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

by
MD HASANUZZAMAN NOOR

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
December 2021

To Vaia

ACKNOWLEDGEMENTS

I express my gratitude to my advisor Dr. Leonidas Fegaras for giving me the opportunity to research under his supervision. I have always learned something new from him every time we had a conversation. His wisdom on computer science in various areas along with this technical knowledge of the latest technologies have guided me to reach the biggest milestone of my life.

I am thankful to the department staff and most importantly our faculty members for continuously supporting us even during the challenging time of the COVID pandemic.

I am thankful to my parents Bilkis Akter and Nurul Islam for always believing in me. I am thankful to vaia for motivating me to pursue a Ph.D. I am also thankful to my other siblings Rizvi and Mamun for always looking after me.

I am thankful to Habib vai and Ata vai for being my guide during this journey. There were a lot of people during this journey without whom I couldn't reach this milestone including Belal, Sadat vai, Hasan vai, Toki, Jon, Luke, Sophie, Daniella, Steve, my mother-in-law Sarah, and my lab mates Aashara, Tanvir and Diba apu.

I am thankful to my wife Savannah for always being there for me and listening to my computer jargon.

November 22, 2021

ABSTRACT

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

Md Hasanuzzaman Noor, Ph.D.

The University of Texas at Arlington, 2021

Supervising Professor: Leonidas Fegaras

Most programs written to operate on data are usually expressed in terms of array operations in sequential loops. However, these programs do not scale to large amount of data generated by scientific experiments and industrial and commercial markets. Given the success of machine learning algorithms on large amount of data and the recent shift of industries to data-driven decision making, the data scientists who are not familiar with Big Data frameworks have to rewrite the sequential programs to distributed data-parallel programs by hand. We present a novel framework, called SQLgen, that automatically translates sequential loops to distributed data-parallel programs. SQLgen translates array-based loops to Spark SQL programs. At first, it translates the input programs to monoid comprehensions, which is the formal framework of our translation model. Then, it translates the comprehensions to Spark SQL programs. We chose Spark SQL because, unlike the Spark Core API programs, Spark SQL programs are optimized by the query optimizer Catalyst. We compare the performance of our generated programs with that of related work on real-world problems and show significant performance gain (up to 78x) while keeping performance close to hand-written Spark SQL programs.

Linear algebra operations, such as matrix multiplication, play a significant role in many machine learning algorithms. The performance of these operations is highly dependent on the storage implementations of these matrices. For example, in a distributed system, computations on block matrices are significantly faster than the computations in the coordinate format in terms of computation and communication cost. Hence, instead of coordinate arrays, these operations can be implemented in block matrices, which are distributed collections of non-overlapping dense/sparse arrays. Moreover, many graph algorithms can be expressed as repetitive computations that resemble matrix multiplication in which the addition and multiplication operations have been replaced with generalized operations that form an algebraic structure known as a semiring. Similar to matrix multiplication, these graph algorithms can be implemented in a distributed system using block arrays. We present a novel framework OSQLgen that automatically parallelizes array-based loop programs containing linear algebra operations and graph programs that are equivalent to a semiring to distributed data-parallel programs on block arrays. 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. On certain problems, OSQLgen is up to 36x faster than GraphX, 25x faster than GraphFrames, 3x faster than MLlib, and 99x faster than hand-written Spark SQL programs on coordinate arrays, giving performance close to that of hand-written Spark SQL programs on block arrays.

TABLE OF CONTENTS

Chapter Page

1. Introduction . . . 1
   1.1 Highlights of Our Approach . . . 3
   1.2 Implementing Block Array Operations . . . 5
   1.3 Dissertation organization . . . 8
2. Related Work . . . 10
   2.1 Program Transformation . . . 10
      2.1.1 Program Transformation in HPC . . . 10
      2.1.2 Program Transformation in DISC Systems . . . 11
      2.1.3 Program Transformation in Database Systems . . . 12
   2.2 Array Storage Systems . . . 12
   2.3 Optimizing Iterative Graph Algorithms . . . 14
3. Formal Framework . . . 15
   3.1 Introduction . . . 15
   3.2 Syntax of the Loop-Based Language . . . 15
   3.3 Recurrence and Restrictions for Parallelization . . . 16
   3.4 Monoid Comprehensions . . . 19
   3.5 DIABLO Framework . . . 23
      3.5.1 Array Representation . . . 23
      3.5.2 Handling Array Updates in a Loop . . . 25
      3.5.3 Eliminating Loop Iterations . . . 27
      3.5.4 Handling Incremental Updates . . . 27
      3.5.5 Program Translation . . . 28
      3.5.6 Examples of Program Translation . . . 28

3.6 Conclusion 30
4. SQLgen: Translation to SQL 31
4.1 Introduction 31
4.2 The SQLgen Framework 32
4.2.1 Translation to SQL 35
4.2.2 Examples of Program Translation 38
4.3 Performance Evaluation 42
4.4 Conclusion 46
5. OSQLgen: Translation to SQL on Block Arrays 47
5.1 Introduction 47
5.2 Background 48
5.2.1 Semiring and Graph Algorithms 49
5.3 The OSQLgen Framework 50
5.3.1 OSQLgen Storage System 53
5.3.2 Translation of Linear Algebra operations to SQL on Block Arrays 55
5.3.3 Translation of Semiring to SQL on Block Arrays 55
5.3.4 Examples of Program Translation 59
5.4 Performance Evaluation 63
5.5 Conclusion 69
6. Conclusion and Future Work 72

LIST OF ILLUSTRATIONS

Figure Page

3.1 Syntax of loop-based programs 16

4.1 Performance of SQLgen relative to DIABLO, hand-written Spark RDD, and Spark SQL 43

5.1 Performance of OSQLgen on linear algebra operations relative to MLlib, hand-written Spark SQL on programs on coordinate and block arrays . 63

5.2 Performance of OSQLgen relative to GraphX, GraphFrames, and hand-written Spark SQL on COO and Block arrays 64

5.3 Performance of OSQLgen on graph algorithms on real datasets relative to GraphX, GraphFrames, MLlib and hand-written Spark SQL on coordinate and Block arrays 68

LIST OF TABLES

Table — Page

5.1 Semirings for Graph Algorithms 49

CHAPTER 1

Introduction

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

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

Since analyzing large amounts of data play an important role in data analysis and in the accuracy of machine learning models, many organizations want to convert their programs, originally written to run on a single computer, to work on current DISC systems so that they can process larger amounts of data. Data analysts and scientists, who are mostly familiar with imperative programming languages and numerical analysis tools that are based on arrays, have to learn new programming paradigms to rewrite their programs to run on DISC systems. This rewriting process slows down the development and deployment process and is non-trivial since the programmers need to address the intricacies and avoid the pitfalls inherent to these frameworks to get optimal performance. The lack of expertise in a particular framework may result in erroneous or suboptimal programs. Furthermore, to achieve flexibility and better performance, instead of using libraries, such as MLlib [10], programmers may often write ad-hoc array-based programs that are specific to their needs. Consequently, instead of rewriting these ad-hoc programs by hand to run on a particular platform, one solution can be to use an automatic translation system that will translate sequential programs with loops to distributed data-parallel programs.

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

1.1 Highlights of Our Approach

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

Consider, for example, a product C of two square matrices $A_{n\times n}$ and $B_{n\times n}$ such that $C_{ij} = \sum_k A_{ik} * B_{kj}$. In a loop-based language, it can be expressed as:

```
**for** $i = 0, d-1$ **do**
  **for** $j = 0, d-1$ **do** {
    $C[i,j] = 0.0;$
    **for** $k = 0, d-1$ **do** {
      $C[i,j]$ += $A[i,k] * B[k,j]);$
  }
}
```

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

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

Here, 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 indexes 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 then translated to the following Spark SQL program, where matrices are represented as tables A, B, and C with schema ((_1, _2), _2):

```
select      struct(A._1._1, B._1._2),
            sum(A._2 * B._2)
from        A join B on A._1._2 = B._1._1
group by    A._1._1, B._1._2
```

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.

1.2 Implementing Block Array Operations

In SQLgen and the related work, 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/sparse array blocks. In Spark [15], a block matrix can be implemented as a distributed collection (an RDD) of fixed-sized dense square tiles of type $\mathrm{RDD}[((\mathrm{Int},\mathrm{Int}),\mathrm{Array}[\mathrm{Double}])]$, where each block $((\mathrm{i},\mathrm{j}),\mathrm{A})$ 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 not only reduces the required storage but also reduces the communication cost of a distributed algorithm, since the amount of data that needs to be transferred over the network is less. Furthermore, for some distributed algorithms, the communication cost is further reduced because the block implementation requires less replication of data.

Consider the matrix multiplication algorithm again where, the coordinate arrays A, and B are represented as tables with schema $\mathrm{A}((\mathrm{i},\mathrm{k}),\mathrm{m})$, and $\mathrm{B}((\mathrm{k}',\mathrm{j}),\mathrm{n})$. If the matrices A and B are of size n^2, 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 [16]. Furthermore, the multiplication between the array blocks can be pushed down to CPU/GPU using efficient Basic Linear Algebra Subprograms (BLAS) routines [17]. Hence, the block matrix approach is superior to the coordinate approach in terms of space, communication, and computation time.

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 $(\oplus,\overline{0}) : S \times S \to S$ with identity $\overline{0}$ and a multiplicative monoid $(\otimes,\overline{1}) : S \times S \to S$ 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. 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 [18]. This can be done by translating array-based loop programs that are equivalent to semirings to programs on block arrays expressed in Spark SQL [14]. 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 $+\infty$.

```
**var** $R$ : matrix[Double] = matrix();
**for** $i = 0, n-1$ **do**
  **for** $j = 0, n-1$ **do** {
    **for** $k = 0, n-1$ **do** {
      $R[i,j] := min(R[i,j], G[i,k] + G[k,j])$; //*update*
    }
    $G[i,j] := R[i,j]$; //*assignment*
  }
```

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:

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

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:

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

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

1.3 Dissertation organization

In chapter 2, we discuss the related work that translate array-based loop programs both in distributed and non-distributed systems. We also discuss about related work in storage systems and graph algorithms in HPC and distributed systems.

In chapter 3, we discuss the theory behind the formal basis of our framework called monoid comprehension. We also discuss about the syntax of the proof-of-concept language that we have used in our system and the translation process of that language to comprehensions.

In chapter 4, we present our first translation system called SQLgen along with the description of the translation process. We also discuss the performance of SQLgen on real-world problems in comparison to related work.

In chapter 5, we present our second translation system called OSQLgen along with the description of the translation process and storage systems. We also discuss the performance of OSQLgen on real-world graph and linear algebra problems in comparison to related work on synthetic and real datasets.

Finally, in chapter 6, we discuss about the limitations of our frameworks and future work.

CHAPTER 2

Related Work

2.1 Program Transformation

In this section, we mention some noteworthy work in the area of program transformation from array-based loops to Big Data systems. First, we mention works in the area of HPC and then in DISC systems.

2.1.1 Program Transformation in HPC

Researchers in HPC explored the problem of automatic parallelization of various kinds of loops. The most common approach for parallelizing loops is DOALL [19]. In DOALL, each iteration is executed in parallel if there is no dependency among the iterations, a restriction commonly known as a loop-carried dependency. DOACROSS [20] addresses this problem of loop-carried dependencies by rewriting the loop to extract the computations that can be performed independently. The independent part is computed in parallel and later synchronized with the dependent part which is computed sequentially. Another approach is DOPIPE [21], a pipelined parallelism where an iteration with a loop-carried dependency is distributed over multiple synchronized loops. Here, each step starts when there is sufficient data available from the previous step.

In HPC, communications are done usually via Message Passing Interface(MPI) which requires coding at the transport layer, for example, decomposition of data structure across processors. Optimization in HPC is slow because to make any changes all MPI code has to be rewritten. There is also no data locality i.e. data are sent to one node to another for a computation which makes the computation slow. Also with the emergence of supercomputing hardware and increasing mean time between failures of the computers, the computations in HPC can be even slower to rebalance the computation on the alternative set of resources. On the other hand, DISC systems don't require the developers to code at the transport layer that saves the development time. Instead of sending data around it sends the computations to the data which reduces network overhead. Most importantly DISC systems provide automatic fault tolerance. So in case of a failure of a node, the DISC systems will rebalance the computations for the developers. The trade-off is DISC systems have a limited communication model compared to MPI. It uses shuffle operation to communicate and the shuffle operation can make computations very slow. However, DISC systems emphasize using this operation as less as possible which makes the system efficient.

2.1.2 Program Transformation in DISC Systems

MOLD [12] is a translator of Java code to Scala code that can be executed either on a single computing node via parallel Scala collections or on a cluster of computers in a distributed manner using Spark. Like DIABLO [13], SQLgen [18] and OSQLgen [22], it uses a group-by operation to parallelize loops with recurrences. That is, the cumulative effects of these recurrences are brought together by grouping the new array values by their destination location. For incremental updates on arrays, the source values of these updates are grouped by the array index of the incremented array and are aggregated in parallel. However, the authors use a template-based rewriting system to match specific templates of Java loops. They use a heuristic search to find the matching templates for each program fragment and to generate the Map-Reduce output program. Another translator is CASPER [11], which translates sequential Java code into semantically equivalent Map-Reduce programs. It uses a program synthesizer to search over the space of sequential program summaries, expressed as IRs. Unlike MOLD, CASPER uses a theorem prover based on Hoare logic to prove that the derived Map-Reduce programs are equivalent to the original sequential programs. Our system differs from both MOLD and CASPER as it translates loops directly to parallel programs using simple meaning preserving transformations, without having to search for rules to apply. The actual rule-based optimization of our translations is done at a second stage using a small set of rewrite rules, thus separating meaning-preserving translation from optimization.

