{do}\ \{ \\
\quad\quad\quad C[i,j]\ \mathrel{+}= A[i,k] * B[k,j]); \\
\quad\quad \} \\
\quad \}
\end{array}
$$

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

$$
\begin{array}{lll}
C & := & \{\, ((i,j), +/v) \mid ((i,k), m) \leftarrow A, \\
& & \quad ((k',j), n) \leftarrow B,\ k = k', \\
& & \quad \mathbf{let}\ v = m * n, \\
& & \quad \mathbf{group\ by}\ (i,j)\,\}.
\end{array}
$$

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

The DIABLO recurrence restrictions are more permissive than those imposed by many current systems and are statically checked at compile-time. For a loop to be parallelizable, many systems require that an array should not be both read and updated in the same loop. But they also reject incremental updates, such as $V[i] \mathrel{+}= 1$, because such an update reads from and writes to the same vector V. DIABLO relaxes these restrictions by accepting incremental updates as long as there are no other recurrences present. However, the destination of a non-incremental update must be a different location at each loop iteration, which is in general impossible to assert at compile time. Instead, if the update destination is an array indexing, DIABLO requires that the array indices be affine and completely cover all surrounding loop indices. An affine expression takes the form $c_0 + c_1 * i_1 + \cdots + c_k * i_k$, where $i_1, \ldots, i_k$ are loop indices and $c_0, \ldots, c_k$ are constants. This restriction does not hold for incremental updates, which allow arbitrary array indices in a destination as long as the array is not read in the same loop.

4 THE SQLGEN FRAMEWORK

In our earlier work, DIABLO translates loop-based programs to monoid comprehensions, which in turn are translated to the monoid algebra, are optimized, and then translated to Java byte code that calls the Spark Core

API. The goal of SQLgen 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.

Consider, for example, the product C of two square matrices $A_{n\times n}$ and $B_{n\times n}$ such that $C_{ij} = \sum_k A_{ik} * B_{kj}$, given in Section 3. As we have seen in Section 3, this operation is calculated by the following comprehension:

$$\begin{array}{ll}\{\,((i,j),+/v) \mid & ((i,k),m) \leftarrow A,\\ & ((k',j),n) \leftarrow B,\ k = k',\\ & \textbf{let}\ v = m * n,\\ & \textbf{group by}\ (i,j)\,\}.\end{array}$$

SQLgen translates this comprehension to the following Spark SQL query, where matrices are represented as tables A, B, and C with schema $((\_1, \_2), \_2)$:

$$\begin{array}{ll}\textbf{select} & struct(A.\_1.\_1,\ B.\_1.\_2)\ \textbf{as}\ \_1,\\ & sum(A.\_2 * B.\_2)\ \textbf{as}\ \_2\\ \textbf{from} & A\ \textbf{join}\ B\ \textbf{on}\ A.\_1.\_2 = B.\_1.\_1\\ \textbf{group by} & A.\_1.\_1,\ B.\_1.\_2.\end{array}$$

Here, the tables A and B are joined on the column $\_1.\_2$ of matrix A and on column $\_1.\_1$ of matrix B and then grouped by the column $\_1.\_1$ of matrix A and column $\_1.\_2$ of matrix B. Finally, the sum of the product of the values for each group is calculated, giving the entries of matrix product as the final result.

The output of our translations is a sequence of statements c that has the following syntax:

Target Code:

$$\begin{array}{lcll} c & ::= & v := s & \text{assignment}\\ & \mid & \{c_1; \ldots; c_n\} & \text{code block}\end{array}$$

where s is a Spark SQL query 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. SQLgen translates these vectors and matrices 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. 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 $\_1.\_2$.

Although Spark SQL supports very powerful syntax and features, SQLgen uses only a small fraction of the language, which makes it easy to switch to a different SQL-based system. The syntax of Spark SQL generated by our translator is as follows:

$$\begin{array}{ll}\textbf{select} & expression\,[\textbf{as}\,alias]\\ & [,\ expression\,[\textbf{as}\,alias],\ ...]\\ \textbf{from} & (relation\,[alias]\mid\\ & relation\,[alias]\,\textbf{join}\,relation\,[alias]\\ & \textbf{on}\,boolean\_expression\\ & [\textbf{join}\,relation\,[alias]\\ & \textbf{on}\,boolean\_expression\,...])\\ [\textbf{where} & boolean\_expression\\ & [\textbf{and}\,boolean\_expression\,\textbf{and}\,...]]\\ [\textbf{group by} & expression\,[,expression,...]]\end{array}$$

where alternatives are shown in parenthesis $(\ldots|\ldots|\ldots)$ and optional parts in square brackets $[\ldots]$. Expressions in the $select$ clause contain column names and may contain an aggregate function. In the $from$ clause, a relation can be a table or a view. To simplify our translation, we assume that the input programs will only have for-loops but no while-loops. We have also restricted our generated SQL queries to use inner joins and no subqueries.

4.1 Translation to SQL

We translate comprehensions in two steps: pattern compilation and comprehension translation. During the pattern compilation step, we remove the pattern variables from the comprehensions. Then in the comprehension translation step, we translate the transformed comprehensions to Spark SQL programs.

4.1.1 Pattern Compilation

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 $\rho = [i \rightarrow x.\_1.\_1,\ j \rightarrow x.\_1.\_2,\ v \rightarrow x.\_2]$, 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 $p = x$:

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

$$\mathcal{C}[\![v]\!]_x = [v \rightarrow x] \tag{2}$$

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 Rule (2), we get

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

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 the following transformation:

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

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:

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

4.1.2 Comprehension Translation

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:

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

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. 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:

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

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:

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

$$\mathcal{Q}[\![e,\ \overline{q}]\!] = \mathcal{Q}[\![\overline{q}]\!] \tag{7}$$

$$\mathcal{Q}[\![\ ]\!] = \emptyset \tag{8}$$

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:

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

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

$$\mathcal{P}[\![\ ]\!] = \emptyset \tag{11}$$

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:

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

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

$$\mathcal{G}[\![e,\ \overline{q}]\!] = \mathcal{G}[\![\overline{q}]\!] \tag{14}$$

$$\mathcal{G}[\![\ ]\!] = \emptyset \tag{15}$$

4.2 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 $\text{inRange}(i, 1, 10)$ returns true if $1 \leq i \leq 10$. 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 Rules (1) and (2) we get $\mathcal{C}[\![(i, v)]\!]_x = [i \rightarrow x.\_1, v \rightarrow x.\_2]$. Therefore, using Rule (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.

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

Consider now the following loop-based program:

$$\textbf{for } i = 1, 10 \textbf{ do } W[K[i]] \mathrel{+}= V[i];$$

For each different key k in K, this program sums the values V_i associated with the same key $K_i = k$ and stores the results in array W. The comprehension of the above program is:

$$\begin{array}{rll} W & := & \{\, (a, +/v) \mid (i, v) \leftarrow V, \text{inRange}(i, 1, 10), \\ & & \qquad (m, a) \leftarrow K, i = m, \\ & & \qquad \textbf{group by } a \,\} \end{array}$$

In this comprehension, there are two generators V and K containing the patterns (i, v) and (m, a). After replacing the patterns with fresh variables x and y and applying Rules (1) and (2), we get: $\mathcal{C}[\![(i, v)]\!]_x = [i \rightarrow x.\_1, v \rightarrow x.\_2]$ and $\mathcal{C}[\![(m, a)]\!]_y = [m \rightarrow y.\_1, a \rightarrow y.\_2]$. Then the patterns are removed from the comprehension using Rule (3):

$$\begin{array}{rll} W & := & \{\, (y.\_2, +/x.\_2) \mid x \leftarrow V, \\ & & \qquad \text{inRange}(x.\_1, 1, 10), \\ & & \qquad y \leftarrow K, x.\_1 = y.\_1, \\ & & \qquad \textbf{group by } y.\_2 \,\} \end{array}$$

Then, we apply (4) where the header of the comprehension $(y.\_2, +/x.\_2)$ is translated to $y.\_2,\ sum(x.\_2)$. Then, using Rules (5)-(8), the semantic function $\mathcal{Q}$ is applied to the transformed comprehension in the $from$ clause:

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

Using Rules (9)-(11), we get:

$$\begin{array}{rll} & \textbf{select} & y.\_2,\ sum(x.\_2) \\ & \textbf{from} & V\ x \textbf{ join } K\ y \textbf{ on } x.\_1 = y.\_1 \\ & \textbf{where} & \mathcal{P}[\![x \leftarrow V, \text{inRange}(x.\_1, 1, 10), \\ & & \quad y \leftarrow K,\ x.\_1 = y.\_1, \\ & & \quad \textbf{group by } y.\_2]\!] \\ & \textbf{group by} & \mathcal{G}[\![q]\!] \\ = & \textbf{select} & y.\_2,\ sum(x.\_2) \\ & \textbf{from} & V\ x \textbf{ join } K\ y \textbf{ on } x.\_1 = y.\_1 \\ & \textbf{where} & 1 \leq x.\_1 \textbf{ and } x.\_1 \leq 10 \\ & \textbf{group by} & \mathcal{G}[\![q]\!] \end{array}$$