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

2.1.3 Program Transformation in Database Systems

Another area related to automated parallelization for DISC systems is deriving SQL queries from imperative code in a non-distributed setting [23]. Unlike our work, this work addresses aggregates, inserts, and appends to lists but does not address array updates. The work by Luo *et al.* [6] adds a new attribute type to relational schemas to capture arrays that can fit in memory and extends SQL with array operators. Although these SQL queries are evaluated in Map-Reduce, the arrays are not fully distributed. Instead, large matrices must be split into multiple rows as indexed tiles and the programmer must use SQL to implement a matrix operation by correlating these tiles. This makes it hard to specify some matrix operations, such as matrix inversion.

2.2 Array Storage Systems

Many array-processing systems use special storage techniques, such as regular tiling, to achieve better performance on certain array computations. TileDB [24] is an array data storage management system that performs complex analytics on scientific data. It organizes array elements into ordered collections called fragments, where each fragment is dense or sparse, and groups contiguous array elements into data tiles of fixed capacity. Unlike our works, the focus of TileDB is on the I/O optimization of array operations by using small block updates to update the array stores. SciDB [25] is a large-scale data management system for scientific analysis based on an array data model with implicit ordering. The SciDB storage manager decomposes arrays into a number of equal-sized and potentially overlapping chunks, in a way that allows parallel and pipeline processing of array data. Like SciDB, ArrayStore [26] stores arrays into chunks, which are typically the size of a storage block. One of their most effective storage method is a two-level chunking strategy with regular chunks and regular tiles. SciHadoop [27] is a Hadoop plugin that allows scientists to specify logical queries over arrays stored in the NetCDF file format. Their chunking strategy, which is called the Baseline partitioning strategy, subdivides the logical input into a set of partitions (sub-arrays), one for each physical block of the input file. SciHive [28] 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 [29] extends the Map-Reduce API to support the processing of the NetCDF and HDF5 scientific formats, in addition to flat-files. SciMATE supports various optimizations specific to scientific applications by selecting a small number of attributes used by an application and perform data partition based on these attributes. TensorFlow [30] is a dataflow language for machine learning that supports data parallelism on multi- core machines and GPUs but has limited support for distributed computing. SystemML [31] 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. There has also been some recent work on combining linear algebra with relational algebra to let programmers implement ML algorithms on relational database systems [5, 6, 32]. Although many of these systems support block matrices, their runtime systems are based on a library of built-in, hand-optimized linear algebra operations, which is hard to extend with new storage structures and algorithms. Furthermore, many of these systems lack a comprehensive framework for automatic inter-operator optimization, such as finding the best way to form the product of several matrices. Like these systems, our frameworks separates specification from implementation, but, unlike these systems, our system will support ad-hoc operations on array collections, rather than a library of built-in array operations, is extensible with customized storage structures, and uses relational-style optimizations to optimize array programs with multiple operations.

2.3 Optimizing Iterative Graph Algorithms

In current distributed systems, the vertex-centric approach is prevalent. In the vertex centric approach, a graph program is expressed as a sequence of iterations, in each of which, all the vertices were updated by the neighboring vertices in parallel. This was introduced by Pregel [33] and later an open-source version was released named Apache Giraph [34] for Hadoop. Both systems provide APIs that pairs the bulk synchronous parallel (BSP) computation model [35] with a vertex-centric programming model for developers to implement graph algorithms. Spark also provides similar APIs for graph algorithms GraphX [36] which is based on RDD and GraphFrames [37] which is based on DataFrames.

In contrast to the vertex-centric approach, graph algorithms have also been studied in terms of linear algebra [38]. The key idea is that many graph algorithms can be represented in a semiring structure. Basically, a semiring consists of two monoids, one is additive and the other one is multiplicative which corresponds to the matrix multiplication operation. Essentially, the graph algorithms can be represented in terms of matrix-matrix/vector operations. Combinatoral BLAS [39] is one of the most popular systems to exploit this concept. It is implemented for the MPI system. Later, Graph Programming Interface(GPI) [40] was implemented on Spark Scala. This gives an interface to the developers to write graph algorithms while hiding the implementation details such as choosing storage format and matrix representations. Our framework OSQLgen [22] for translating array-based loops containing iterative graph algorithms to a DISC system differs from related work because most recent works are implemented on HPC systems with the exception of GPI. While GPI is implemented on the Spark Core API system, we implement our system on Spark SQL so the system can leverage on Catalyst optimizer and massive parallelism by GPUs if available.

CHAPTER 3

Formal Framework

3.1 Introduction

Our frameworks SQLgen and OSQLgen are built on top of our earlier framework DIABLO(a Data-Intensive Array-Based Loop Optimizer) [13]. The source language used in these frameworks to express loops with array operations is the same proof-of-concept language used in DIABLO. DIABLO translates loop-based programs to comprehensions and then to Spark Core API programs. In this chapter, we discuss the syntax of our proof-of-concept language, comprehensions, and the translation from loop programs to comprehensions.

3.2 Syntax of the Loop-Based Language

The syntax of the loop-based language is given in Figure 3.1. This is a proof-of-concept loop-based language; many other languages, such as Java or C, can be used instead. Types of values include parametric types for various kinds of collections, such as vectors, matrices, key-value maps, bags, lists, etc. To simplify our translation rules and examples in this section, we do not allow nested arrays, such as vectors of vectors. There are two kinds of assignments, an incremental update $d \oplus\!= e$ for some commutative operation $\oplus$, which is equivalent to the update $d := d \oplus e$, and all other assignments $d := e$. To simplify translation, variable declarations, **var** $v : t = e$, cannot appear inside for-loops. There are two kinds of for-loops that can be parallelized: a for-loop in which an index variable iterates over a range of integers, and a for-loop in which a variable iterates over the elements of a collection, such as the values of an array. Our current framework generates sequential code from a while-loop. Furthermore, if a for-loop contains a while-loop in its body, then this

Type:			
t	$::=$	v	basic type (int, float, ...)
	$\mid$	$v[t]$	parametric type
	$\mid$	$(t_1, \ldots, t_n)$	tuple type
	$\mid$	$\langle\, A_1 : t_1, \ldots, A_n : t_n \,\rangle$	record type
Expression:			
e	$::=$	d	a destination (L-value)
	$\mid$	$e_1 \star e_2$	any binary operation $\star$
	$\mid$	$(e_1, \ldots, e_n)$	tuple construction
	$\mid$	$\langle\, A_1{=}\, e_1, \ldots, A_n{=}\, e_n \,\rangle$	record construction
	$\mid$	$const$	constant (int, float, ...)
Destination:			
d	$::=$	v	variable
	$\mid$	$d.A$	record projection
	$\mid$	$v[e_1, \ldots, e_n]$	array indexing
Statement:			
s	$::=$	$d \mathrel{+}= e$	incremental update
	$\mid$	$d := e$	assignment
	$\mid$	**var** $v : t = e$	declaration
	$\mid$	**for** $v = e_1, e_2$ **do** s	iteration
	$\mid$	**for** v **in** e **do** s	traversal
	$\mid$	**while** (e) s	loop
	$\mid$	**if** (e) s_1 [**else** s_2]	conditional
	$\mid$	$\{\, s_1; \ldots; s_n \}$	statement block

Figure 3.1. Syntax of loop-based programs.

for-loop too becomes sequential and it is treated as a while-loop. Finally, a statement block contains a sequence of statements.

3.3 Recurrence and Restrictions for Parallelization

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. DIABLO can translate for-loops to equivalent DISC programs when these loops satisfy certain restrictions described in this section.

Our restrictions use the following definitions. For any statement s in a loop-based program, we define the following three sets of L-values (destinations): the readers $\mathcal{R}[\![s]\!]$, the writers $\mathcal{W}[\![s]\!]$, and the aggregators $\mathcal{A}[\![s]\!]$. The **readers** are the L-values read in s, the **writers** are the L-values written (but not incremented) in s, and the **aggregators** are the L-values incremented in s. For example, for the following statement:

$$V[W[i]] \mathrel{+}= n * C[i] * C[i+1]$$

where i is a loop index, the aggregators are $\mathcal{A}[\![s]\!] = \{V[W[i]]\}$, the readers are $\mathcal{R}[\![s]\!] = \{W[i], n, C[i], C[i+1]\}$, and the writers are $\mathcal{W}[\![s]\!] = \emptyset$. Two L-values d_1 and d_2 **overlap**, denoted by overlap(d_1, d_2), if they are the same variable, or they are equal to the projections $d_1'.A$ and $d_2'.A$ with overlap(d_1', d_2'), or they are array accesses over the same array name. The **context** of a statement s, context(s), is the set of outer loop indexes for all loops that enclose s. Note that, each for-loop must have a distinct loop index variable; if not, the duplicate loop index is replaced with a fresh variable. For an L-value d, indexes(d) is the set of loop indexes used in d.

An **affine** expression [41] takes the form

$$c_0 + c_1 * i_1 + \cdots + c_k * i_k$$

where $i_1, \ldots, i_k$ are loop indexes and $c_0, \ldots, c_k$ are constants. For an L-value d in a statement s, affine(d, s) is true if d is a variable, or a projection $d'.A$ with affine(d', s), or an array indexing $v[e_1, \ldots, e_n]$, where each index e_i is an affine expression and all loop indexes in context(s) are used in d. In other words, if affine(d, s) is true, then d is stored at different locations for different values of the loop indexes in context(s).

Definition 3.3.1 (Affine For-Loop). *A for-loop statement s is affine if s satisfies the following properties:*

1. *for any update $d := e$ in s,* affine(d, s)*;*
2. *there are no dependencies between any two statements s_1 and s_2 in s, that is, if there are no L-values $d_1 \in (\mathcal{A}[\![s_1]\!] \cup \mathcal{W}[\![s_1]\!])$ and $d_2 \in \mathcal{R}[\![s_2]\!]$ such that* overlap(d_1, d_2)*, with the following exceptions:*
   - *(a) if $d_1 \in \mathcal{W}[\![s_1]\!]$, $d_1 = d_2$, and s_1 precedes s_2;*
   - *(b) if $d_1 \in \mathcal{A}[\![s_1]\!]$, $d_1 = d_2$,* affine(d_2, s_2)*, s_1 precedes s_2, and* context(s_1) $\cap$ context(s_2) = indexes(d_1).

Restriction 1 indicates that the destination of any non-incremental update must be a different location at each loop iteration. If the update destination is an array access, the array indexes must be affine and completely cover all surrounding loop indexes. This restriction does not hold for incremental updates, which allow arbitrary array indexes in a destination as long as the array is not read in the same loop. Restriction 2 combined with exception (a) rejects any read and write on the same array in a loop except when the read is after the write and the read and write are at the same location ($d_1 = d_2$), which, based on Restriction 1, is a different location at each loop iteration. Exception (b) indicates that if we first increment and then read the same location, then these two operations must not be inside a for-loop whose loop index is not used in the destination. This is because the increment of the destination is done within the for-loops whose loop indexes are used in the destination and across the rest of the surrounding for-loops. For example, the following loop:

```
**for** $i = \dots$ **do** {
    **for** $j = \dots$ **do** { $V[i] \mathrel{+}= 1$ }; $W[i] := V[i]$
}
```

increments and reads $V[i]$. The contexts of the first and second updates are $\{i, j\}$ and $\{i\}$, respectively, and their intersection gives $\{i\}$, which is equal to the indexes of $V[i]$. If there were another statement $M[i, j] := V[i]$ inside the inner loop, this would violate Exception (b) since their context intersection would have been $\{i, j\}$, which is not equal to the indexes of $V[i]$.

For example, the incremental update:

```
**for** $i = \dots$ **do** $C[V[i].K] \mathrel{+}= V[i].D$
```

which counts all $V[i].D$ in groups that have the same key $V[i].K$, satisfies our restrictions since it increments but does not read C. On the other hand, some non-incremental updates may outright be rejected. For example, the loop:

```
**for** $i = \dots$ **do**
    $V[i] := (V[i-1] + V[i+1])/2$
```

will be rejected by Restriction 2 because V is both a reader and a writer.

To alleviate this problem, one may rewrite this loop as follows:

$$\begin{array}{l}\textbf{for } i = \ldots \textbf{ do } V'[i] := V[i];\\ \textbf{for } i = \ldots \textbf{ do } V[i] := (V'[i-1] + V'[i+1])/2\end{array}$$

which first stores V to V' and then reads V' to compute V. This program satisfies our restrictions but is not equivalent to the original program because it uses the previous values of V to compute the new ones. Another example is:

$$\textbf{for } i = \ldots \textbf{ do } \{\, n := V[i];\ W[i] := f(n)\,\}$$

which is also rejected because n is not affine as it does not cover the loop indexes (namely, i). To fix this problem, one may redefine n as a vector and rewrite the loop as:

$$\textbf{for } i = \ldots \textbf{ do } \{\, n[i] := V[i];\ W[i] := f(n[i])\,\}$$

Redefining variables by adding to them more array dimensions is currently done manually by a programmer, but we believe that it can be automated when a variable that violates our restrictions is detected.