Then, using Rule (12), we get:

$$\begin{array}{rll} & \textbf{select} & y.\_2,\ sum(x.\_2) \\ & \textbf{from} & V\ x \textbf{ join } K\ y \textbf{ on } x.\_1 = y.\_1 \\ & \textbf{where} & 1 \leq x.\_1 \textbf{ and } 10 \leq x.\_1 \\ & \textbf{group by} & \mathcal{G}[\![x \leftarrow V, \text{inRange}(x.\_1, 1, 10), \\ & & \quad y \leftarrow K, x.\_1 = y.\_1, \\ & & \quad \textbf{group by } y.\_2]\!] \\ = & \textbf{select} & y.\_2,\ sum(x.\_2) \\ & \textbf{from} & V\ x \textbf{ join } K\ y \textbf{ on } x.\_1 = y.\_1 \\ & \textbf{where} & 1 \leq x.\_1 \textbf{ and } x.\_1 \leq 10 \\ & \textbf{group by} & y.\_2 \end{array}$$

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

As yet another example, consider the matrix multiplication between the matrices M and N, which is

stored in the matrix R:

$$
\begin{array}{l}
\textbf{for } i = 0, 10 \textbf{ do} \\
\quad \textbf{for } j = 0, 10 \textbf{ do } \{ \\
\qquad R[i,j] = 0.0; \\
\qquad \textbf{for } k = 0, 10 \textbf{ do } \{ \\
\qquad\quad R[i,j] \ += M[i,k] * N[k,j]); \\
\qquad \} \\
\quad \}
\end{array}
$$

It is translated to the following comprehension:

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

To keep this example simple, we omit the $\mathrm{inRange}$ 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 Rules (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 Rule (3) and the comprehension is transformed to:

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

Then, we can get the equivalent SQL using Rule (4). In the $select$ clause, we get, $struct(x.\_1.\_1, y.\_1.\_2), sum(x.\_2 * y.\_2)$ 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 Rules (5-8), we get:

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

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 Rule (12), we get:

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

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

5 THE OSQLGEN FRAMEWORK

In SQLgen, matrices and vectors are represented using a coordinate format which accompanies each nonzero element with its row and column indices. Although straightforward, this storage format is space inefficient and adds a communication overhead during data shuffle. Instead, one can use an efficient compact array storage format, such as a block matrix, which is a distributed collection of non-overlapping dense array blocks. In Spark, a block matrix can be implemented as an RDD of fixed-sized dense square tiles of type $\mathrm{RDD[((Int, Int), Array[Double])]}$, where each block $\mathrm{((i, j), A)}$ has block coordinates i and j and values stored in the dense matrix A, which has a fixed size $N * N$, for some constant N. This storage format is more compact and it reduces the communication cost of a distributed algorithm since the amount of data that needs to be transferred over the network is smaller.

Consider the matrix multiplication algorithm again, given in Section 3 where the coordinate arrays A, and B are represented as tables with schema $\mathrm{A((i, k), m)}$, and $\mathrm{B((k', j), n)}$. If the matrices A and B are of size $n \times n$, then during the join, each row of the input matrices is copied n times which is also known as replication rate (r). Instead, we can store the matrices as block matrices with schema $Ab((I,K),M)$ and $Bb((K',J),N)$, where I, K, K' and J are block coordinates and M and N are array blocks. Then, the replication rate r to compute the product $Cb_{IJ} = \sum_K A_{IK} * B_{KJ}$ is $\sqrt{n}$ times less than the coordinate approach [38]. Furthermore, the multiplication between the array blocks can be pushed down to CPU/GPU using efficient Basic Linear Algebra Subprograms (BLAS) routines [5]. Hence, the block matrix approach is superior to the coordinate approach in terms of space, communication, and computation time. Unfortunately, generating SQL code on block arrays from an arbitrary comprehension is very hard in general

Table 1: Semirings for graph algorithms

Semirings	Applications
$+(A * B)$	All-pairs shortest path
$+(A * b)$	PageRank
$max(A * B)$	Maximum reliability path
$min(max(A,\ B))$	Minimum spanning tree
$max(min(A,\ B))$	Maximum capacity path

because some operations may require multiple blocks from an array to construct a single block of the result. To address this problem, we focus on certain patterns of comprehensions that can be directly translated to Spark SQL on block arrays that look similar to SQL on coordinate arrays, except that the arithmetic operations in the query must now work on array blocks instead of values.

Over the past few decades, researchers have proposed solutions to generalize graph algorithms in a form similar to the matrix multiplication algorithm. These algorithms are represented using a general algebraic structure called a *semiring*, where the $+$ and $*$ operations of matrix multiplication are replaced with an additive monoid $\oplus$ and a multiplicative monoid $\otimes$, respectively. Formally, a semiring $(S, \oplus, \otimes, \overline{0}, \overline{1})$ is an algebraic structure defined over a set S, equipped with two monoids: an additive monoid $(\oplus, \overline{0}) : S \times S \rightarrow S$ with identity $\overline{0}$ and a multiplicative monoid $(\otimes, \overline{1}) : S \times S \rightarrow S$ with identity $\overline{1}$. The additive monoid must be associative and commutative and the multiplicative monoid needs to be associative and distribute over the additive monoid. For example, in terms of a semiring, the matrix multiplication algorithm between matrices A and B can be represented as $+(A * B)$, where $\oplus$ and $\otimes$ are equal to $+$ and $*$, respectively. Similarly, the classical graph algorithm problem all-pairs-shortest-path can be represented in terms of the semiring $min(G + G)$ where G is the transition matrix of the input graph G. Some other well-known algorithms that fall under this umbrella are shown in Table **??**. On the other hand, the block matrix multiplication can also be represented in terms of the semiring $(S, +_b, *_b, \overline{0}, \overline{1})$, where the set S consists of $N \times N$ blocks and $+_b$, and $*_b$ represent addition and multiplication of blocks. We will show that, the algorithms that can be expressed in terms of semirings and are based on scalar operations can also be expressed in terms of semirings that are based on block operations. We have provided a proof of equivalency in terms of comprehensions in Appendix A. Given this equivalency, an array-based loop program that is equivalent to a semiring can be translated to a DISC program on block arrays so it can leverage the performance benefits of block implementation.

Our goal is to capitalize on this performance gain based on semiring and implement this in our framework SQLgen [33]. This can be done by translating array-based loop programs that are equivalent to semirings to programs on block arrays expressed in Spark SQL [3]. At first, loops are translated to equivalent monoid comprehensions, as in SQLgen, but instead of directly translating the comprehensions to Spark SQL programs, we check if a comprehension is equivalent to a semiring. In that case, we translate the comprehension to a Spark SQL program on block arrays. If the comprehensions are not equivalent to a semiring, we translate the programs to a Spark SQL program by following the rules of SQLgen.

Let's consider one iteration of the all-pairs shortest path algorithm on an input graph G written using arrays and loops. A graph G is represented by a transition matrix G where G_{ij} = distance between the nodes i and j with $G_{ii} = 0$. When there is no path between two nodes, the distance is initialized to $+\infty$.

$$
\begin{array}{l}
\textbf{var}\ R : \text{matrix[Double]} = \text{matrix}(); \\
\textbf{for}\ i = 0, n-1\ \textbf{do} \\
\quad \textbf{for}\ j = 0, n-1\ \textbf{do}\ \{ \\
\quad\quad \textbf{for}\ k = 0, n-1\ \textbf{do}\ \{ \\
\quad\quad\quad //update \\
\quad\quad\quad R[i,j]\ := min(R[i,j],\ G[i,k] + G[k,j]); \\
\quad\quad \} \\
\quad\quad\quad //assignment \\
\quad\quad G[i,j] := R[i,j]; \\
\quad \}
\end{array}
$$

The above program consists of two key steps: the update step and the assignment step. In the update step, for each vertex, the algorithm finds the minimum distance path among other vertices, and in the assignment step, the updated graph replaces the existing graph.

SQLgen translates this all-pairs shortest program to a comprehension as follows:

$$
\begin{array}{rcl}
R & := & \{\, ((i,j), min/v) \mid ((i,k), m) \leftarrow G, \\
 & & \quad\quad ((k',j), n) \leftarrow G,\ k = k', \\
 & & \quad\quad \textbf{let}\ v = m + n, \\
 & & \quad\quad \textbf{group by}\ (i,j)\,\}
\end{array}
$$

This comprehension retrieves the values $G_{ik} \in G$ and $G_{kj} \in G$ in coordinate format as triples $((i,k), m)$ and $((k',j), n)$ so that $k = k'$, and sets $v = m + n = G_{ik} + G_{kj}$. After we group the values by the indices i and j, the variable v is lifted to a bag of numerical values $G_{ik} + G_{kj}$, for all k. Hence, the aggregation min/v will return the minimum of all the values in the bag v, deriving $min_k\{A_{ik} + B_{kj}\}$ for the ij element of the resulting array. Since this comprehension is equivalent to a semiring

comprehension, we translate this comprehension to a semiring comprehension on block arrays. At first, the array G is converted to block array Gb with nested schema $((\mathrm{I}:\mathrm{Int},\mathrm{J}:\mathrm{Int}),\mathrm{V}:\mathrm{Array[Double]})$ where I, J represents block indices and V represents a block. The scalar operations min, and $+$ are replaced with block min (min_b) and block addition $(+_b)$, respectively:

$$\begin{aligned} Rb \;:=\; & \{\,((X.\_1.\_1, Y.\_1.\_2), (min_b/V)) \mid \\ & X \leftarrow Gb, Y \leftarrow Gb, X.\_1.\_2 = Y.\_1.\_1, \\ & \mathbf{let}\, V = X.\_2 +_b Y.\_2, \\ & \mathbf{group\ by}\ (X.\_1.\_1, Y.\_1.\_2)\,\}. \end{aligned}$$

Similar to SQLgen, the input to our translation system is monoid comprehensions and the output is a list of statements c where we added while loop for iterations.

5.1 OSQLgen Storage System

An array in OSQLgen is stored in a relational table with two columns: the first column is a nested struct column of $\mathrm{StructType}$ that contains the block coordinates and the second column is a block of values. For example, a vector V of type double is stored in a table V with schema $(\_1 : \mathrm{Int},\ \_2 : \mathrm{Vector})$, where $\_1$ is the block coordinate. The Vector is in memory and can be either dense or sparse. A dense Vector is of type Array[Double] while a sparse vector is is stored as a pair of arrays of equal size: $(\mathrm{indices} : \mathrm{Array[Int]}, \mathrm{values} : \mathrm{Array[Double]})$. For example, a vector $(1.0, 0.0, 2.0)$ is stored in dense format as $[1.0, 0.0, 2.0]$ and in sparse format as $([0, 2], [1.0, 2.0])$. A matrix M, on the other hand, of type is stored a table M with schema

$$(\_1 :\ \mathrm{Struct}(\_1 :\ \mathrm{Int},\ \_2 :\ \mathrm{Int}), \_2 : \mathrm{Matrix}),$$

where column $\_1$ is the block coordinates (row and column coordinates) and $\_2$ is the block. The Matrix block can be either dense or sparse. The dense matrix is stored in the column-major format as Array[Double]. For example, the following matrix:

$$\begin{bmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \\ 5.0 & 6.0 \end{bmatrix}$$

is stored in Matrix as $[1.0, 3.0, 5.0, 2.0, 4.0, 6.0]$. The sparse matrix is represented in Compressed Sparse Column (CSC) format as a triple of arrays:

$$(\mathrm{colPtrs} : \mathrm{Array[Int]},\ \mathrm{rowIndices} :\ \mathrm{Array[Int]},\ \mathrm{values} :\ \mathrm{Array[Double]})$$

where the values array contains all the non-zero entries of the matrix in a column-major order, $\mathrm{rowIndices}$ array contains the row indices of the values in the values array, and $\mathrm{colPtrs}$ contains the pointers to the first elements of each column appended by the number of non-zero elements in the matrix. For example, the following matrix:

$$\begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 3.0 \\ 2.0 & 0.0 \end{bmatrix}$$

is stored with $\mathrm{values} = [1.0, 2.0, 3.0]$, $\mathrm{rowIndices} = [0, 2, 1]$, and $\mathrm{colPointers} = [0, 2, 3]$.

5.2 Mapping to Block Matrix

We use the following mapping to transform a coordinate matrix A to a block matrix:

$$\begin{aligned} \{\,((I,J), \mathrm{matrix}(m)) \mid\ & ((i,j),v) \leftarrow A, \\ & \mathbf{let}\, I = i/N, \mathbf{let}\, J = j/N, \\ & \mathbf{let}\, m = ((i\%N, j\%N), v), \\ & \mathbf{group\ by}\ (I,J)\,\} \end{aligned}$$

In the head of this comprehension, we have the coordinates (I, J) of the block and a call to a function $\mathrm{matrix(m)}$, which takes a list of coordinates of form $((i, j), v)$ and returns an array representing a dense matrix. This function is implemented as follows in Scala:

```
def matrix(L : List[(i, j), v)]) : Array[T] ={
  val V = Array.ofDim[T](N ∗ N);
  for {((i, j), v) ← L}
    V(i + N ∗ j) := v;
  V;
}
```

For a vector B, we use the following mapping to transform a coordinate vector to a block vector:

$$\begin{aligned} \{\,(I, \mathrm{vector}(m)) \mid\ & (i,v) \leftarrow B, \mathbf{let}\, I = i/N, \\ & \mathbf{let}\, m = (i\%N, v), \\ & \mathbf{group\ by}\ I\,\}, \end{aligned}$$

where the function $\mathrm{vector(m)}$ takes a list of index-value of form (i, v) and returns an array representing a dense vector. This function is defined as follows:

```
def vector(L : List[(i, v)]) : Array[T] ={
  val V = Array.ofDim[T](N);
  for {(i, v) ← L}
    V(i ∗ N) := v;
  V;
}
```

5.3 Translation of Semiring Operations to SQL

In our framework, the array-based graph programs that match a semiring structure are translated to Spark SQL programs on block arrays. These graph programs

generally consist of two steps: update, and assignment, which are repeated until a stopping condition is met. In the update step, a new graph is constructed from an existing graph, and in the assignment step, the updated graph replaces the existing graph. These two steps generate two comprehensions inside a code block in our framework. We provide a new semantic function $\mathcal{S}$ to pattern-match a semiring (the update step) in the input comprehension and transform the input comprehensions to comprehensions on block arrays. Next, we apply pattern compilation to remove the patterns from the comprehensions, as it was done in SQLgen.

We provide a semantic function $\mathcal{B}$ to translate the comprehension on block arrays to a Spark SQL query on block arrays. In our framework, a comprehension between two arrays A, B that is equivalent to a semiring has the following structure:

$$\{\,(g, \oplus/v) \mid (\bar{i}, m) \leftarrow A,\ (\bar{j}, n) \leftarrow B,\ \rho_1(\bar{i}) = \rho_2(\bar{j}), \mathbf{let}\ v = m \otimes n,\ \mathbf{group\ by}\ g : f(\bar{i}, \bar{j})\,\},$$

where $\oplus$ and $\otimes$ form a semiring. This comprehension retrieves the key-value pairs $(\bar{i}, m)$, and $(\bar{j}, n)$ from generators A and B, where $\bar{i}$ and $\bar{j}$ are tuples that contain indices of the generators. In the join condition $(\rho_1(\bar{i}) = \rho_2(\bar{j}))$, functions ρ_1 and ρ_2 are applied on the indices of A and B to get the columns on which the generators are joined. The let-binding qualifier sets v to the multiplicative monoid $\otimes$ applied to the values of the generators. Then, the values are grouped by the key g which depends on the indices of the generators (indicated as a function f of the indices). Finally, the comprehension head contains a key-value pair where the key is the group-by key and the value is calculated by applying additive monoid $\oplus$ on v.

For each code block, we pattern-match the derived comprehension using the semantic function $\mathcal{S}$ to check if it is equivalent to a semiring. If it does, the comprehension on coordinate arrays is transformed to comprehension on block arrays:

$$\begin{aligned}\mathcal{S}[\![\{\,(g, \oplus/v) \mid (\bar{i}, m) \leftarrow A,\ (\bar{j}, n) \leftarrow B, \rho_1(\bar{i}) = \rho_2(\bar{j}),\\ \mathbf{let}\ v = m \otimes n, \mathbf{group\ by}\ g : f(\bar{i}, \bar{j})\,\}\!] =\\ \{\,(G, \oplus_b/V) \mid (\overline{I}, M) \leftarrow Ab,\ (\overline{J}, N) \leftarrow Bb,\\ \rho_1(\overline{I}) = \rho_2(\overline{J}),\ \mathbf{let}\ V = M \otimes_b N,\\ \mathbf{group\ by}\ G : f(\overline{I}, \overline{J})\,\} \qquad (16)\end{aligned}$$

The coordinate matrices A and B of type $\{((\mathrm{Long}, \mathrm{Long}), \mathrm{Double})\}$ in Rule (16) are transformed to block matrices Ab and Bb of type $\{((\mathrm{Int}, \mathrm{Int}), \mathrm{Array}[\mathrm{Double}])\}$, where $\overline{I}$ and $\overline{J}$ represent the coordinates of each block M of Ab and N of Bb, respectively. If the generator B is a vector of type $\{(\mathrm{Long}, \mathrm{Double})\}$, it is transformed to block vector Bb of type $\{(\mathrm{Int}, \mathrm{Array}[\mathrm{Double}])\}$, where $\overline{J}$ represents the index of the block, and N represents the values of the block. The updated graph computed using rule (16) replaces the current graph using the following rule:

$$\mathcal{S}[\![\{\,h \mid x \leftarrow A\,\}]\!] = \{\,H \mid X \leftarrow Ab\,\}, \qquad (17)$$

where h and H refer to the headers of coordinate and block matrices respectively.

The coordinate arrays are then mapped to block arrays using the mappings described in Section 5.2. Then pattern compilation is applied on the transformed comprehension using a semantic function described in section 4.1.1. For example, the comprehension:

$$\{\,((I, J), \oplus_b/V) \mid ((I, K), M) \leftarrow Ab,\ ((K', J), N) \leftarrow Bb, K = K',\ \mathbf{let}\ V = M \otimes_b N,\ \mathbf{group\ by}\ (I, J)\,\}$$

is translated to:

$$\{\,((X.\_1.\_1, Y.\_1.\_2), \oplus_b/V) \mid (X \leftarrow Ab,\ Y \leftarrow Bb,\ X.\_1.\_2 = Y.\_1.\_1,\ \mathbf{let}\ V = X.\_2 \otimes_b Y.\_2,\ \mathbf{group\ by}\ (X.\_1.\_1, Y.\_1.\_2)\,\}.$$

Finally, the transformed comprehensions are translated to SQL programs on block arrays using the semantic function $\mathcal{B}$:

$$\begin{aligned}&\mathcal{B}[\![\{\,(G, \oplus_b/V) \mid ((X \leftarrow Ab,\ Y \leftarrow Bb, K = K',\\ &\qquad \mathbf{let}\ V = M \otimes_b N,\ \mathbf{group\ by}\ G\,\}]\!] =\\ &''\mathbf{select}\quad struct(G),\ tile\_sum(\\ &\qquad collect\_list(mult\_tiles(\mathrm{M}, \mathrm{N})))\\ &\mathbf{from}\quad Ab\ X\ \mathbf{join}\ Bb\ Y\ \mathbf{on}\ K = K'\\ &\mathbf{group\ by}\quad G'' \qquad (18)\end{aligned}$$