3.4 Monoid Comprehensions

Our framework is based on monoid comprehensions, which are translated to a monoid algebra that consists of monoid homomorphisms. Monoids and monoid homomorphisms directly capture the most important property required for data parallelism, namely associativity. Given a type T, a binary operator $\otimes$ from (T, T) to T, and a value $1_\otimes$ of type T, the triple $(T, \otimes, 1_\otimes)$ is called a monoid if $\otimes$ is associative and has an identity $1_\otimes$,that is, $x \otimes 1_\otimes = 1_\otimes \otimes x = x$. Given that a monoid can be identified by its operation $\otimes$, it is simply referred to as $\otimes$. A container monoid $(C[T], \otimes, 1_\otimes, U_\otimes)$ over a parametric type $C[T]$ is a monoid equipped with a unit injection function $U_\otimes$ from T to $C[T]$. Data collections, such as lists, sets, and bags, can be captured as container monoids. For example, the unit function for the bag monoid $\uplus$ maps a value v of type T to the bag $\{v\}$ of type $\{T\}$ (ie, a bag$[T]$). A *homomorphism* H from a container monoid $(C[T], \otimes, 1_\otimes, U_\otimes)$ to a monoid $(S, \oplus, 1_\oplus)$ is defined as follows:

$$\begin{aligned} H(X \otimes Y) &= H(X) \oplus H(Y) \\ H(U_\otimes(x)) &= f(x) \\ H(1_\otimes) &= 1_\oplus \end{aligned}$$

for a function f from T to S. All operations in our monoid algebra are homomorphisms. Not all homomorphisms are well-behaved; some may actually lead to contradictions. In general, a homomorphism from a container monoid $\otimes$ to a monoid $\oplus$ is well-behaved if $\oplus$ satisfies all the laws that $\otimes$ does (the laws in our case are commutativity and idempotence). For example, converting a list to a bag is well-behaved, while converting a bag to a list and set cardinality are not.

If the target monoid of a homomorphism is also a container monoid, it is called a flatMap. When restricted to bags, flatMap(f, X) maps a bag X of type $\{T\}$ to a bag of type $\{S\}$ by applying the function f from type T to type $\{S\}$ to each element of X, yielding one bag for each element, and then by merging these bags to form a single bag of type $\{S\}$:

$$\begin{aligned} \text{flatMap}(f, X \uplus Y) &= \text{flatMap}(f, X) \uplus \text{flatMap}(f, Y) \\ \text{flatMap}(f, \{a\}) &= f(a) \\ \text{flatMap}(f, \{\}) &= \{\} \end{aligned}$$

More complex homomorphisms, such as groupBy, coGroup, and array scans, are defined using the following monoid $\Updownarrow_\oplus$, which depends on some monoid $\oplus$:

$$\begin{aligned} X \Updownarrow_\oplus Y = &\{(k, a \oplus b) | (k, a) \epsilon X, (k', b) \epsilon Y, k = k'\} \\ &\uplus \{(k, a) | (k, a) \epsilon X, \forall (k', b) \epsilon Y : k' \neq k\} \\ &\uplus \{(k, b) | (k, b) \epsilon Y, \forall (k', a) \epsilon X : k' \neq k\} \end{aligned}$$

The first term is a join between X and Y, the second is the subset of X not joined with Y, and the third is the subset of Y not joined with X. It returns an indexed set (a key-value map) in which the values that correspond to the same key are merged using the monoid $\oplus$. Given a bag X of type $\{(K, V)\}$, groupBy(X) groups the elements of X by their first component of type K (the group-by key) and returns a bag of type $\{(K, \{V\})\}$. It is a homomorphism over the monoid $\Updownarrow_{\uplus}$:

$$\begin{aligned}\text{groupBy}(X \uplus Y) &= \text{groupBy}(X) \Updownarrow_{\uplus} \text{groupBy}(Y)\\ \text{groupBy}(\{(k, v)\}) &= \{(k, \{v\})\}\\ \text{groupBy}(\{\}) &= \{\}\end{aligned}$$

Similarly, a coGroup(X, Y) between a bag X of type $\{(K, V)\}$ and a bag Y of type $\{(K, W)\}$ over their first component of type K (the join key) returns a bag of type $\{(K, (\{V\}, \{W\}))\}$. It is a homomorphism over the monoid $\Updownarrow_{\odot}$, where $(X, Y) \odot (X', Y')$=$(X \uplus X', Y \uplus Y')$:

$$\begin{aligned}\text{coGroup}((X, Y) \odot (X', Y')) &= \text{coGroup}(X \uplus X', Y \uplus Y')\\ &= \text{coGroup}(X, Y) \Updownarrow_{\odot} \text{coGroup}(X', Y')\end{aligned}$$

Finally, reduce$(\oplus, X)$ aggregates a collection X of type $\{V\}$ to a value of type V using the monoid $\oplus$, which is from (V, V) to V.

The target of our translations consists of monoid comprehensions, which are equivalent to the SQL select-from-where-group-by-having syntax. Monoid comprehensions are translated to a monoid algebra that consists of monoid homomorphisms.

A monoid comprehension has the following syntax:

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

where the expression e is the comprehension head and a qualifier q_i is defined as follows:

$$\begin{array}{lcll}\multicolumn{4}{l}{\textbf{Qualifier}:}\\ q & ::= & p \leftarrow e & \text{generator}\\ & \mid & \textbf{let}\ p = e & \text{let-binding}\\ & \mid & e & \text{condition}\\ & \mid & \textbf{group by}\ p\,[: e\,] & \text{group-by}\\ \multicolumn{4}{l}{\textbf{Pattern}:}\\ p & ::= & v & \text{pattern variable}\\ & \mid & (p_1, \ldots, p_n) & \text{tuple pattern.}\end{array}$$

The domain e of a generator $p \leftarrow e$ must be a bag. This generator draws elements from this bag and, each time, it binds the pattern p to an element. A condition qualifier e is an expression of type boolean. It is used for filtering out elements drawn by the generators. A let-binding **let** $p = e$ binds the pattern p to the result of e. A group-by qualifier uses a pattern p and an optional expression e. If e is missing, it is taken to be p. The group-by operation groups all the pattern variables in the same comprehension that are defined before the group-by (except the variables in p) by the value of e (the group-by key), so that all variable bindings that result to the same key value are grouped together. After the group-by, p is bound to a group-by key and each one of these pattern variables is lifted to a bag of values. The result of a comprehension $\{\, e \mid q_1,\ \ldots,\ q_n \,\}$ is a bag that contains all values of e derived from the variable bindings in the qualifiers.

Comprehensions can be translated to algebraic operations that resemble the bulk operations supported by many DISC systems, such as groupBy, join, map, and flatMap. We use $\overline{q}$ to represent the sequence of qualifiers $q_1,\ \ldots,\ q_n$, for $n \geq 0$. To translate a comprehension $\{\, e \mid \overline{q} \,\}$ to the algebra, the group-by qualifiers are first translated to groupBy operations from left to right. Given a bag X of type $\{(K, V)\}$, groupBy(X) groups the elements of X by their first component of type K (the group-by key) and returns a bag of type $\{(K, \{V\})\}$. Let $v_1, \ldots, v_n$ be the pattern variables in the sequence of qualifiers $\overline{q_1}$ that do not appear in the group-by pattern p, then we have:

$$\begin{aligned}
&\{\, e' \mid \overline{q_1},\ \textbf{group by}\, p : e,\ \overline{q_2} \,\} \\
&= \{\, e' \mid (p, s) \leftarrow \text{groupBy}(\{\, (e, (v_1, \ldots, v_n)) \mid \overline{q_1} \,\}), \\
&\qquad \forall i :\ \textbf{let}\ v_i = \{\, v_i \mid (v_1, \ldots, v_n) \leftarrow s \,\},\ \overline{q_2} \,\}
\end{aligned}$$

That is, for each pattern variable v_i, this rule embeds a let-binding so that this variable is lifted to a bag that contains all v_i values in the current group. Then, comprehensions without any group-by are translated to the algebra by translating the qualifiers from left to right:

$$\begin{aligned}
\{\, e' \mid p \leftarrow e,\ \overline{q} \,\} &= \text{flatMap}(\lambda p.\{\, e' \mid \overline{q} \,\},\ e) \\
\{\, e' \mid \textbf{let}\ p = e,\ \overline{q} \,\} &= \textbf{let}\ p = e\ \textbf{in}\ \{\, e' \mid \overline{q} \,\}
\end{aligned}$$

$$\{\!\!\{\, e' \mid e,\, \overline{q} \,\}\!\!\} = \textbf{if}\ e\ \textbf{then}\ \{\!\!\{\, e' \mid \overline{q} \,\}\!\!\}\ \textbf{else}\ \emptyset$$
$$\{\!\!\{\, e' \mid \,\}\!\!\} = \{e'\}$$

Given a function f that maps an element of type T to a bag of type $\{S\}$ and a bag X of type $\{T\}$, the operation $\text{flatMap}(f, X)$ maps the bag X to a bag of type $\{S\}$ by applying the function f to each element of X and unioning together the results. Although this translation generates nested flatMaps from join-like comprehensions, there is a general method for identifying all possible equi-joins from nested flatMaps, including joins across deeply nested comprehensions, and translating them to joins and coGroups.

Finally, nested comprehensions can be unnested by the following rule:

$$\begin{aligned} &\{\!\!\{\, e_1 \mid \overline{q_1},\, p \leftarrow \{\!\!\{\, e_2 \mid \overline{q_3} \,\}\!\!\},\, \overline{q_2} \,\}\!\!\} \\ &\qquad = \{\!\!\{\, e_1 \mid \overline{q_1},\, \overline{q_3},\, \textbf{let}\ p = e_2,\, \overline{q_2} \,\}\!\!\} \end{aligned} \tag{3.1}$$

for any sequence of qualifiers $\overline{q_1}$, $\overline{q_2}$, and $\overline{q_3}$. This rule can only apply if there is no group-by qualifier in $\overline{q_3}$ or when $\overline{q_1}$ is empty. It may require renaming the variables in $\{\!\!\{\, e_2 \mid \overline{q_3} \,\}\!\!\}$ to prevent variable capture.

3.5 DIABLO Framework

3.5.1 Array Representation

In DIABLO, a sparse array, such as a sparse vector or a matrix, is represented as 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. More specifically, a sparse vector of type $\text{vector}[T]$ is captured as a key-value map of type $\{(\text{long}, T)\}$, while a sparse matrix of type $\text{matrix}[T]$ is captured as a key-value map of type $\{((\text{long}, \text{long}), T)\}$.

Merging two compatible arrays is done with the array merging operation $\lhd$, defined as follows:

$$\begin{aligned} X \lhd Y = &\{\!\!\{\, (k, b) \mid (k, a) \leftarrow X,\, (k', b) \in Y,\, k = k' \,\}\!\!\} \\ &\uplus\ \{\!\!\{\, (k, a) \mid (k, a) \leftarrow X,\, k \notin \Pi_1(Y) \,\}\!\!\} \\ &\uplus\ \{\!\!\{\, (k, b) \mid (k, b) \leftarrow Y,\, k \notin \Pi_1(X) \,\}\!\!\} \end{aligned}$$

where $\Pi_1(X)$ returns the keys of X. That is, $X \lhd Y$ is the union of X and Y, except when there is $(k,x) \in X$ and $(k,y) \in Y$, in which case it chooses the latter value, (k,y). For example, $\{(3,10),(1,20)\} \lhd \{(1,30),(4,40)\}$ is equal to $\{(3,10),(1,30),(4,40)\}$. On Spark, the $\lhd$ operation can be implemented as a coGroup.

An update to a vector $V[e_1] := e_2$ is equivalent to the assignment $V := V \lhd \{(e_1,e_2)\}$. That is, the new value of V is the current vector V but with the value associated with the index e_1 (if any) replaced with e_2. Similarly, an update to a matrix $M[e_1,e_2] := e_3$ is equivalent to the assignment $M := M \lhd \{((e_1,e_2),e_3)\}$. Array indexing though is a little bit more complex because the indexed element may not exist in the sparse array. Instead of a value of type T, indexing over an array of T should return a bag of type $\{T\}$, which can be $\{v\}$ for some value v of type T, if the value exists, or $\emptyset$, if the value does not exist. Then, the vector indexing $V[e]$ is $\pmb{\{}\, v \mid (i,v) \leftarrow V,\, i = e \,\pmb{\}}$, which returns a bag of type $\{T\}$. Similarly, the matrix indexing $M[e_1,e_2]$ is $\pmb{\{}\, v \mid ((i,j),v) \leftarrow M,\, i = e_1,\, j = e_2 \,\pmb{\}}$. We are now ready to express any assignment that involves vectors and matrices. For example, consider the matrices R, M, and N of type matrix[float]. The assignment:

$$R[i,j] := M[i,k] * N[k,j] \tag{3.2}$$

is translated to the assignment:

$$\begin{aligned} R := R \lhd \pmb{\{}\, ((i,j), m*n) \mid\, & ((i,k),m) \leftarrow M, \\ & ((k',j),n) \leftarrow N,\, k = k' \,\pmb{\}} \end{aligned} \tag{3.3}$$

which uses a bag comprehension equivalent to a join between the matrices M and N. This assignment can be derived from assignment (3.2) using simple transformations. To understand these transformations, consider the product $X * Y$. Since both X and Y have been lifted to bags, because they may contain array accesses, this product must also be lifted to a comprehension that extracts the values of X and Y, if any, and returns their product:

$$X * Y = \pmb{\{}\, x * y \mid x \leftarrow X,\, y \leftarrow Y \,\pmb{\}}$$

Given that matrix accesses are expressed as:

$$\begin{aligned} M[i,k] &= \{\, m \mid ((I,J),m) \leftarrow M,\ I=i,\ J=k \,\} \\ N[k,j] &= \{\, n \mid ((I,J),n) \leftarrow N,\ I=k,\ J=j \,\} \end{aligned}$$

the product $M[i,k] * N[k,j]$ is equal to:

$$\begin{aligned} \{\, x*y \mid\ & x \leftarrow \{\, m \mid ((I,J),m) \leftarrow M,\ I=i,\ J=k \,\}, \\ & y \leftarrow \{\, n \mid ((I,J),n) \leftarrow N,\ I=k,\ J=j \,\} \,\} \end{aligned}$$

which is normalized as follows using Rule (3.1), after some variable renaming:

$$\begin{aligned} & \{\, x*y \mid ((I,J),m) \leftarrow M,\ I=i,\ J=k,\ \textbf{let}\ x=m, \\ & \qquad ((I',J'),n) \leftarrow N,\ I'=k,\ J'=j,\ \textbf{let}\ y=n \,\} \\ & = \{\, m*n \mid ((I,J),m) \leftarrow M,\ I=i,\ J=k, \\ & \qquad ((I',J'),n) \leftarrow N,\ I'=k,\ J'=j \,\} \end{aligned}$$

Lastly, since the value of e in the assignment $R[i,j] := e$ is lifted to a bag, this assignment is translated to $R := R \lhd \{\, ((i,j),v) \mid v \leftarrow e \,\}$, that is, R is augmented with an indexed set that results from accessing the lifted value of e. If e contains a value, the comprehension will return a singleton bag, which will replace $R[i,j]$ with that value. After substituting the value e with the term derived for $M[i,k] * N[k,j]$, we get an assignment equivalent to the assignment (3.3).

3.5.2 Handling Array Updates in a Loop

We now address the problem of translating array updates in a loop. We classify updates into two categories:

1. Incremental updates of the form $d := d \oplus e$, for some commutative operation $\oplus$, where d is an update destination, which is also repeated as the left operand of $\oplus$. It can also be written as $d \mathrel{\oplus=} e$. For example, $V[i] \mathrel{+=} 1$ increments $V[i]$ by 1.
2. All other updates of the form $d := e$.

Consider the following loop with a non-incremental update:

$$\textbf{for}\ i = 1, N\ \textbf{do}\ V[g(i)] := W[f(i)] \tag{3.4}$$

for some vectors V and W, and some terms $f(i)$ and $g(i)$ that depend on the index i. Our framework translates this loop to an update to the vector V, where all the elements of V are updated at once, in a parallel fashion:

$$\begin{aligned} V := V \lhd \{\, (g(i), v) \mid\, & i \leftarrow \text{range}(1, N), \\ & (k, v) \leftarrow V,\ k = f(i) \,\} \end{aligned} \tag{3.5}$$

But this expression may not produce the same vector V as the original loop if there are recurrences in the loop, such as, when the loop body is $V[i] := V[i-1]$. Furthermore, the join between range$(1, N)$ and W in (3.5) looks unnecessary. We will transform such joins to array traversals in Section 3.5.3.

In our framework, for-loops are embedded as generators inside the comprehensions that are associated with the loop assignments. Consider, for example, matrix copying:

$$\textbf{for } i = 1, 10 \textbf{ do for } j = 1, 20 \textbf{ do } M[i, j] := N[i, j]$$

Using the translation of the assignment $M[i, j] := N[i, j]$, the loop becomes:

$$\begin{aligned} & \textbf{for } i = 1, 10 \textbf{ do} \\ & \quad \textbf{for } j = 1, 20 \textbf{ do} \\ & \qquad M := M \lhd \{\, ((i, j), n) \mid ((I, J), n) \leftarrow N,\ I = i,\ J = j \,\} \end{aligned} \tag{3.6}$$

To parallelize this loop, we embed the for-loops inside the comprehension as generators:

$$\begin{aligned} M := M \lhd \{\, ((i, j), n) \mid\, & i \leftarrow \text{range}(1, 10),\ j \leftarrow \text{range}(1, 20), \\ & ((I, J), n) \leftarrow N,\ I = i,\ J = j \,\} \end{aligned} \tag{3.7}$$

Notice the difference between the loop (3.6) and the assignment (3.7). The former will do 10*20 updates to M while the latter will only do one bulk update that will replace all $M[i, j]$ with $N[i, j]$ at once. This transformation can only apply when there are no recurrences across iterations.

3.5.3 Eliminating Loop Iterations

Before we present the details of program translation, we address the problem of eliminating index iterations, such as range(1, N) in assignment (3.5), and range(1, 10) and range(1, 20) in assignment (3.7). If there is a right inverse F of f such that $f(F(k)) = k$, then the assignment (3.5) is optimized to:

$$V := V \lhd \{\, (g(F(k)), v) \mid (k, v) \leftarrow W,\ \text{inRange}(F(k), 1, N) \,\} \tag{3.8}$$

where the predicate inRange$(F(k), 1, N)$ returns true if $F(k)$ is within the range $[1, N]$. Given that the right-hand side of an update may involve multiple array accesses, we can choose one whose index term can be inverted. For example, for $V[i{-}1]$, the inverse of $k = i - 1$ is $i = k + 1$. In the case where no such inverse can be derived, the range iteration simply remains as is. One such example is the loop **for** $i = 1, N$ **do** $V[i] := 0$, which is translated to $V := V \lhd \{\, (i, 0) \mid i \leftarrow \text{range}(1, N) \,\}$.

3.5.4 Handling Incremental Updates

There is an important class of recurrences in loops that can be parallelized using group-by and aggregation. Consider, for example, the following loop with an incremental update:

$$\textbf{for } i = 1, N \textbf{ do } V[g(i)] \mathrel{+}= W[i] \tag{3.9}$$

Let's say, for example, that there are 3 indexes overall, i_1, i_2, and i_3, that have the same image under g, ie, $k = g(i_1) = g(i_2) = g(i_3)$. Then, $V[k]$ must be set to $V[k] + W[i_1] + W[i_2] + W[i_3]$. In general, we need to bring together all values of W whose indexes have the same image under g. That is, we need to group by $g(i)$. Hence, the loop can be translated to a comprehension with a group-by:

$$\begin{aligned} V := V \lhd \{\, (k, v + (+/w)) \mid\ & (i, w) \leftarrow W,\ \text{inRange}(i, 1, N), \\ & \textbf{group by } k : g(i),\ (j, v) \leftarrow V,\ j = k \,\} \end{aligned}$$

which groups W by the destination index $g(i)$ and, for each group, it calculates the aggregation $+/w$ of all values $w = W[i]$ with the same $g(i)$ value, but also adds the original value $v = V[g(i)]$ before the group-by.

If the destination of the incremental update is a variable, such as in $n \mathrel{+}= W[i]$, then the group-by is over ( ), since there are no indexes used in n:

$$n := \{\!\!\{\, n + (+/w) \mid (i,w) \leftarrow W,\ \text{inRange}(i,1,N),\ \textbf{group by}\ k : (\,) \,\}\!\!\}$$

This group-by can be eliminated because it forms a single group; in which case the variable w is lifted to a bag that contains all the values of W:

$$n := \{\!\!\{\, n + (+/\{\!\!\{\, w \mid (i,w) \leftarrow W,\ \text{inRange}(i,1,N) \,\}\!\!\}) \,\}\!\!\}$$

3.5.5 Program Translation

DIABLO translates a loop-based program in pieces, in a bottom-up fashion over the abstract syntax tree (AST) representation of the program, by translating every AST node to a comprehension. The target of DIABLO is a list of statements, where a statement c has the following syntax:

Target Code:

c	$::=$	$v := e$	assignment
	$\mid$	$\text{while}(e, c)$	loop
	$\mid$	$[\,c_1, \ldots, c_n\,]$	code block

In the target code, an assignment to a variable v of type t gets a value e of type $\{t\}$. An assignment to an array is done in bulk, by replacing the entire array with a new one. The while-loop corresponds to the while statement in Figure 3.1; it repeats the code c in its body while the condition e is true. Finally, a code block is like a block of statements that need to be evaluated in order.

3.5.6 Examples of Program Translation

Consider, for example, the incremental update $A[e] \mathrel{+}= v$ in a loop, for a sparse vector A. The cumulative effects of all these updates throughout the loop can be performed 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 can be replaced with these new values. For instance, assuming that the values of C were zero before the loop, the following program:

```
for $i = 0, 9$ do
  $C[A[i].K]\ \mathrel{+}= A[i].V)$
```

can be evaluated in bulk by grouping the elements $A[i]$ of the vector A by $A[i].K$ (the group-by key), and summing up all the values $A[i].V$ associated with each different group-by key. Then the resulting key-sum pairs are the new values for the vector C.

Our framework translates the previous loop-based program to the following bulk assignment that calculates all the values of C using a bag comprehension that returns a bag of index-value pairs:

$$C \ := \ \{\!\!\{\, (k, +/v) \mid (i,k,v) \leftarrow A,\ \mathbf{group\ by}\ k \,\}\!\!\}$$

A group-by operation in a comprehension lifts each pattern variable defined before the group-by (except the group-by keys) from some type t to a bag of t, indicating that each such variable must now contain all the values associated with the same group-by key value. Consequently, after we group by k, the variable v is lifted to a bag of values, one bag for each different k. In the comprehension result, the aggregation $+/v$ sums up all the values in the bag v, thus deriving the new values of C for each index k.

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

```
for $i = 0, d-1$ do
  for $j = 0, d-1$ do {
    $R[i,j] = 0.0;$
    for $k = 0, d-1$ do {
      $R[i,j]\ \mathrel{+}= M[i,k] * N[k,j]);$
  }
}
```

A sparse matrix M can be represented as a bag of tuples (i,j,v) such that $v = M_{ij}$. This program too can be translated to a single assignment that replaces the entire content of the matrix R with a new content, which is calculated using bulk relational operations.

DIABLO translates the loop-based program for matrix multiplication to the following assignment:

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

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

3.6 Conclusion

In DIABLO, the array-based loop programs are translated to comprehensions and these comprehensions are translated to Spark Core API programs. However, in the next chapters we will see that instead of translating to Core API programs, we can translate them to SQL programs. We will also see that SQL programs perform significantly faster than Core API programs.

CHAPTER 4

SQLgen: Translation to SQL

4.1 Introduction

In this chapter, we discuss our framework SQLgen [18]. SQLgen translates array-based loops to a declarative domain-specific language (DSL), more specifically, Spark SQL [14]. At first, loops are translated to equivalent monoid comprehensions [13] and then to Spark SQL. We chose Spark SQL as our target language since it is in general more efficient than the Spark Core API because it takes advantage of existing extensive work on SQL optimization for relational database systems. In Spark SQL, datasets are expressed as DataFrames, which are distributed collection of data, organized into named columns. The schema of a DataFrame must be known, while DataFrame computations are done on columns of named and typed values. Operations from the Spark Core API, on the other hand, are higher-order with arguments that are functions coded in the host language and compiled to bytecode, which cannot be analyzed during program optimization. Hence, Spark SQL can find and apply optimizations that are very hard to detect automatically when the same program is written in the Spark Core API. Spark DataFrames have two specialized back-end components, Catalyst (the query optimizer) and Tungsten (the off-heap serializer), which facilitate optimized performance on other Spark components, such as MLlib, that are primarily based on DataFrame API. Catalyst supports both rule-based and cost-based optimization. For example, it can optimize a query by reordering the operations, such as pushing a filter operation before a join operation. The operations in Spark SQL reduce the amount of data sent over the network by selecting only the relevant columns and partitions from the dataset necessary for the computation.

Consequently, loops translated to Spark SQL performs better than loops translated to Spark RDD operations, as it was done by earlier frameworks.

The contributions of this work are summarized as follows:

- We present a novel framework 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 our system on a variety of data analysis and machine learning programs and we compare its performance relative to DIABLO and to hand-written Spark programs expressed in the Spark Core API (RDDs) and in Spark SQL. Our performance results indicate that, for these programs, SQLgen outperforms the equivalent DIABLO and the hand-written Spark Core programs, giving performance close to that of hand-written programs in Spark SQL.

4.2 The SQLgen Framework

In our earlier work DIABLO, loop-based programs are first translated to monoid comprehensions, then to the monoid algebra, and finally to Java byte code that calls the Spark Core API. The goal of 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.