Here, the user-defined functions $tile\_sum$, and $mult\_tiles$ represent additive, and multiplicative monoids on block arrays respectively. The implementations of these user-defined functions are provided by our framework. We provide four different implementations of $mult\_tiles$: multiplication between dense matrices, multiplication between sparse matrices, multiplication between dense and sparse matrices, and multiplication between sparse and dense matrices. We also provide four different implementations of $mult\_tiles$ between a matrix and vector. For example, the multiplicative monoid between two dense matrices

$*_b$ is defined as:

```
def mult_tiles(M : Array[T], N : Array[T])
                                        :Array[T] = {
  val V = Array.ofDim[T](N * N);
  for {i ← 0 until N; j ← 0 until N}{
     V[i + N * j] := ⊕zero
     for {k ← 0 until N}
       V[i + N * j] := V[i + N * j] ⊕
            (M[i + N * k] ⊗ N[k + N * j]);
  }
  V;
}
```

User-defined function $tile\_sum$ applies additive monoid after aggregating the blocks using built-in function $collect\_list$. We provide two different implementations of $tile\_sum$ for a sequence of both dense and sparse matrices and vectors. For example, the additive monoid $+_b$ for a sequence of dense matrices is defined as:

```
def tile_add(M : List[Array[T]) : Array[T] ={
 val V = Array.ofDim[T](N * N);
 for {x ← M; i ← 0 until N; j ← 0 until N}
       V[i + N * j]  := V[i + N * j] ⊕ x(i + N * j);
 V;
}
```

We also translate basic linear algebra operations, such as matrix transpose to SQL program on block arrays. Let's consider the comprehension of the transpose of matrix M:

$$\{\,((j,i),m) \mid ((i,j),m) \leftarrow M\,\}.$$

At first, matrix M is mapped to the block matrix Mb and then is translated to following SQL program:

```
select   struct(X._1._2, X._1._1),
         tile_transpose(X._2)
from     Mb X.
```

We also provide special functions when the header of the comprehension in Rule (17) contains scalar operation on a column. For example, for the addition of a constant c to the value column we provide the following function:

```
def vec_sum(M : Array[T], c : Double)
                                    : Array[T] ={
 val V = Array.ofDim[T](N * N);
 for {i ← 0 until N}
   V(i) := V(i) + c;
 V;
}
```

5.4 Examples of Program Translation

Let's consider one iteration of all-pairs shortest path computation of an input graph G written using arrays and loops. A graph G is represented by a transition matrix where G_{ij} = distance between node i and j, $G_{ii} = 0$ and $G_{ij} = +\infty$ if there is no edge between i and j.

```
var R : matrix[Double] = matrix();
for i = 0, n − 1 do
  for j = 0, n − 1 do {
    for k = 0, n − 1 do {
      R[i, j]  := min(R[i, j], G[i, k] + G[k, j]);
    }
    G[i, j] := R[i, j];
  }
```

The derived comprehensions of this program are:

$$R := \{\,((i,j), min/v) \mid ((i,k),m) \leftarrow G, \\ ((k',j),n) \leftarrow G,\ k = k',\ \mathbf{let}\ v = m + n, \\ \mathbf{group\ by}\ (i,j)\,\} \tag{19}$$

$$G := R \tag{20}$$

We apply the semantic function $\mathcal{S}$ on them which transforms the comprehensions on coordinate arrays to comprehensions on block arrays using Rules (16) and (17):

$$Rb := \{\,((I,J), min_b/V) \mid ((I,K),M) \leftarrow Gb, \\ ((K',J),N) \leftarrow Gb,\ K = K', \\ \mathbf{let}\ V = M +_b N, \mathbf{group\ by}\ (I,J)\,\} \tag{21}$$

$$Gb := Rb \tag{22}$$

Then pattern compilation is applied to remove the patterns:

$$Rb := \{\,((X.\_1.\_1, Y.\_1.\_2), (min_b/V)) \mid (X \leftarrow Gb, \\ Y \leftarrow Gb, X.\_1.\_2 = Y.\_1.\_1, \mathbf{let}\ V = X.\_2 +_b \\ Y.\_2, \mathbf{group\ by}\ (X.\_1.\_1, Y.\_1.\_2)\,\} \tag{23}$$

$$Gb := Rb \tag{24}$$

Then, the transformed comprehension (23) is translated to Spark SQL program on block arrays using Rule (18):

```
Rb := select   struct(X._1._1, Y._1._2), tile_min(
                collect_list(min_tiles(X._2, Y._2)))
      from     Gb X join Gb Y on X._1._2 = Y._1._1
      group by X._1._1, Y._1._2                        (25)
```

As a second example, let's consider one iteration of PageRank computation of a graph using the following array equation: $r' = \beta M r + (1-\beta)/n$. At first, the rank vector r is initialized to $1/n$, where n is the total number of nodes in the graph. The value $(1-\beta)/n$ is assigned to variable a which corresponds to a random jump to a random page with probability $(1-\beta)$. In the

transition matrix M, M_{ij} has value $1/k$ if the page j has k outgoing edges. Then M is multiplied with β and assigned to the variable G:

$$
\begin{array}{l}
\textbf{var}\ s : \text{vector}[\text{Double}] = \text{vector}(); \\
\textbf{for}\ i = 0, n-1\ \textbf{do}\ \{ \\
\quad s[i] := 0; \\
\quad \textbf{for}\ j = 0, n-1\ \textbf{do}\ \{ \\
\quad\quad s[i]\ +\!= G[i,j] * r[j]; \\
\quad \} \\
\quad r[i] := s[i] + a; \\
\}
\end{array}
$$

The comprehensions derived from this program are:

$$s := \{\, (i, +/v) \mid ((i,k),m) \leftarrow G, (k',n) \leftarrow r, \; k = k', \textbf{let}\, v = m * n, \textbf{group by}\, i \,\} \quad (26)$$

$$r := \{\, (i, v + a)) \mid (i,v) \leftarrow s \,\} \quad (27)$$

These comprehensions are transformed to comprehensions on block arrays by applying the semantic function $\mathcal{S}$ using Rules (16) and (17):

$$sb := \{\, (I, +_b/V) \mid ((I,K),M) \leftarrow Gb, (K',N) \leftarrow rb, K = K', \textbf{let}\, V = M *_b N, \textbf{group by}\, I \,\} \quad (28)$$

$$rb := \{\, (I, V +_b a)) \mid (I,V) \leftarrow sb \,\} \quad (29)$$

Then pattern compilation is applied to remove the patterns using formulas from our earlier work:

$$sb := \{\, X.\_1.\_1, +/V) \mid (X \leftarrow Gb, Y \leftarrow rb, \textbf{let}\, V = X.\_2 *_b Y.\_2, X.\_1.\_2 = Y.\_1, \textbf{group by}\, X.\_1.\_1) \,\} \quad (30)$$

$$rb := \{\, (X.\_1, X.\_2 + a) \mid X \leftarrow sb \,\} \quad (31)$$

The first transformed comprehension (30) is translated to Spark SQL program on block arrays using Rule (18):

$$
\begin{array}{rl}
sb := \textbf{select} & \text{X.\_1.\_1}, \mathit{tile\_sum}(\mathit{collect\_list}(\\
 & \qquad \mathit{mult\_tiles}(\text{X.\_2}, \text{Y.\_2}))) \\
\textbf{from} & \text{Gb}\, X\ \textbf{join}\ \text{rb}\, Y\ \textbf{on}\ \text{X.\_1.\_2} = \text{Y.\_1} \\
\textbf{group by} & \text{X.\_1.\_1}
\end{array} \quad (32)
$$

For the second comprehension, the header contains a scalar operation in the second column, which is translated to a block operation using the special function vec_sum:

$$
\begin{array}{rll}
rb := & \textbf{select} & \text{X.\_1}, \text{X.\_2} + \text{a} \\
 & \textbf{from} & \mathcal{Q}[\![\text{X} \leftarrow \text{sb}]\!] \\
 & \textbf{where} & \mathcal{P}[\![\text{X} \leftarrow \text{sb}]\!] \\
 & \textbf{group by} & \mathcal{G}[\![\text{X} \leftarrow \text{sb}]\!] \\
= & \textbf{select} & \text{X.\_1}, \text{vec\_sum}(\text{X.\_2} + \text{a}) \\
 & \textbf{from} & \text{sb}\, \text{X}
\end{array}
$$

6 Performance Evaluation

Our translation system SQLgen is implemented on top of DIABLO [19]. At first, array-based loops are translated to monoid comprehensions, and then the monoid comprehensions are translated to Spark SQL by SQLgen.

6.1 Evaluation of 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) [46]. 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. Hence, 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 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

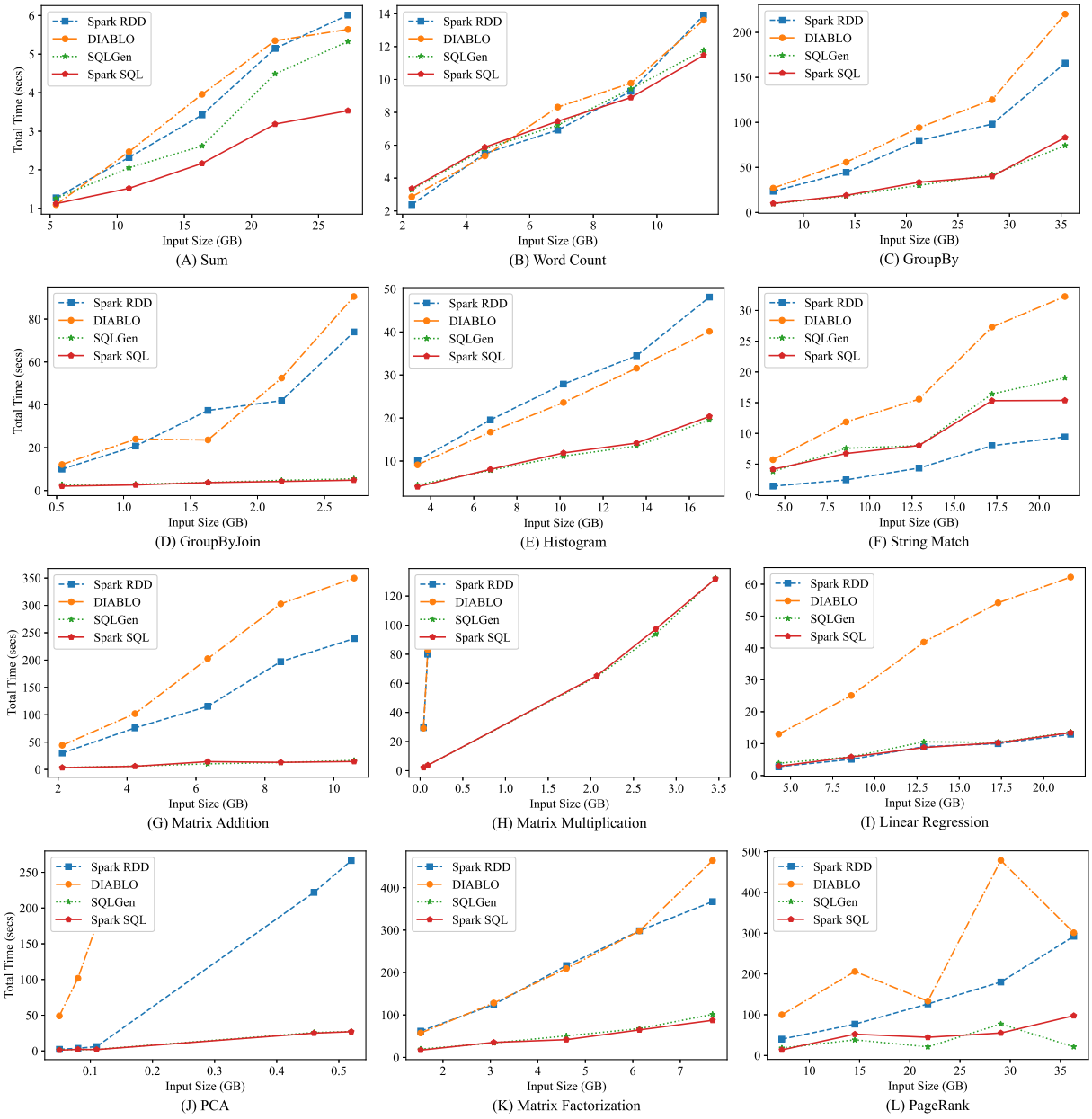


Figure 2: Performance of SQLgen relative to DIABLO, hand-written Spark RDD code, and Spark SQL code

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 were $14.6\times$ 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 and the translation time of loops to Spark SQL programs is insignificant. 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.

6.2 Evaluation of OSQLgen

Our solution to generate optimized queries based on semirings is integrated into our existing system SQLGen [33] which in turn is implemented on top of DIABLO [19]. The input programs are translated to monoid comprehensions and then to optimized SQL programs. The generated query is compiled to bytecode at compile-time, which in turn is embedded in the bytecode generated by the rest of the Scala program.

In the first part, we evaluated the performance of our system on matrix-matrix multiplication, matrix-vector multiplication, and linear regression on synthetic datasets. We compared the performance of our system with MLlib, hand-written SQL programs on coordinate and block arrays. In the second part, we evaluated the performance of our system on all-pairs shortest path, and PageRank on synthetic datasets. We compared the performance of our system with GraphX [21], GraphFrames [11], handwritten Spark SQL programs on coordinate, and block arrays. The platform used in our experiments is the XSEDE Expanse cloud computing infrastructure at SDSC (San Diego Supercomputer Center) [45]. Each program was run on a cluster of 5 nodes where each node is equipped with 128 core AMD EPYC 7742 processor with 2.5GHz clock speed,

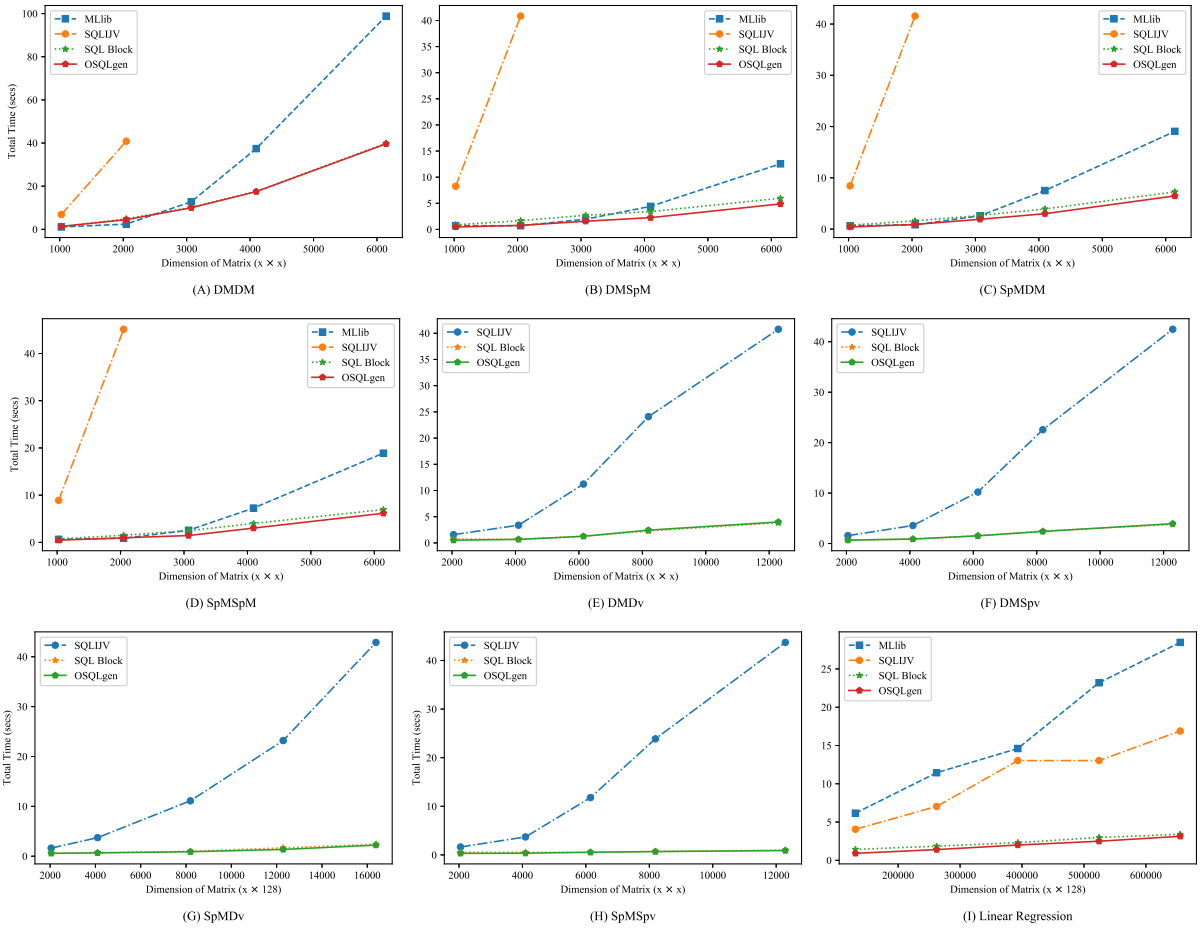


Figure 3: Performance of OSQLgen on linear algebra operations relative to MLlib, hand-written Spark SQL on programs on coordinate and block arrays

256 RAM and 1TB SSD. The programs were run on Apache Spark 3.0.1 on Apache Hadoop 3.2.0. Each Spark executor on Spark was configured to have 30 cores and 60 GB memory. So there were 4 executors per node, giving a total of 20 executors, from which 2 were reserved. The input data for each program were generated using GraphX synthetic graph generators. 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. Hence, each data point in the plots in Figure 3 and Figure 4 represents the mean time on the rest of the 3 iterations.

First, we compared the performance of OSQLgen on some linear algebra operations as shown in Figure 3.

Matrix-matrix Multiplication: We compared the performance of OSQLgen on matrix product of two matrices with four different combinations: Dense-Dense (DMDM), Dense-Sparse (DMSpM), Sparse-Dense (SpMDM), and Sparse-Sparse (SpMSpM). The dense matrices were complete and the sparse matrices had 87.5% sparsity. Each program was run for 5 sizes of input datasets with each array block of size 1024, except SQL programs on coordinate arrays which were run for first 2 datasets because it was very slow. The largest dense matrix generated in these experiments had $5,120 \times 5,120$ elements and the largest sparse matrix had 640×640 elements. For these experiments, the average speedups of OSQLgen were 2.09× (DMDM), 2.06× (DMSpM), 2.41× (SpMDM), and 2.5× (SpMSpM) over MLlib programs and 10.86× (DMDM), 40.22× (DMSpM), 38.02× (SpMDM), and 37.7× (SpMSpM) over handwritten Spark SQL programs on coordinate arrays and had similar performance to the hand-written Spark SQL program on block arrays.