The syntax of a monoid comprehension is as follows:

$$\begin{array}{rcll} e & ::= & \{\, e \mid q_1,\ \ldots,\ q_n \,\} & \text{comprehension} \\ & \mid & \oplus/e & \text{reduction} \end{array}$$

where in the comprehension, the comprehension head e is an expression and q_i is a qualifier, and $\oplus/e$ is a total aggregation over a comprehension e. The output of our translations is a list of statements c that have the following syntax:

Target Code:

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

where s is the Spark SQL generated from a comprehension, which is assigned to a variable v. Multiple assignments can be grouped in a code block c.

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

Vectors and matrices in DIABLO are translated to DataFrames in Spark SQL. Basically, a sparse array is translated to a relational table with two columns: the first column is a tuple that contains the index elements and the second column is the element value, which can be a primitive type or a composite type, such as StructType, which is represented by a case class in Scala. For example, a vector V of type $\{(\text{Long}, \text{Double})\}$ in DIABLO is mapped to a table V of schema (_1 : Long, _2 : Double), while a matrix M of type $\{((\text{Long}, \text{Long}), \text{Double})\}$ in DIABLO is mapped to a table M of schema (_1 : Struct (_1 : Long, _2 : Long), _2 : 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.

The syntax of Spark SQL though is very limited compared to the full SQL syntax. The syntax of Spark SQL generated by our translator is as follows:

select	*expression* [**as** *alias*]
	[, *expression* [**as** *alias*], ...]
from	(*relation* [*alias*] |
	relation [*alias*] **join** *relation* [*alias*] **on**
	boolean_expression [**join** *relation* [*alias*]
	on *boolean_expression* ...])
[**where**	*boolean_expression*
	[**and** *boolean_expression* **and** ...]]
[**group by**	*expression* [, *expression*, ...]]

where alternatives are shown in parenthesis (. . . | . . . | . . .) and optional parts in square brackets [. . .]. Expressions in the *select* clause contain column names and may contain 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 only inner joins and no subqueries.

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

```
**for** $i = 0, d-1$ **do**
  **for** $j = 0, d-1$ **do** {
    $R[i,j] = 0.0;$
    **for** $k = 0, d-1$ **do**
      $R[i,j]$ += $M[i,k] * N[k,j]);$
  }
}
```

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

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

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

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

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

Our framework, called SQLgen, has been implemented in Scala using compile-time reflection. The source language used to expressed loops with array operations is the same proof-of-concept language defined in DIABLO, 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.

4.2.1 Translation to SQL

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

4.2.1.1 Pattern Compilation

Pattern variables in a comprehension are defined in the generators and are 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 $p = x$:

$$\mathcal{C}[\![(p_1, \dots, p_n)]\!]_x = \mathcal{C}[\![p_1]\!]_{x.\_1} \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} \mathcal{C}[\![p_n]\!]_{x.\_n} \tag{4.1}$$

$$\mathcal{C}[\![v]\!]_x = [v \rightarrow x] \tag{4.2}$$

where $\mathbin{+\!\!+}$ merges bindings.

For our example, after applying (4.1) on $((i,j),v)$ we get the bindings:

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

Then, applying (4.2) we get, $\mathcal{C}[\![((i,j),v)]\!]_x = [i \rightarrow x.\_1.\_1,\ j \rightarrow x.\_1.\_2, v \rightarrow x.\_2]$. Before the translation to SQL, we eliminate the patterns from a comprehension. For each generator $p \leftarrow e'$, and any sequences of qualifiers $\overline{q_1}$ and $\overline{q_2}$ in a comprehension, we do:

$$\{\, e \mid \overline{q_1},\, p \leftarrow e', \overline{q_2} \,\} = \{\, \rho(e) \mid \overline{q_1},\, x \leftarrow e', \rho(\overline{q_2}) \,\} \tag{4.3}$$

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:

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

4.2.1.2 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:

$$\begin{aligned}\mathcal{SQL}[\![\{\, h \mid \overline{q} \,\}]\!] = &\ \mathbf{select}\, h \\ &\ \mathbf{from}\, \mathcal{Q}[\![\overline{q}]\!] \\ &\ \mathbf{where}\, \mathcal{P}[\![\overline{q}]\!] \\ &\ \mathbf{group\ by}\, \mathcal{G}[\![\overline{q}]\!]\end{aligned} \tag{4.4}$$

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:

$$\begin{aligned}&\mathcal{Q}[\![\overline{q_1},\, v_1 \leftarrow e_1,\, \overline{q_2},\, v_2 \leftarrow e_2,\, \overline{q_3},\, e_3 = e_4,\, \overline{q_4}]\!] = \\ &\quad \mathcal{Q}[\![\overline{q_1},\, (v_1,v_2) \leftarrow (e_1\,\textbf{join}\,e_2\,\textbf{on}\,e_3 = e_4),\, \overline{q_2},\, \overline{q_3},\, \overline{q_4}]\!]\end{aligned} \tag{4.5}$$

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:

$$\mathcal{Q}[\![v \leftarrow e,\, \overline{q}]\!] = \; e\,v,\; \mathcal{Q}[\![\overline{q}]\!] \tag{4.6}$$

$$\mathcal{Q}[\![e,\, \overline{q}]\!] = \; \mathcal{Q}[\![\overline{q}]\!] \tag{4.7}$$

$$\mathcal{Q}[\![\;]\!] = \; \emptyset \tag{4.8}$$

where (4.6) 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:

$$\mathcal{P}[\![e,\, \overline{q}]\!] = \; e\,\textbf{and}\,\mathcal{P}[\![\overline{q}]\!] \tag{4.9}$$

$$\mathcal{P}[\![p \leftarrow e,\, \overline{q}]\!] = \; \mathcal{P}[\![\overline{q}]\!] \tag{4.10}$$

$$\mathcal{P}[\![\;]\!] = \; \emptyset \tag{4.11}$$

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:

$$\mathcal{G}[\![\textbf{group by}\, p,\, \overline{q}]\!] = p \tag{4.12}$$

$$\mathcal{G}[\![p \leftarrow e,\, \overline{q}]\!] = \mathcal{G}[\![\overline{q}]\!] \tag{4.13}$$

$$\mathcal{G}[\![e,\, \overline{q}]\!] = \mathcal{G}[\![\overline{q}]\!] \tag{4.14}$$

$$\mathcal{G}[\![\ ]\!] = \emptyset \tag{4.15}$$

4.2.2 Examples of Program Translation

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

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

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

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

where the predicate inRange$(x._1, 1, 10)$ returns true if $1 \leq x.\_1 \leq 10$. Before the translation to SQL, we eliminate the patterns from the comprehension. The only pattern in the comprehension is (i, v), which is replaced with a fresh variable x. Then, using (4.1) and (4.2) we get $\mathcal{C}[\![(i, v)]\!]_x = [i \rightarrow x.\_1, v \rightarrow x.\_2]$. Therefore, using (4.3), the comprehension is transformed to:

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

where pattern variable i is replaced with $x.\_1$ and v is replaced with $x.\_2$. Next, we translate this transformed comprehension to an equivalent SQL query. To generate the equivalent SQL query, we use (4.4) to translate the transformed comprehension to SQL.

The semantic functions $\mathcal{Q}, \mathcal{P}$, and $\mathcal{G}$ take the qualifiers of the transformed comprehension as their input and translate to the equivalent SQL program:

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

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:

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

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 (4.1) and (4.2), 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 (4.3):

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

Then, we apply (4.4) where the header of the comprehension $(y.\_2, +/x.\_2)$ is translated to $y.\_2$, $sum(x.\_2)$.

Then, using (4.5)-(4.8), the semantic function $\mathcal{Q}$ is applied to the transformed comprehension in the *from* clause:

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

Using (4.9)-(4.11) we get:

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

Then, using (4.12), we get:

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

$$
\begin{array}{rll}
= & \textbf{select} & y.\_2,\ sum(x.\_2) \\
 & \textbf{from} & V\,x\,\textbf{join}\,K\,y\,\textbf{on}\,x.\_1 = y.\_1 \\
 & \textbf{where} & 1 \leq x.\_1\,\textbf{and}\,x.\_1 \leq 10 \\
 & \textbf{group by} & y.\_2
\end{array}
$$

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:

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

The comprehension of matrix multiplication is as follows:

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

To keep this example simple, we omit the inRange qualifiers. In the comprehension above, the patterns $((i, k), m)$ and $((k', j), n)$ are replaced with fresh variables x and y. Then, after applying (4.1) and (4.2), 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 (4.3) and the comprehension is transformed to:

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

Then, we can get the equivalent SQL from (4.4). In the *select* clause, we get, $x.\_1.\_1, y.\_1.\_2, sum(x.\_2 * y.\_2)$ where v is substituted by the let-binding expression. Next, the semantic function $\mathcal{Q}$ is applied to the transformed comprehension in the *from* clause. Using (4.5-4.8), we get:

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

Next, we apply the semantic function $\mathcal{P}$ to the transformed comprehension in the *where* clause, which is not shown here. Then, we apply semantic function $\mathcal{G}$ to the transformed comprehension in the *group by* clause. Using (4.12), we get:

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

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

4.3 Performance Evaluation

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

We evaluated the performance of SQLgen on 12 different programs and compared it with equivalent DIABLO programs, hand-written RDD-based Spark programs, and Spark SQL programs as shown in Figure 4.1. The platform used in our experiments is the XSEDE Comet cloud computing infrastructure at SDSC (San Diego Supercomputer Center). Each program was run on a cluster of 10 nodes where

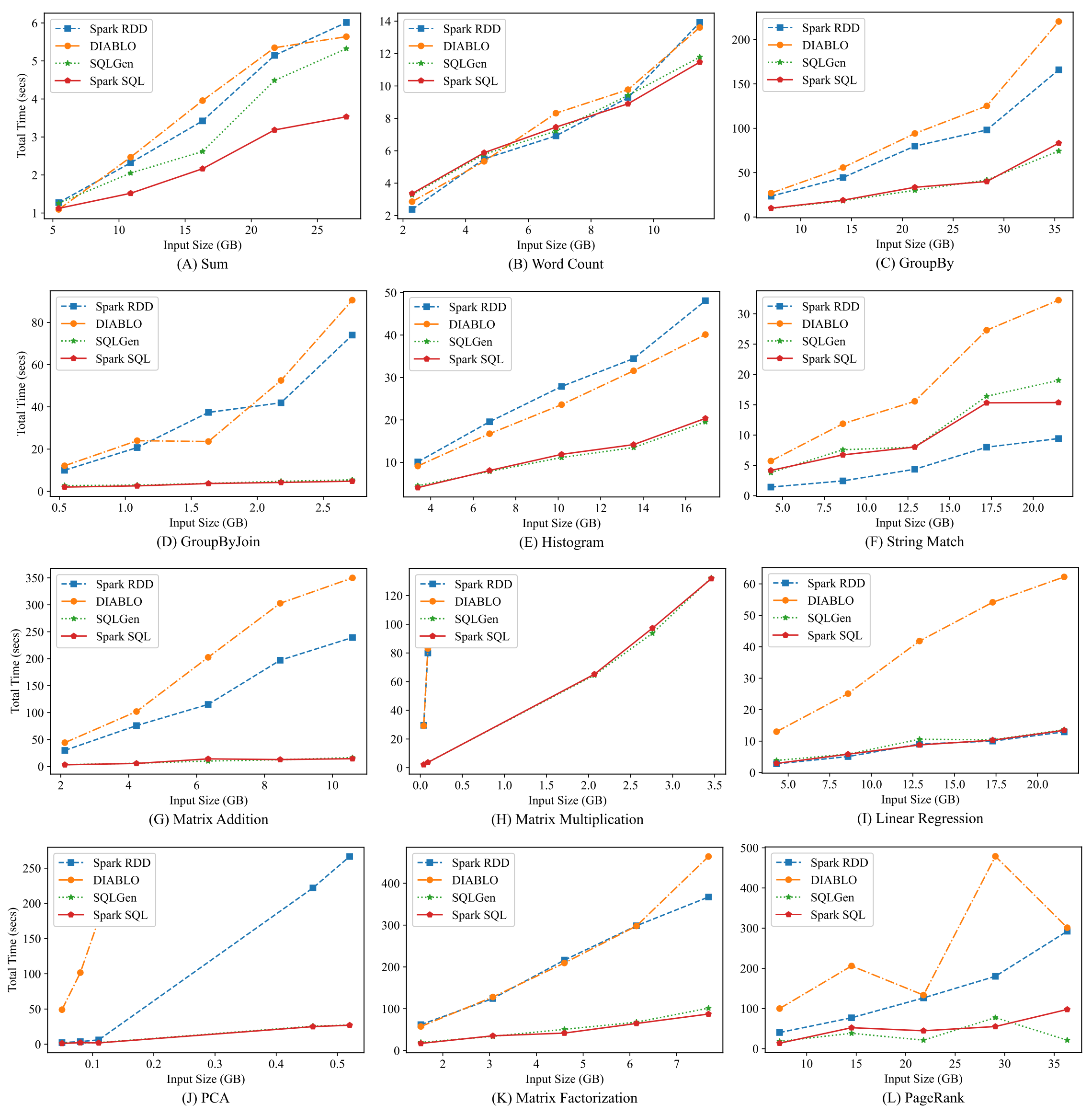

Figure 4.1. Performance of SQLgen relative to DIABLO, hand-written Spark RDD, and Spark SQL.

each node is equipped with 24 core Xeon E5-2680v3 processor with 2.5GHz clock speed, 128GB RAM and 320GB SSD. The programs were run on Apache Spark 2.2.0 on Apache Hadoop 2.6.0. Each Spark executor on Spark was configured to have 4 cores and 23 GB RAM. So, there were $24/4 = 6$ executors per node, giving a total of 60 executors, from which 2 were reserved. The input data for each program were randomly generated. Each program was evaluated 4 times on each of 5 different sizes of datasets. From the 4 iterations over each dataset, the results from the first iteration were ignored to avoid the possible overhead due to the JIT warm-up time. So, each data point in the plots represents the mean time on the rest of the 3 iterations. The input dataset size was calculated by multiplying the length of the dataset by the size of each serialized dataset element. For example, the size of a serialized RDD of the key-value pair 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. 2.