Matrix-vector Multiplication: We compared the performance of OSQLgen on the products of matrices

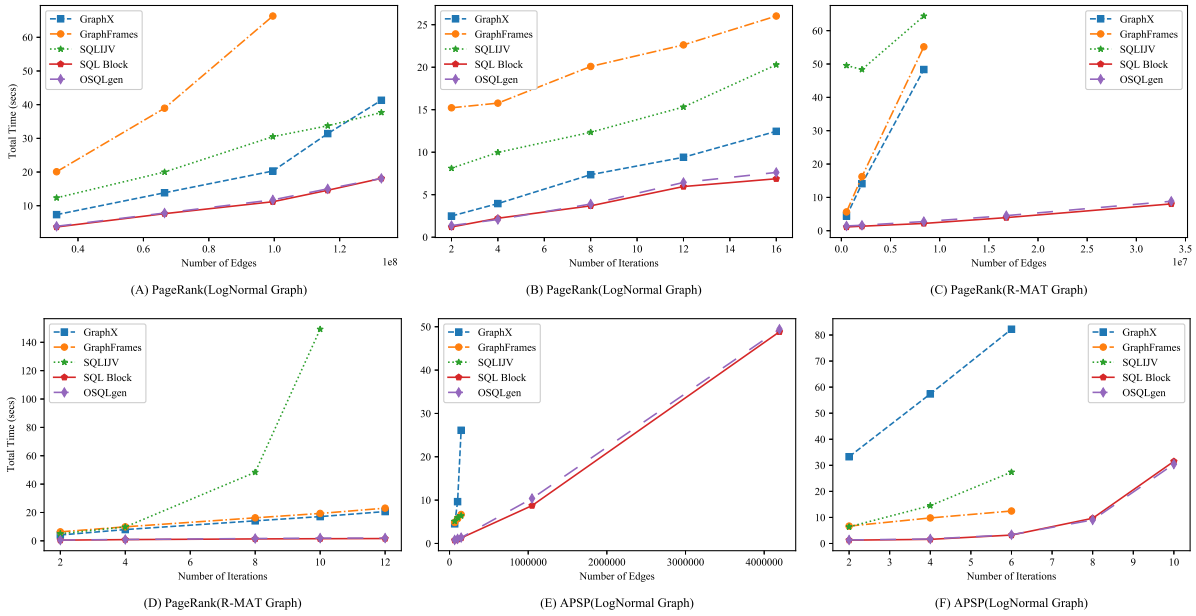


Figure 4: Performance of OSQLgen on graph algorithms relative to GraphX, GraphFrames, and hand-written Spark SQL on coordinate and Block arrays

and vectors with four different combinations: Dense-Dense (DMDv), Dense-Sparse (DMSpv), Sparse-Dense (SpMDv), and Sparse-Sparse (SpMSpv). In these experiments, we couldn't compare the performance of our system with Spark MLlib since Spark MLlib doesn't have block vector. The dense matrices and vectors were complete and the sparse matrices and vectors had 87.5%, and 50% sparsity respectively. Each program was run for 5 sizes of input datasets with each array block of size 2048. The largest dense matrix generated in these experiments had $12,288 \times 12,288$ elements and the largest sparse matrix had 1536×1536 elements. On the other hand, the largest dense vector generated in these experiments had $12,288$ elements and the largest sparse matrix had 6144 elements. For these experiments, the average speedups of OSQLgen were 9.13× (DMDv), 8.53× (DMSpv), 14.5× (SpMDv), and 30.18× (SpMSpv) over handwritten Spark SQL programs on coordinate arrays and had similar performance to the hand-written Spark SQL program on block arrays.

Linear Regression: We compared the performance of OSQLgen for one iteration of the linear regression algorithm using the following formula: $theta = theta - (a * 1/m * X^T) \times (X \times theta - y)$ where $a, m, theta$ and y represent learning rate, number of examples, parameter and label vector respectively. The feature matrix, parameter and label vector were all dense. Each program was run for 5 sizes of input datasets with each array block of size 128. The largest dense matrix generated in these experiments had 131072×128 elements. For these experiments, the average speedups of OSQLgen were 8.44×, and 5.44× over MLlib and handwritten Spark SQL program on coordinate arrays respectively and had similar performance to the hand-written Spark SQL program on block arrays.

Next, we compared the performance of OSQLgen on some graph algorithms as shown in Figure 4. In the first set of experiments, we compare the performance of graph algorithms on different graph sizes by keeping the block size and number of iterations fixed. In the second set of experiments, we compare the performances of graph algorithms on different numbers of iterations and by keeping the graph size and block size fixed.

PageRank: This program assigns a rank to each vertex of a graph using the PageRank algorithm which measures its importance relative to the other vertices in the graph. We compare the performance of OSQLgen with the PageRank algorithms provided by the GraphX, GraphFrames, and hand-written Spark SQL program on coordinate and block arrays.

The input graph used in experiment *A* was generated by the GraphX log-normal graph generator with parameters, mean of out-degree distribution, μ = 4.0, and standard deviation of out-degree distribution, σ =1.3. The largest graph generated in this experiment had 2^{20} vertices and 2^{27}(approx.) edges. Each program was run for 8 iterations for 5 sizes of input datasets, except

GraphFrames which was run for the first 3 datasets because it was very slow. For this experiment, the speedup ranges of OSQLgen were 1.81×-2.28×, 5.1×-5.9×, and 2.1×-3.3× with average speedups of 2.01×, 5.21×, and 2.62× over GraphX, GraphFrames, and handwritten Spark SQL program on coordinate arrays respectively and had similar performance to the hand-written Spark SQL program on block arrays.

In experiment *B*, we evaluate the performance of OSQLgen with the increasing number of iterations (up to 18) for a fixed size of input graph with 2^{18} vertices and 2^{25}(approx.) edges generated by GraphX log-normal graph generator with the same parameters as in experiment *A*. For this experiment, the speedup ranges of OSQLgen were 1.58×-2.07×, 3.8×-12.7×, and 2.6×-6.7× with average speedups of 1.84×, 6.55×, and 4.02× over GraphX, GraphFrames, and handwritten Spark SQL program on coordinate arrays respectively and had similar performance to the hand-written Spark SQL program on block arrays. In experiment *C*, we use GraphX R-MAT (Recursive MATrix) graph generator with parameters a=0.45, b=0.15, c=0.15, d=0.25 for our input datasets. The largest graph generated in this experiment had 2^{19}(approx.) vertices and 2^{25} edges. Each program was run for 8 iterations for 3 sizes of input datasets for GraphX, GraphFrames, and Spark SQL on coordinate arrays and 5 sizes of input dataset for OSQLgen, and hand-written Spark SQL programs on block arrays. For this experiment, the speedup ranges of OSQLgen were 3.91×-22.01×, 5.02×-25.21×, and 29.43×-44.05× with average speedups of 12.14×, 14.06×, and 36.37× over GraphX, GraphFrames, and handwritten Spark SQL program on coordinate arrays respectively and had similar performance to the hand-written Spark SQL program on block arrays.

In experiment *D*, we evaluate the performance of OSQLgen with increasing number of iterations (up to 18) for a fixed size of input graph with 2^{15} vertices and 2^{23}(approx.) edges generated by GraphX R-MAT graph generator with the same parameters as in experiment *C*. Each program was run for 5 sizes of input datasets except hand-written Spark SQL programs on coordinate arrays which were run for 4 sizes of input datasets. For this experiment, the speedup ranges of OSQLgen were 8.58×-12.8×, 11.41×-14.29×, and 10.48×-99.01× with average speedups of 10.46×, 12.73×, and 39.10× over GraphX, GraphFrames, and handwritten Spark SQL program on coordinate arrays respectively and had similar performance to the hand-written Spark SQL program on block arrays.

All-pairs Shortest Path: This program computes the shortest cost path among all pairs of vertices. The input graph used in experiment *E* was synthetic data generated by the GraphX log-normal graph generator with the same parameters as in experiment *A*. We compare the performance of OSQLgen with hand-written programs written in GraphX, GraphFrames, and Spark SQL program on coordinate, and block arrays. The largest graph generated in this experiment had 2^{11} vertices and 2^{18}(approx.) edges. The rest of the entries between the edges were filled with 0 if it was in between a vertex to itself else they are filled with ∞. The total number of edges of our largest graph was 2^{22}. Each program was run for 2 iterations for 3 sizes of input datasets for GraphX, GraphFrames, and Spark SQL on coordinate arrays and 5 sizes of input datasets for OSQLgen, and hand-written Spark SQL programs on block arrays. Since the size of the first 3 datasets are significantly smaller than the last two datasets the data points of the first 3 experiments are very close to each other in the figure. For this experiment, the speedup ranges of OSQLgen were 5.97×-20.53×, 5.25×-6.4×, and 5.02×-6.77× with average speedups of 12.1×, 5.8×, and 5.93× over GraphX, GraphFrames, and handwritten Spark SQL program on coordinate arrays respectively and had similar performance to the hand-written Spark SQL program on block arrays.

In experiment *F*, we evaluate the performance of OSQLgen with increasing number of iterations (up to 10) for a fixed size of input graph generated by GraphX log-normal graph generator with the same parameters as in experiment *A*. The largest graph generated in this experiment had 1.5×2^8 vertices and 2^{15}(approx.) edges. The rest of the entries between the edges were filled with 0 if it was in between a vertex to itself else they are filled with ∞. The total number of edges of our largest graph was 9×2^{14}. Each program was run for for 3 sizes of input datasets for GraphX, GraphFrames, and Spark SQL on coordinate arrays and 5 sizes of input datasets for OSQLgen, and hand-written Spark SQL programs on block arrays. For this experiment, the speedup ranges of OSQLgen were 25.51×-36.23×, 3.87×-6.19×, and 5.02×-9.16× with average speedups of 29.3×, 5.1×, and 7.56× over GraphX, GraphFrames, and handwritten Spark SQL program on coordinate arrays respectively and had similar performance to the hand-written Spark SQL program on block arrays.

Finally, we compared the performance of OSQLgen for linear regression, PageRank, and all-pairs shortest path problems on real datasets as shown in Figure 5. For linear regression, we used Car dataset from Craigslist [39] and Housing dataset of England and Wales [22]. In these experiments, all the systems performed similar except MLlib which performed slower than other systems on Housing dataset. For PageRank algorithm, we used Google web graph [26] and twitter dataset [27]. In these experiments, GraphX, OSQLgen, and hand-written Spark SQL program on block arrays

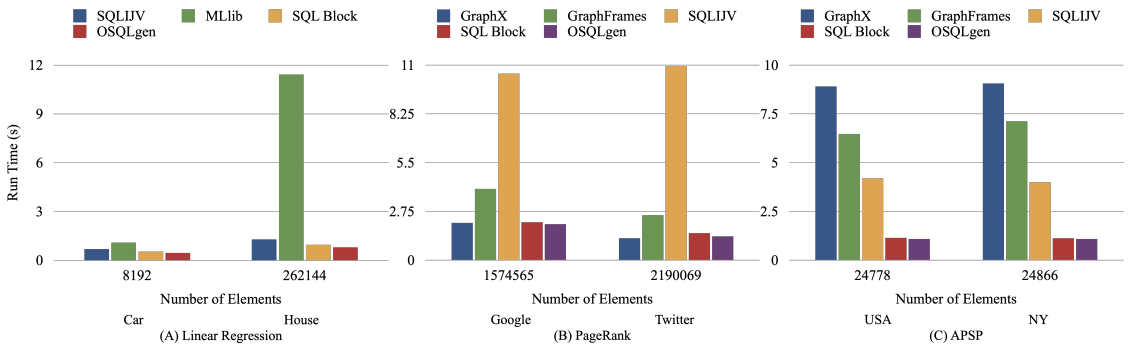


Figure 5: Performance of OSQLgen on graph algorithms on real datasets relative to GraphX, GraphFrames, MLlib and hand-written Spark SQL on coordinate and Block arrays

performed faster than GraphFrames and SQL program on coordinate arrays. For all-pairs shortest path problem, we have used USA and New York road graphs [14]. In these experiments, OSQLgen performed similar to hand-written Spark SQL program on block arrays and significantly faster than GraphX, GraphFrames and SQL program on coordinate arrays.

From all these experiments, we see that programs generated by OSQLgen have similar performance to the hand-written Spark SQL programs on block arrays and perform significantly faster than MLlib, GraphX, GraphFrames, and Spark SQL programs on coordinate arrays.

7 Conclusion and Future Work

We have presented two frameworks for translating programs on very large arrays, SQLgen and OSQLgen. These frameworks translate array-based loop programs to Spark SQL queries. SQLgen translates these programs to SQL queries on coordinate arrays, while OSQLgen translates them to more efficient SQL queries on block arrays, provided that these programs match a semiring pattern. The block computations used in OSQLgen queries are implemented as user-defined functions (UDF) over blocks. As a future work, we plan to extend our methods to translate loop-based programs to other SQL-based systems, such as Flink SQL. Since Spark has recently started providing GPU support, as a future work, we plan to implement our UDFs that implement block operations on GPUs. We also plan to use a real imperative language to express array-based loops, such as Java or C++.

Acknowledgements

This work used the Extreme Science and Engineering Discovery Environment (XSEDE) [45] Expanse, which is supported by National Science Foundation grant number ACI-1548562 through allocation TG-CCR200036.

References

[1] S. K. Abdali and B. D. Saunders, "Transitive closure and related semiring properties via eliminants," *Theoretical Computer Science*, vol. 40, pp. 257–274, 1985.

[2] 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.

[3] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *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. [Online]. Available: https://spark.apache.org/sql/

[4] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, and S. Tatikonda, "Systemml: Declarative machine learning on spark," *Proc. VLDB Endow.*, vol. 9, no. 13, p. 1425–1436, 2016.

[5] R. Bosagh Zadeh *et al.*, "Matrix computations and optimization in apache spark," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 31–38.

[6] R. Bosagh 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 *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Association for Computing Machinery, 2016, p. 31–38.

[7] 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.

[8] 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.

[9] R. Cytron, "Doacross: Beyond vectorization for multiprocessors," in *Proc. of the Int. Conf. on Parallel Processing, 1986*, 1986.

[10] 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.

[11] A. Dave, A. Jindal, L. E. Li, R. Xin, J. Gonzalez, and M. Zaharia, "Graphframes: an integrated api for mixing graph and relational queries," in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, 2016, pp. 1–8.

[12] T. A. Davis, "Algorithm 1000: Suitesparse: Graphblas: Graph algorithms in the language of sparse linear algebra," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 4, pp. 1–25, 2019.

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

[14] DIMACS, "9th dimacs implementation challenge - shortest paths," accessed November, 2021. [Online]. Available: https://www.diag.uniroma1.it/challenge9/download.shtml

[15] K. Ebcioğlu, "A compilation technique for software pipelining of loops with conditional jumps," in *Proceedings of the 20th annual workshop on Microprogramming*. ACM, 1987, pp. 69–79.

[16] K. Ekanadham, W. P. Horn, M. Kumar, J. Jann, J. Moreira, P. Pattnaik, M. Serrano, G. Tanase, and H. Yu, "Graph programming interface (gpi) a linear algebra programming model for large scale graph computations," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2016, pp. 72–81.

[17] 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.

[18] M. A. 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. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[19] 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.

[20] Y. Geng, X. Huang, M. Zhu, H. Ruan, and G. Yang, "Scihive: Array-based query processing with hiveql," in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2013, pp. 887–894.

[21] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 599–613.

[22] HM Land Registry, "Uk housing prices paid," accessed November, 2021. [Online]. Available: https://www.kaggle.com/hm-land-registry/uk-housing-prices-paid

[23] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1988, pp. 319–329.

[24] 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.

[25] D. J. Lehmann, "Algebraic structures for transitive closure," *Theoretical Computer Science*, vol. 4, no. 1, pp. 59–76, 1977.

[26] J. Leskovec, "Google web graph," accessed November, 2021. [Online]. Available: https://snap.stanford.edu/data/web-Google.html

[27] J. Leskovec, "Social circles: Twitter," accessed November, 2021. [Online]. Available: https://snap.stanford.edu/data/ego-Twitter.html

[28] 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.

[29] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.

[30] D. Marten and A. Heuer, "Machine learning on large databases: Transforming hidden markov models to sql statements," *Open Journal of Big Data (OJBD)*, vol. 4, no. 1, pp. 22–42, 2017. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:101:1-2017100112181

[31] D. Marten, H. Meyer, D. Dietrich, and A. Heuer, "Sparse and dense linear algebra for machine learning on parallel-rdbms using sql," *Open Journal of Big Data (OJBD)*, vol. 5, no. 1, pp. 1–34, 2019. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:101:1-201812231834106917295 7

[32] 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.

[33] M. H. Noor and L. Fegaras, "Translation of array-based loops to spark sql," in *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2020, pp. 469–476.

[34] M. H. Noor and L. Fegaras, "Translation of array-based graph programs to spark sql on block arrays," in *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, 2021.

[35] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson, "The tiledb array data storage manager," *Proc. VLDB Endow.*, vol. 10, no. 4, p. 349–360, 2016.

[36] 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.

[37] 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.

[38] A. Rajaraman and J. D. Ullman, *Mining of massive datasets*. Cambridge University Press, 2011.

[39] A. Reese, "Used cars dataset," accessed November, 2021. [Online]. Available: https://www.kaggle.com/austinreese/craigslist-carstrucks-data

[40] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[41] E. Soroush, M. Balazinska, S. Krughoff, and A. Connolly, "Efficient iterative processing in the scidb parallel array engine," in *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*. Association for Computing Machinery, 2015.

[42] E. Soroush, M. Balazinska, and D. Wang, "Arraystore: A storage manager for complex parallel array processing," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 2011, p. 253–264.

[43] A. Takahashi and S. Sedukhin, "Parallel blocked algorithm for solving the algebraic path problem on a matrix processor," in *International Conference on High Performance Computing and Communications*. Springer, 2005, pp. 786–795.

[44] R. E. Tarjan, "A unified approach to path problems," *Journal of the ACM (JACM)*, vol. 28, no. 3, pp. 577–593, 1981.

[45] J. Towns, T. Cockerill, M. Dahan, I. Foster *et al.*, "Xsede: Accelerating scientific discovery," *Computing in Science & Engineering*, vol. 16, no. 5, pp. 62–74, 2014.

[46] 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.

[47] Y. Wang, W. Jiang, and G. Agrawal, "Scimate: A novel mapreduce-like framework for multiple scientific data formats," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, 2012, pp. 443–450.

[48] Yahoo, "TensorFlowOnSpark," accessed November, 2021. [Online]. Available: https://github.com/yahoo/TensorFlowOnSpark

[49] 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.