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 the average. The largest dataset used had 2×10^8 elements and size 35.39 GB. SQLgen is approximately 3 times faster than DIABLO. GroupByJoin joins two datasets, groups the result by some component, and returns the sum of another component in each group. The join keys of both datasets were random long integers with 10 duplicates on the average. The largest datasets had 2×10^7 elements and size 2.72 GB each. For this experiment, SQLgen is up to 18 times faster than 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, SQLgen is up to twice as fast as 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 performance of the SQLgen program was approximately 1.5 times faster than 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. Multiplication on DIABLO and the hand-written RDD-based program was very slow, so it was only run on 2 datasets. For both addition and multiplication, SQLgen is approximately 20 times faster than DIABLO and 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, SQLgen is approximately 4.5 times faster than 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 SQLgen performed approximately 78.5 times faster than DIABLO. For the next two datasets, where the number of columns was increased to 400, SQLgen performed approximately 10 times faster than the hand-written RDD-based program.

Matrix Factorization (K): This program is one iteration of matrix factorization using gradient descent [42]. 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, SQLgen is approximately 4.5 times faster than 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 [43]. The graphs used in our experiments were synthetic data generated by the RMAT (Recursive MATrix) Graph Generator [44] 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, SQLgen is more than 5 times faster than DIABLO.

4.4 Conclusion

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. 2, the SQLgen lines coincide with that of the hand-written Spark SQL lines, which implies that the derived SQL queries from SQLgen are equivalent (although not equal) to the hand-written SQL queries. On the other hand, compared to hand-written RDD-based programs and the programs generated by DIABLO, SQLgen is significantly faster except for the simple programs Sum and Word Count, where the performance of all four programs was similar.

CHAPTER 5

OSQLgen: Translation to SQL on Block Arrays

5.1 Introduction

Over the past few decades, researchers have expressed many algorithms, especially 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 for all-pairs shortest path problem 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 5.1. 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 5.5. 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 improve the framework that we have developed in our earlier work SQLgen [18], by translating array-based loop programs that are equivalent to semirings to programs on block arrays expressed in Spark SQL [14]. 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. The contributions of this work are summarized as follows:

- We present 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.
- OSQLgen can also translate basic linear algebra operations such as matrix multiplication, matrix transpose, etc. on coordinate arrays to more optimal block arrays.
- We compare the performance of our system on real-world problems relative to GraphX, GraphFrames, MLlib, and hand-written Spark SQL programs on coordinate and block arrays. Our performance results indicate that, for these programs, OSQLgen outperforms GraphX, GraphFrames, MLlib and Spark SQL programs on coordinate arrays giving performance close to that of hand-written Spark SQL programs on block arrays.

5.2 Background

OSQLgen is built on top of our earlier work SQLgen described in chapter 4. In OSQLgen, the input programs are same proof-of-concept language used in SQLgen, and DIABLO.

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

Table 5.1. Semirings for Graph Algorithms

5.2.1 Semiring and Graph Algorithms

The formal basis of our framework is the monoid comprehension. In abstract algebra, a monoid is an algebraic structure equipped with a single associative binary operation and a single identity element. More formally, given a set S, a binary operator $\oplus$ from $S \times S$ to S, and an element $e \in S$, the structure $(S, \oplus, e)$ is called a monoid if $\oplus$ is associative and has an identity e:

$$\begin{aligned} x \oplus (y \oplus z) = (x \oplus y) \oplus z \quad & \text{for all} \quad x, y, z \in S \\ x \oplus e = x = e \oplus x \qquad & \text{for all} \quad x \in S \end{aligned}$$

Monoids may satisfy additional algebraic laws. The monoid $(S, \oplus, e)$ is commutative if $x \oplus y = y \oplus x$, for all $x, y \in S$. A semiring$(S, \oplus, \otimes, \overline{1}, \overline{0})$ is an algebraic structure equipped with two monoids: additive$(\oplus, \overline{0}) : S \times S \to S$ and multiplicative$(\otimes, \overline{1}) : S \times S \to S$, where the additive monoid is commutative and the multiplicative monoid distributes over the additive monoid. We use the notation $\oplus(A \otimes B)$ to represent semiring operation between the sets A and B over S. This notation is used to represent algorithms that follow the properties of a semiring. For example, multiplication between matrices A and B is represented in terms of the semiring $C = +(A * B)$. Many graph algorithms can also be represented in terms of semirings. For example, the computation of all-pairs shortest path can be represented as $C = min(A + B)$. On the other hand, the product of two matrices can be computed by operating on the submatrices after partitioning the matrices into blocks of submatrices [45]. We denote the multiplication of two block matrices as $Cb = +_b(Ab *_b Bb)$ where $*_b$ represents multiplication between blocks and $+_b$ represents addition of blocks. In a distributed system such as Spark, multiplication of matrices can be done by first partitioning the input matrices conformably and the partitions/blocks are small enough to fit in each worker's memory. Then the blocks are multiplied in each worker node to compute the partial results and then added together to get the final result. Here, the order of the operands in the worker nodes to multiply the blocks ($*_b$) is set by the input program. Similarly, graph algorithms that can be represented as $\oplus_b(Ab \otimes_b Bb)$ can also be implemented on Spark, where $\otimes_b$ represents multiplicative operation on blocks in worker nodes and $\oplus_b$ represents a additive operation on blocks across the worker nodes.

5.3 The OSQLgen Framework

Similar to SQLgen the input to our translation system is monoid comprehensions. The syntax of a monoid comprehension is as follows:

$$\begin{array}{rcll} e & ::= & \{\, e \mid q_1,\ \ldots,\ q_n \,\} & \text{comprehension} \\ & \mid & \oplus/e & \text{reduction} \end{array}$$

where in the comprehension, the comprehension head e is an expression, q_i is a qualifier, and $\oplus/e$ is a total aggregation over an expression e that reduces the results of e using the monoid $\oplus$.

The output OSQLgen is a list of statements c that have syntax:

$$\begin{array}{rcll} c & ::= & v := e & \text{assignment} \\ & \mid & \text{while}(e, c) & \text{loop} \\ & \mid & \{c_1; \ldots; c_n\} & \text{code block} \end{array}$$

Here, in $v := e$, a comprehension e is translated to a Spark SQL query. Multiple assignments can be grouped in a code block using $\{c_1; \ldots; c_n\}$.

The syntax of Spark SQL on block arrays generated by OSQLgen when the input program is a semiring is as follows:

$$\begin{array}{ll} \textbf{select} & A.\_1.\_1,\ B.\_1.\_2\ \textbf{as}\ \_1, +_b(\mathit{collect\_list}(*_b(A.\_2, B.\_2)))\ \textbf{as}\ \_2 \\ \textbf{from} & \mathit{relation}\,A\,\textbf{join}\,\mathit{relation}\,B\,\textbf{on}\,A.\_1.\_2 = B.\_1.\_1 \\ \textbf{group by} & A.\_1.\_1,\ B.\_1.\_2 \end{array}$$

Loop-based programs are translated to comprehension using a method described in our earlier work [13]. Then, we translate the resulting comprehension in two steps: pattern compilation and comprehension translation. In the pattern compilation step, we transform the comprehension by eliminating the patterns from the comprehension. Then, in the comprehension translation step, we translate the transformed comprehension to Spark SQL program using a few semantic functions.

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 $+\infty$.

```
**var** $R$ : matrix[Double] = matrix();
**for** $i = 0, n-1$ **do**
  **for** $j = 0, n-1$ **do** {
    **for** $k = 0, n-1$ **do** {
      $R[i,j] := min(R[i,j],\ G[i,k] + G[k,j])$; //*update*
    }
    $G[i,j] := R[i,j]$; //*assignment*
  }
```

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. In general, a program that is equivalent to a semiring on array A and B has the following structure:

```
**var** $C$ : matrix[Double] = matrix();
**for** $i = 0, n-1$ **do**
  **for** $j = 0, n-1$ **do** {
    **for** $k = 0, n-1$ **do** {
      $C[i,j] := C[i,j] \oplus (A[i,k] \otimes B[k,j])$; //*update*
    }
    $A[i,j] := C[i,j]$; //*assignment*
  }
```

Here, in the update step, the additive monoid $\oplus$ is applied to the result of the multiplicative monoid $\otimes$ applied to the arrays A and B. Our example program to compute all-pairs shortest path has the same structure since min, and $+$ follow the properties of $\oplus$ and $\otimes$, respectively and the arrays A and B are the transition matrix G.

SQLgen translates this all-pairs shortest program to a bulk assignment to array R that calculates all the values of R in one shot using a bag comprehension that returns the new content of R. The cumulative effects of all the updates to the array R throughout the iterations are performed in bulk by grouping the values across the iterations by the indices and then by summing up these values for each group. Then the array R is replaced with these new values. The array R then replaces the array G. More specifically, the above program is translated to a comprehension as follows:

$$\begin{aligned}
R &:= \{\!\!\{\, ((i,j), min/v) \mid ((i,k),m) \leftarrow G, ((k',j),n) \leftarrow G,\ k = k', \\
&\qquad\qquad\qquad\qquad \mathbf{let}\, v = m + n,\ \mathbf{group\ by}\ (i,j) \,\}\!\!\} \\
G &:= R
\end{aligned}$$

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 ((I:Int, J:Int), V: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:

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

Then, from this block comprehension, we generate the following SQL program which operates on the block matrices:

$$\begin{aligned} Rb := \textbf{select}\quad & struct(\mathrm{X}.\_1.\_1,\ \mathrm{Y}.\_1.\_2),\ tile\_min(\\ & \quad collect\_list(min\_tiles(\mathrm{X}.\_2, \mathrm{Y}.\_2))) \\ \textbf{from}\quad & \mathrm{Gb}\, X \ \textbf{join}\ \mathrm{Gb}\, Y\ \textbf{on}\ \mathrm{X}.\_1.\_2 = \mathrm{Y}.\_1.\_1 \\ \textbf{group by}\quad & \mathrm{X}.\_1.\_1,\ \mathrm{Y}.\_1.\_2 \end{aligned}$$

Here, min_tiles is a user-defined function that computes min_b the minimum distance values between the pair of blocks of arrays, and $tile\_min$ is a user-defined function that computes $+_b$ the minimum distance values of the blocks after aggregating the blocks using the built-in function $collect\_list$.

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

5.3.1 OSQLgen Storage System

Vectors and matrices in DIABLO are translated to DataFrames in Spark SQL. Basically, a sparse array is translated to a relational table with two columns: the first column is a nested struct column of StructType that contains the block 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 $\{(\mathrm{Long}, \mathrm{Double})\}$ in DIABLO is mapped to a table V of schema (_1 : Int, _2 : Vector) where _1 represents the block index. The vector can be either dense or sparse. A dense vector is represented as a vector of type $\{(\mathrm{values} : \mathrm{Array[Double]})\}$. On the other hand, a sparse vector is represented as a vector of type $\{(\mathrm{size} : \mathrm{Int}, \mathrm{indices} : \mathrm{Array[Int]}, \mathrm{values} : \mathrm{Array[Double]})\}$. For example, a vector $(1.0, 0.0, 2.0)$ can be represented in dense format as $[1.0, 0.0, 2.0]$ or in sparse format as $(3, [0, 2], [1.0, 2.0])$, where 3 is the size of the vector.

On the other hand, a matrix M of type $\{((\mathrm{Long}, \mathrm{Long}), \mathrm{Double})\}$ in DIABLO is mapped to a table M of schema (_1 : Struct (_1 : Int, _2 : Int), _2 : Matrix), where the index column is nested with the block row index column referred to as _1._1 and the block column index referred to as _1._2. The matrix can be either dense or sparse. The dense matrix is represented as $\{(\text{values} : \text{Array[Double]})\}$. The entry values of the matrix are stored as of type Array[Double] in a column-major order. For example, the following dense matrix:

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

is stored in values array 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 $\{(\text{colPtrs} : \text{Array[Int]}), \text{rowIndices} : \text{Array[Int]}), \text{values} : \text{Array[Double]})\}$ where the values array contains all the non-zero entries of the matrix in a column-major order, rowIndices array contains the row indices of the values in the values array, and finally 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 as values array: $[1.0, 2.0, 3.0]$, rowIndices array $=[0, 2, 1]$, and colPointers array$=[0, 2, 3]$.

5.3.1.1 Mapping to Block Matrix

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

$$\{\, ((I, J), \text{matrix}(m)) \mid ((i, j), v) \leftarrow A,\ \textbf{let}\, I = i/N,\ \textbf{let}\, J = j/N, \\ \textbf{let}\, m = ((i\%N, j\%N), v),\ \textbf{group by}\ (I, J) \,\}$$

In the head of this comprehension, we have the indices of the block and a call to a function matrix(m).

Function matrix(m) takes a list of coordinates of form $((i, j), v)$ and returns an array representing a block. The function is defined 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;
}
```

If the generator B is a vector we use mapping to transform a coordinate vector to a block vector:

$$\{\, (I, \text{vector}(m)) \mid (i, v) \leftarrow B,\ \textbf{let}\, I = i/N,\ \textbf{let}\, m = (i\%N, v),\ \textbf{group by}\ I \,\}$$

The function vector(m) takes a list of coordinates of form (i, v) and returns an array representing a block. The 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;
}
```

5.3.2 Translation of Linear Algebra operations to SQL on Block Arrays

OSQlgen can match a few basic linear algebra operations such as matrix-matrix multiplication, matrix-vector multiplication, matrix transpose, etc. in the input programs translate them to Spark SQL program on block arrays. A programmer can decide the type of the input array blocks to be either sparse or dense.

5.3.3 Translation of Semiring to SQL on Block Arrays

In our framework, the array-based graph programs that are based on a semiring are translated 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 update step, a new graph is produced from an existing graph where the computation is equivalent to a semiring, and in 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 (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 comprehension. 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:

$$\begin{aligned}\{\, (g, \oplus/v) \mid (\bar{\imath}, m) \leftarrow A,\ (\bar{\jmath}, n) \leftarrow B, \rho_1(\bar{\imath}) = \rho_2(\bar{\jmath}),\\ \textbf{let}\, v = m \otimes n,\ \textbf{group by}\ g : f(\bar{\imath}, \bar{\jmath}) \,\}\end{aligned}$$

This comprehension retrieves the key-value pairs $(\bar{\imath}, m)$, and $(\bar{\jmath}, n)$ from generators A and B, where $\bar{\imath}$ and $\bar{\jmath}$ are tuples that contain indices of the generators. In the join condition ($\rho_1(\bar{\imath}) = \rho_2(\bar{\jmath})$), 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, we group v using group-by key g which is a function f applied on the indices of the generators. 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 input code block, we pattern-match using the semantic function $\mathcal{S}$ to check if it is equivalent to a semiring. If a match is found, the comprehension on coordinate arrays is transformed to comprehension on block arrays:

$$\begin{aligned}\mathcal{S}[\![\{\, (g, \oplus/v) \mid (\bar{\imath}, m) \leftarrow A,\ (\bar{\jmath}, n) \leftarrow B, \rho_1(\bar{\imath}) = \rho_2(\bar{\jmath}),\\ \textbf{let}\, v = m \otimes n,\ \textbf{group by}\ g : f(\bar{\imath}, \bar{\jmath}) \,\}]\!] =\\ \{\, (G, \oplus_b/V) \mid (\bar{I}, M) \leftarrow Ab,\ (\bar{J}, N) \leftarrow Bb,\ \rho_1(\bar{I}) = \rho_2(\bar{J}),\\ \textbf{let}\, V = M \otimes_b N,\ \textbf{group by}\ G : f(\bar{I}, \bar{J}) \,\} \qquad (5.1)\end{aligned}$$

$$\mathcal{S}[\![\{\, (h \mid x \leftarrow A \,\}]\!] = \{\, H \mid X \leftarrow Ab \,\} \qquad (5.2)$$

Here, in rule (5.1) the coordinate matrices A and B of type $\{((\text{Long}, \text{Long}), \text{Double})\}$ is transformed to block matrices Ab and Bb of type $\{((\text{Int}, \text{Int}), \text{Array[Double]})\}$, where $\overline{I}$, $\overline{J}$ represent the indices of each block of Ab, and Bb, respectively, and M and N represent the values of the block. If the generator B is a vector of type $\{(\text{Long}, \text{Double})\}$, it is transformed to block vector Bb of type $\{(\text{Int}, \text{Array[Double]})\}$, where $\overline{J}$ represents the index of the block, and N represents the values of the block. The updated graph computed using rule (5.1) replaces the current graph using rule (5.2), where h and H refer to the headers of coordinate and block matrices respectively.

5.3.3.1 Pattern Compilation and Comprehension Translation

After the coordinate arrays are mapped to block arrays, we apply pattern compilation to get rid of the pattern variables since SQL doesn't support patterns. To do that, we use the semantic function described in section 4.2.1.1. For example, after applying pattern compilation, the following comprehension:

$$\begin{aligned}\pmb{\{}\, ((I,J), \oplus_b/V) \,\pmb{|}\, & ((I,K),M) \leftarrow Ab,\ ((K',J),N) \leftarrow Bb, \\ & K = K',\ \mathbf{let}\, V = M \otimes_b N, \\ & \mathbf{group\ by}\ (I,J)\,\pmb{\}}\end{aligned}$$

is transformed to:

$$\begin{aligned}\pmb{\{}\, ((X.\_1.\_1, Y.\_1.\_2), \oplus_b/V) \,\pmb{|}\, & (X \leftarrow Ab,\ Y \leftarrow Bb, \\ & X.\_1.\_2 = Y.\_1.\_1, \\ & \mathbf{let}\, V = X.\_2 \otimes_b Y.\_2, \\ & \mathbf{group\ by}\ (X.\_1.\_1, Y.\_1.\_2)\,\pmb{\}}\end{aligned}$$

Here, the pattern variables in the generators Ab and Bb are replaced with new variables X and Y and the pattern variables in the rest of the qualifiers are expressed in terms of these new variables. Here, the pattern variables I, K and M are expressed as $X.\_1.\_1$, $X.\_1.\_2$, and $X.\_2$. Similarly, for the generator Bb, the pattern variables K', J and N are expressed as $Y.\_1.\_1$, $Y.\_1.\_2$, and $Y.\_2$.

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

$$\begin{aligned}
&\mathcal{B}[\![\{\, (G, \oplus_b/V) \mid ((X \leftarrow Ab,\, Y \leftarrow Bb, K = K', \\
&\quad \textbf{let}\, V = M \otimes_b N,\ \textbf{group by}\ G\,\}]\!] = \\
&\quad {''}\textbf{select} \qquad struct(G),\ tile\_sum(\\
&\qquad\qquad\qquad collect\_list(mult\_tiles(\mathrm{M}, \mathrm{N}))) \\
&\quad \textbf{from} \qquad Ab\, X\ \textbf{join}\ Bb\, Y\ \textbf{on}\ K = K' \\
&\quad \textbf{group by} \quad G'' && (5.3) \\
&\mathcal{B}[\![v \leftarrow e,\, \overline{q}]\!] = \mathcal{B}[\![\overline{q}]\!] && (5.4) \\
&\mathcal{B}[\![e,\, \overline{q}]\!] = \mathcal{B}[\![\overline{q}]\!] && (5.5) \\
&\mathcal{B}[\![\ ]\!] = \emptyset && (5.6)
\end{aligned}$$

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. The multiplicative monoid $*_b$ 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;
}
```

In Spark SQL, user-defined functions are called UDFs. If $\oplus$ and $\otimes$ represent $+$ and $*$, this is simply a multiplication between the tiles/blocks of the input matrices.

User-defined function *tile_sum* applies additive monoid after aggregating the blocks using built-in function *collect_list*. The additive monoid $+_b$ is defined as:

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

We also provide special functions if the header of the comprehension in (5.2) contains scalar operation on a column. For example, for addition of a constant c to 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 {$i \leftarrow 0$ until $N$}
   $V(i) := V(i) + c$;
  V;
}
```

If there are code blocks containing comprehensions on the same array after a code block containing a semiring that are not semirings, we translate the block array back to coordinate array:

$$C := \{\, ((I * N + i, J * N + j), V(i + N * j) \mid ((I, J), V) \leftarrow Cb,\ i \leftarrow 0 \ \textbf{until}\ N, j \leftarrow 0 \ \textbf{until}\ N \,\} \tag{5.7}$$

That way, the SQL programs generated from the non-semiring code block will operate on coordinate array.

5.3.4 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 comprehensions of this program are:

$$R := \{\, ((i,j), min/v) \mid ((i,k),m) \leftarrow G, ((k',j),n) \leftarrow G,\ k = k', \mathbf{let}\, v = m + n,\ \mathbf{group\ by}\ (i,j) \,\} \tag{5.8}$$

$$G := R \tag{5.9}$$

We apply semantic function $\mathcal{S}$ on them which transforms the comprehensions on coordinate arrays to comprehensions on block arrays using (5.1) and (5.2):

$$Rb := \{\, ((I,J), min_b/V) \mid ((I,K),M) \leftarrow Gb,\ ((K',J),N) \leftarrow Gb, K = K',\ \mathbf{let}\, V = M +_b N,\ \mathbf{group\ by}\ (I,J) \,\} \tag{5.10}$$

$$Gb := Rb \tag{5.11}$$

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

$$Rb := \{\, ((X.\_1.\_1, Y.\_1.\_2), (min_b/V)) \mid (X \leftarrow Gb,\ Y \leftarrow Gb, X.\_1.\_2 = Y.\_1.\_1, \mathbf{let}\, V = X.\_2 +_b Y.\_2, \mathbf{group\ by}\ (X.\_1.\_1, Y.\_1.\_2) \,\} \tag{5.12}$$

$$Gb := Rb \tag{5.13}$$

Then, the transformed comprehension (5.12) is translated to Spark SQL program on block arrays using (5.3):

$$\begin{aligned}
Rb := \textbf{select}\quad & \textit{struct}(\mathrm{X.\_1.\_1},\ \mathrm{Y.\_1.\_2}),\ \textit{tile\_min}(\\
& \quad \textit{collect\_list}(\textit{min\_tiles}(\mathrm{X.\_2}, \mathrm{Y.\_2}))) \\
\textbf{from}\quad & \mathrm{Gb}\,X\ \textbf{join}\ \mathrm{Gb}\,Y\ \textbf{on}\ \mathrm{X.\_1.\_2} = \mathrm{Y.\_1.\_1} \\
\textbf{group by}\quad & \mathrm{X.\_1.\_1},\ \mathrm{Y.\_1.\_2}
\end{aligned} \tag{5.14}$$

Now, let's consider one iteration of PageRank computation of a graph using the following equation: $r' = \beta Mr + (1-\beta)/n$. At first, rank vector r is initialized to $1/n$, where n is the total number of nodes in the graph. $(1-\beta)/n$ is assigned to variable a which represents the introduction of a new random surfer at a random page with probability $(1-\beta)$. In the transition matrix M, M_{ij} has value $1/k$ if page j has k outgoing edges. Then M is multiplied with β and assigned to the variable G.

```
var $s$ : vector[Double] = vector();
for $i = 0, n-1$ do {
  $s[i] := 0$;
  for $j = 0, n-1$ do {
    $s[i]$ += $G[i,j] * r[j]$;
  }
    $r[i] := s[i] + a$;
}
```

The comprehensions of this program are:

$$s := \{\, (i, +/v) \mid ((i,k), m) \leftarrow G, (k', n) \leftarrow r, k = k', \textbf{let}\ v = m * n,\ \textbf{group by}\ i \,\} \tag{5.15}$$

$$r := \{\, (i, v + a)) \mid (i, v) \leftarrow s \,\} \tag{5.16}$$

The comprehensions are transformed to comprehensions on block arrays by applying semantic function $\mathcal{S}$ using (5.1) and (5.2):

$$\begin{aligned} sb &:= \{\, (I, +_b/V) \mid ((I,K),M) \leftarrow Gb, (K',N) \leftarrow rb,\ K = K', \\ &\qquad\qquad \textbf{let}\, V = M *_b N,\ \textbf{group by}\ I \,\} && (5.17) \\ rb &:= \{\, (I, V +_b a)) \mid (I,V) \leftarrow sb \,\} && (5.18) \end{aligned}$$

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

$$\begin{aligned} sb &:= \{\, X.\_1.\_1, +/V) \mid (X \leftarrow Gb,\ Y \leftarrow rb,\ X.\_1.\_2 = Y.\_1, \\ &\qquad\qquad \textbf{let}\, V = X.\_2 *_b Y.\_2,\ \textbf{group by}\ X.\_1.\_1) \,\} && (5.19) \\ rb &:= \{\, (X.\_1, X.\_2 + a) \mid X \leftarrow sb \,\} && (5.20) \end{aligned}$$

Then the first transformed comprehension (5.19) is translated to Spark SQL program on block arrays using (5.3):

$$\begin{aligned} sb := \textbf{select}\quad & \mathrm{X.\_1.\_1},\ \mathit{tile\_sum}(\mathit{collect\_list}(\\ & \qquad \mathit{mult\_tiles}(\mathrm{X.\_2}, \mathrm{Y.\_2}))) \\ \textbf{from}\quad & \mathrm{Gb}\, X\ \textbf{join}\ \mathrm{rb}\, Y\ \textbf{on}\ \mathrm{X.\_1.\_2} = \mathrm{Y.\_1} \\ \textbf{group by}\quad & \mathrm{X.\_1.\_1} && (5.21) \end{aligned}$$