APPENDIX A: CORRECTNESS PROOF

Block arrays: for k-dimensional arrays we have k-dimension blocks of size $\overbrace{D * D * ... * D}^{\text{k}} = D^k$ for fixed D. A block array Ab is converted to a coordinate array A using mapping G:

$$G(Ab) = \{\, (\overline{I} * D + \overline{i}, m) \mid (\overline{I}, A) \leftarrow Ab, \\ (\overline{i}, m) \leftarrow F(A) \,\}$$

where F converts a block to a coordinate list:

$$F(A) = \{\, (\overline{i}, A[\overline{i}]) \mid \overline{i} \leftarrow 0 \mathinner{..} D \,\}$$

When $k = 2$, we have:

$$G(Ab) = \{\, ((I * D + i,\ J * D + j), m) \mid \\ ((I, J), M) \leftarrow Ab,\ ((i, j), m) \leftarrow F(M) \,\}$$

where $F(M)$ is:

$$\{\, ((i, j), M[i, j]) \mid i \leftarrow 0 \mathinner{..} D, j \leftarrow 0 \mathinner{..} D \,\}$$

The mapping of application of additive monoid on two block arrays to coordinate array is defined as:

$$\begin{aligned} F(M \oplus_b N) &= \{\, ((i, k), m \oplus n) \mid ((i, j), m) \\ &\quad \leftarrow F(M),\ ((i', j'), n) \leftarrow F(N), \\ &\quad i = i', j = j' \,\} \\ &= \{\, ((i, k), M[i, j] \oplus N[i, j]) \mid \\ &\quad i \leftarrow 0 \mathinner{..} D,\ j \leftarrow 0 \mathinner{..} D \,\} \end{aligned}$$

The mapping of application of multiplicative monoid on two block arrays to coordinate array is defined as:

$$\begin{aligned} F(M \otimes_b N) &= \{\, ((i, j), \oplus/v) \mid ((i, k), m) \leftarrow F(M), \\ &\quad ((k', j), n) \leftarrow F(N), k = k', \textbf{let} \\ &\quad v = m \otimes n, \textbf{group by } (i, j) \,\} \\ &= \{\, ((i, j), \oplus/v) \mid i \leftarrow 0 \mathinner{..} D, \\ &\quad k \leftarrow 0 \mathinner{..} D, j \leftarrow 0 \mathinner{..} D, k = k', \\ &\quad \textbf{let } M[i, k] \otimes N[k, j], \\ &\quad \textbf{group by } (i, j) \,\} \end{aligned}$$

The semiring comprehension q on k-dimensional coordinate arrays A, and B is defined as:

$$q(A, B) = \{\, (k, \oplus/v) \mid (\overline{i}, m) \leftarrow A,\ (\overline{j}, n) \leftarrow B, \\ \rho_1(\overline{i}) = \rho_2(\overline{j}), \textbf{let } v = m \otimes n, \\ \textbf{group by } k : f(\overline{i}, \overline{j}) \,\}$$

when $k = 2$, $q(A, B)$ equals:

$$\{\, ((i, j), \oplus/v) \mid ((i, k), m) \leftarrow A,\ ((k', j), n) \leftarrow B, \\ k = k', \textbf{let } v = m \otimes n, \textbf{group by } (i, j) \,\}$$

The semiring comprehension Q on block arrays Ab, and Bb is:

$$Q(Ab, Bb) = \{\, (K, \oplus_b/V) \mid (\overline{I}, M) \leftarrow Ab, \\ (\overline{J}, N) \leftarrow Bb,\ \rho_1(\overline{I}) = \rho_2(\overline{J}), \\ \textbf{let } V = M \otimes_b N, \\ \textbf{group by } K : f(\overline{I}, \overline{J}) \,\}$$

When $k = 2$, $Q(Ab, Bb)$ equals to:

$$\{\, ((I, J), \oplus_b/V) \mid ((I, K), M) \leftarrow Ab, \\ ((K', J), N) \leftarrow Bb, K = K', \\ \textbf{let } V = M \otimes_b N, \textbf{group by } (I, J) \,\}$$

Theorem 1 *Given the block arrays Ab, and Bb, the semiring comprehension q on these arrays after applying G is equivalent to G applied to the semiring comprehension Q on Ab, and Bb:*

$$\forall Ab, Bb :\ q(G(Ab), G(Bb)) = G(Q(Ab, Bb))$$

where G maps block arrays to coordinate arrays.

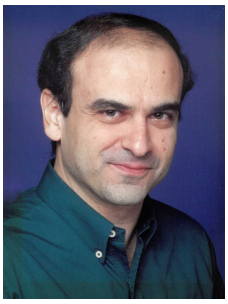
Proof: We have provided the proof for $k = 2, \oplus = +, \oplus_b = +_b, \otimes = *, \otimes_b = *_b$:

$$\begin{aligned} &q(G(Ab), G(Bb)) \\ &= \{\, ((i, j), +/v) \mid ((i, k), m) \leftarrow G(Ab), \\ &\quad ((k', j), n) \leftarrow G(Bb), k = k', \textbf{let } v = m * n, \\ &\quad \textbf{group by } (i, j) \,\} \\ &= \{\, ((I * D + i, J * D + j), +/v) \mid ((I, K), M) \\ &\quad \leftarrow Ab, ((i, k), m) \leftarrow F(M), ((K', J), N) \\ &\quad \leftarrow Bb, ((k', j), n) \leftarrow F(N), \\ &\quad K' * D + k' = K * D + k, \textbf{let } v = m * n \\ &\quad \textbf{group by } (i, j) \,\} \\ &= \{\, ((I * D + i, J * D + j), +/v) \mid ((I, K), M) \\ &\quad \leftarrow Ab, ((i, k), m) \leftarrow F(M), ((K', J), N) \\ &\quad \leftarrow Bb, ((k', j), n) \leftarrow F(N), K = K', \\ &\quad \textbf{let} k = k', v = m * n, \textbf{group by } (i, j) \,\} \\ &\quad \textit{(since } K' * D + k' = K * D + k \textit{ and} \\ &\quad\ k, k' < D \textit{ implies } K{=}K' \textit{ and } k{=}k') \end{aligned}$$

$$
\begin{aligned}
&= \{\, ((I*D+i, J*D+j), +/v) \mid \\
&\quad ((I,K),M) \leftarrow Ab, ((K',J),N) \leftarrow Bb, \\
&\quad K = K', \mathbf{group\ by}\ (I,J), \\
&\quad ((i,k),m) \leftarrow F(M), ((k',j),n) \leftarrow F(N), \\
&\quad k = k', \mathbf{let}\ v = m*n, \mathbf{group\ by}\ (i,j)\,\} \\
&\qquad \textit{(since } I = i/D \textit{ and } J = j/D\textit{, then} \\
&\qquad\quad \textbf{\textit{group by}}\ (i,j)\ \textit{implies}\ \textbf{\textit{group by}}\ (I,J)) \\
&= \{\, ((I*D+i, J*D+j), +/v) \mid \\
&\quad ((I,J),V) \leftarrow Q(Ab,Bb), V = M *_b N, \\
&\quad ((i,k),m) \leftarrow F(M), ((k',j),n) \leftarrow F(N), \\
&\quad k = k', \mathbf{let}\ v = m*n, \mathbf{group\ by}\ (i,j)\,\} \\
&= \{\, ((I*D+i, J*D+j), +/v) \mid \\
&\quad ((I,J),V) \leftarrow Q(Ab,Bb), V = M *_b N, \\
&\quad ((i,j),v) \leftarrow F(M *_b N)\,\} \\
&= \{\, ((I*D+i, J*D+j), +/v) \mid \\
&\quad ((I,J),V) \leftarrow Q(Ab,Bb), ((i,j),v) \leftarrow F(V)\,\} \\
&= G(Q(Ab,Bb)) \qquad \square
\end{aligned}
$$

AUTHOR BIOGRAPHIES

Md Hasanuzzaman Noor is an Adjunct Professor in the Computer Science & Engineering Department at the University of Texas at Arlington. Before joining UTA, he was a lecturer at Port City International University. He graduated with a BSc from Chittagong University of Engineering & Technology in Computer Science & Engineering and a PhD in Computer Science from the University of Texas at Arlington. His research interest is in the area of big data, cloud computing, distributed system, and machine learning.



Leonidas Fegaras is an Associate Professor in the Computer Science & Engineering department at the Univ. of Texas at Arlington (UTA). Prior to joining UTA, he was a Senior Research Scientist at the OGI School of Science and Engineering. Fegaras graduated with a B.Tech in EE from the National Technical Univ. of Athens, Greece in 1985, an MS in ECE from the Univ. of Massachusetts- Amherst in 1987, and a PhD in Computer Science from Univ. of Massachusetts-Amherst in 1992. His research interests include Big Data management, distributed computing, Web data management, data stream processing, and query processing and optimization.

Tanvir Ahmed Khan is a first year Ph.D. student in the Computer Science & Engineering Department at the University of Texas at Arlington. Before joining UTA, he was a software engineer at Samsung Research, Bangladesh. He graduated with a BSc from Dhaka University in Electrical Engineering. His research interest is in the area of big data, cloud computing, distributed system, and machine learning.

Tanzima Sultana is a second year Ph.D. student in the Computer Science & Engineering Department at the University of Texas at Arlington. Before joining UTA, she was a software engineer at Samsung Research, Bangladesh. She graduated with a BSc from Khulna University of Engineering and Technology in Computer Science & Engineering. Her research interest is in the area of big data, cloud computing, distributed system, and machine learning.