For the second comprehension, the header contains a scalar operation with the second column. Therefore, our special function *vec_sum* is applied on the header and then using semantic functions described in our earlier work for comprehension translation *rb* equals to:

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

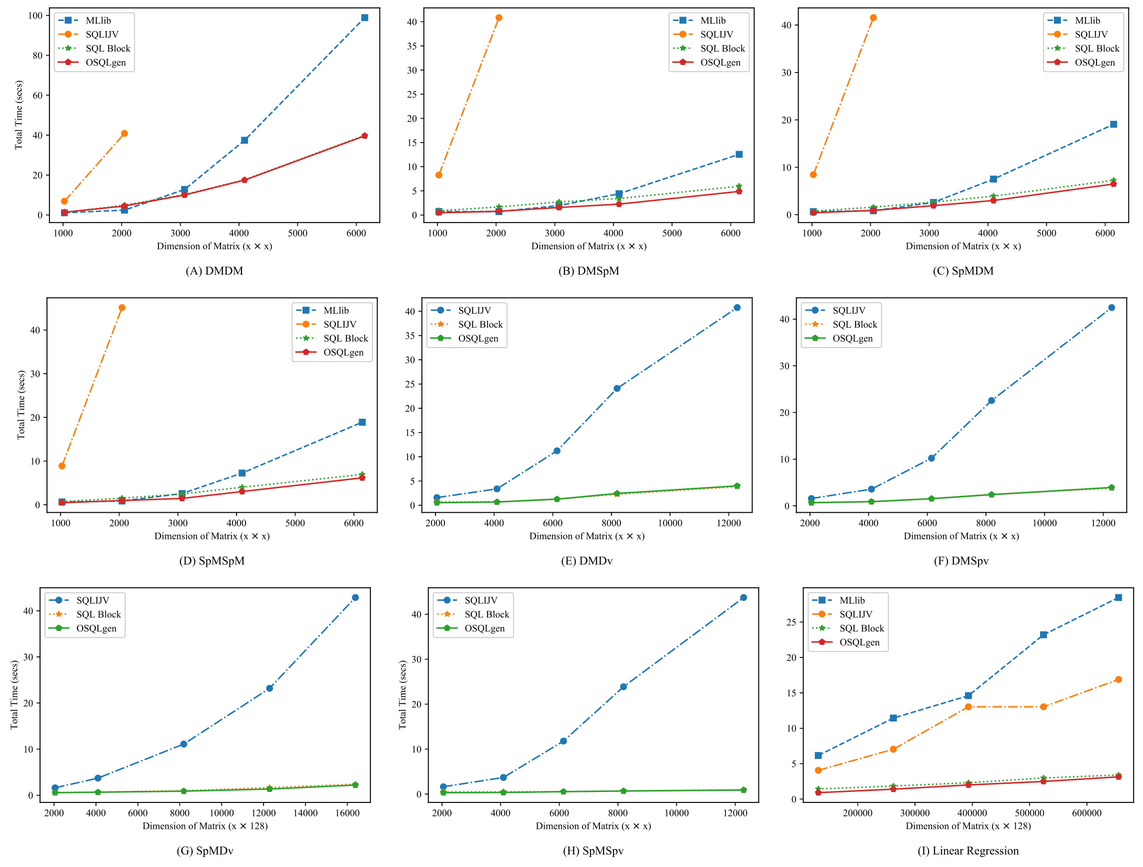

Figure 5.1. Performance of OSQLgen on linear algebra operations relative to MLlib, hand-written Spark SQL on programs on coordinate and block arrays.

5.4 Performance Evaluation

Our solution to generate optimized queries based on semirings is integrated into our existing system SQLGen [18] which in turn is implemented on top of DIABLO [13]. The input programs are translated to monoid comprehensions and then to optimized SQL program. 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 [46], GraphFrames [37],

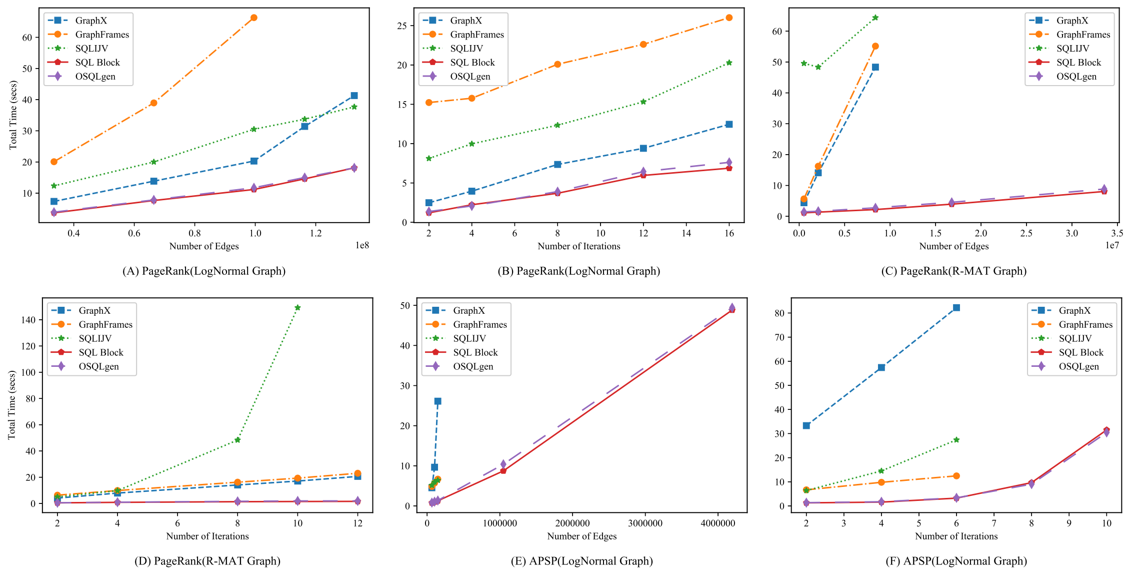

Figure 5.2. Performance of OSQLgen relative to GraphX, GraphFrames, and handwritten Spark SQL on COO and Block arrays.

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) [47]. 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, 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 5.1 and Figure 5.2 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 5.1.

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 was 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 hand-written 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 product of a matrix and vector with four different combinations: Dense-Dense (DMDv), Dense-Sparse (DMSpv), Sparse-Dense (SpMDv), and Sparse-Sparse (SpM-Spv). 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, 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× (Sp-MDv), 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 linear regression algorithm. The formula used to do this is the following: $theta = theta - (a * 1/m * X^T) \times (X \times theta - y)$ 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 this 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 5.2. In first set of experiments, we compare the performance of graph algorithms on different graph size 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 number 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 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, and the GraphFrames libraries, 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}(apprx.) edges. Each program was run for 8 iterations for 5 sizes of input datasets, except GraphFrames which was run for 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 increasing number of iterations (up to 18) for a fixed size of input graph with 2^{18} vertices and 2^{25}(apprx.) edges generated by GraphX log-normal graph generator with 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}(apprx.) 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}(apprx.) edges generated by GraphX R-MAT graph generator with 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 was run for 4 sizes of input datasets. For this experiment, the speedup ranges of OSQLgen was 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 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 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}(apprx.) 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 same parameters as in experiment A. The largest graph generated in this experiment had 1.5×2^8 vertices and 2^{15}(apprx.) 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.

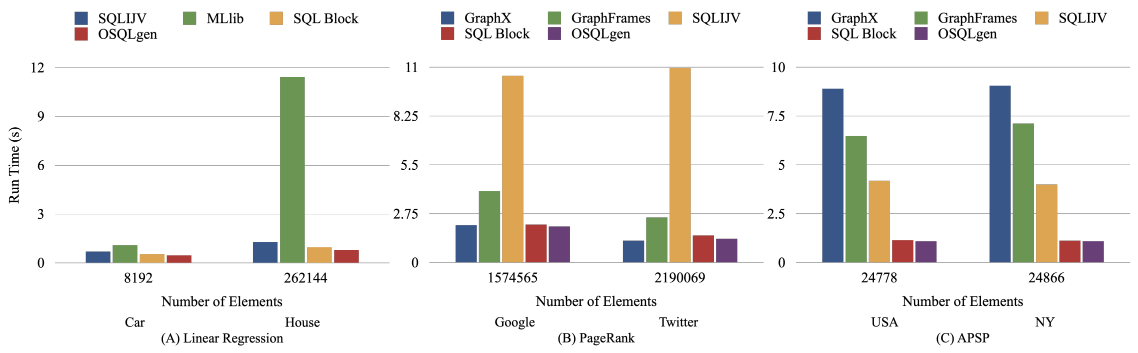

Figure 5.3. Performance of OSQLgen on graph algorithms on real datasets relative to GraphX, GraphFrames, MLlib and hand-written Spark SQL on coordinate and Block arrays.

Finally, we compared the performance of OSQLgen for linear regression, PageRank, and all-pairs shortest path problems on real datasets as shown in Figure 5.3. For linear regression, we used Car dataset from Craigslist [48] and Housing dataset of England and Wales [49]. In these experiments, all the systems performed similar except MLlib which performed slower than other systems on Housing dataset. For PageRank algorithm, we have used Google web graph [50] and twitter dataset [51]. In these experiments, GraphX, OSQLgen, and hand-written Spark SQL program on 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 [52]. 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.

5.5 Conclusion

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

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:

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

where F converts a block to a coordinate list:

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

When $k = 2$:

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

where $F(M)$ is:

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

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

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

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

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

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

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

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

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

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

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

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

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

Theorem 5.5.1. *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 $k = 2, \oplus = +, \oplus_b = +_b, \otimes = *, \otimes_b = *_b$:

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

CHAPTER 6

Conclusion and Future Work

We have presented two frameworks, SQLgen and OSQLgen. These frameworks translate array-based loop programs to Spark SQL programs. SQLgen translates array-based loop programs to comprehensions and then to Spark SQL programs on coordinate arrays. On the other hand, OSQLgen translates array-based loop programs to comprehensions and then checks if the comprehension is a semiring comprehension. If it is, it translates these comprehensions to Spark SQL programs on block arrays. The block computations are implemented as user-defined functions (UDF) on Spark. Since Spark has recently started providing GPU support, as a future work, we plan to implement more UDFs on GPUs. We also plan to explore different storage formats for the array blocks since choosing the right storage format can result in a significant performance gain both in terms of storage and computation. Furthermore, we plan to explore different partitioning strategies for arrays such that our system can automatically choose an optimal partitioning scheme for the arrays.

REFERENCES

[1] 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

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

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

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

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

[6] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine, “Scalable linear algebra on a relational database system,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 7, pp. 1224–1238, 2018.

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

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

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

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

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

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

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

[14] 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/

[15] M. Zaharia, R. S. Xin, Wendell, *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

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

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson, "The tiledb array data storage manager," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 349–360, 2016.

[25] 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*, 2015, pp. 1–6.

[26] 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*, 2011, pp. 253–264.

[27] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt, "Scihadoop: Array-based query processing in hadoop," in *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 2011, pp. 1–11.

[28] 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.* IEEE, 2013, pp. 887–894.

[29] 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).* IEEE, 2012, pp. 443–450.

[30] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *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.

[31] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "Systemml: Declarative machine learning on mapreduce," in *2011 IEEE 27th International Conference on Data Engineering.* IEEE, 2011, pp. 231–242.

[32] L. Chen, A. Kumar, J. Naughton, and J. M. Patel, "Towards linear algebra over normalized data," *arXiv preprint arXiv:1612.07448*, 2016.

[33] 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.

[34] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[35] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[36] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *First international workshop on graph data management experiences and systems*, 2013, pp. 1–6.

[37] 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.

[38] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra.* SIAM, 2011.

[39] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[40] 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.

[41] A. V. Aho, M. S. Lam, R. Sethi, , and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Reading, Massachusetts: Addison-Wesley, 2007.

[42] S. Funk, "Netflix update: Try this at home," 2006.

[43] T. Haveliwala, "Efficient computation of pagerank," Stanford, Tech. Rep., 1999.

[44] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 2004, pp. 442–446.

[45] H. Eves, *Elementary Matrix Theory*. Allyn and Bacon Inc, 1966.

[46] 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.

[47] 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.

[48] Used cars dataset. [Online]. Available: https://www.kaggle.com/austinreese/craigslist-carstrucks-data

[49] Uk housing prices paid. [Online]. Available: https://www.kaggle.com/hm-land-registry/uk-housing-prices-paid

[50] Google web graph. [Online]. Available: https://snap.stanford.edu/data/web-Google.html

[51] Social circles: Twitter. [Online]. Available: https://snap.stanford.edu/data/ego-Twitter.html

[52] Shortest paths. [Online]. Available: https://www.diag.uniroma1.it/challenge9/download.shtml

BIOGRAPHICAL STATEMENT

Md Hasanuzzaman Noor was born in the district of Chittagong, Bangladesh. He received his B.Sc in Computer Science and Engineering from Chittagong University of Engineering and Technology, Chittagong. His research interests include Big Data, Cloud Computing, Graph Algorithms, and Machine Learning. He has published in top-tier conferences such as VLDB, IEEE Big Data, etc.