A Query Processing Framework for Large-Scale Scientific Data Analysis[⋆]

Leonidas Fegaras

University of Texas at Arlington
fegaras@cse.uta.edu

Abstract. Current scientific applications must analyze enormous amounts of array data using complex mathematical data processing methods. This paper describes a distributed query processing framework for large-scale scientific data analysis that captures array-based computations using SQL-like queries and optimizes and evaluates these computations using state-of-the-art parallel processing algorithms. Instead of providing a library of concrete distributed algorithms that implement certain matrix operations efficiently, we generalize these algorithms by making them parametric in such a way that the same efficient implementations that apply to the concrete algorithms can also apply to their generic counterparts. By specifying matrix operations as generic algebraic operators, we are able to perform inter-operator optimizations, such as fusing matrix transpose with matrix multiplication, resulting to new instantiations of the generic algebraic operators, without having to introduce new efficient algorithms on the fly. We report on a prototype implementation of our framework on three Big Data platforms: Hadoop Map-Reduce, Apache Spark, and Apache Flink, using Apache MRQL, which is a query processing and optimization system for large-scale, distributed data analysis. Finally, we evaluate the effectiveness of our framework through experiments on three queries: a matrix multiplication query, a simple query that combines matrix multiplication with matrix transpose, and a complex iterative query for matrix factorization.

Keywords: Big Data; Scientific Data Analysis; Query Processing

1 Introduction

In recent years, it has become easier and cheaper than ever to collect data but harder to turn these data into value. In computational science, the explosion in scientific data generated by experiments and simulations has created a major challenge for many scientific projects. For data scientists who need to analyze vast volumes of data, data-intensive processing is fast becoming a necessity. They need algorithms capable of scaling to petabytes and faster tools that are more sophisticated, more reliable, and easier to use.

As datasets grow larger, new frameworks in distributed Big Data analytics have become essential tools to large-scale machine learning and scientific discoveries. Among

[⋆] Appeared at the Transactions on Large-Scale Data- and Knowledge-Centered Systems (TLDKS), 38:119-145, 2018. LNCS, volume 11250. The final publication is available at Springer via https://doi.org/10.1007/978-3-662-58384-5_5 these frameworks, the Map-Reduce programming model [14] has emerged as a generic, scalable, and cost effective solution for Big Data processing on clusters of commodity hardware. The Map-Reduce paradigm is a scale-out solution that brings computations to the data, rather than data to the computations. This is a drastic departure from high-performance computing models, which make a clear distinction between processing and storage nodes. Very soon, though, it became apparent that the Map-Reduce model has many limitations. To address these limitations, new alternative frameworks have been introduced recently that perform better than Map-Reduce for a wider spectrum of workloads, such as Google's Pregel [33], Apache Spark [9], and Apache Flink [2], which are in-memory distributed computing systems.

Currently, many programmers prefer to use a higher-level declarative language to code their data-centric applications, such as Apache Hive [5] and PigLatin [36], instead of coding them directly in an algorithmic language, such as Java. For instance, Hive is used for over 90% of Facebook Map-Reduce jobs. Most Map-Reduce query languages though provide a limited syntax for operating on data collections, in the form of simple relational joins and group-bys. They cannot express complex data analysis tasks, such as PageRank, data clustering, and matrix factorization, using SQL-like syntax exclusively. Because of these limitations, these languages enable users to plug-in custom scripts into their queries for those jobs that cannot be declaratively coded in their query language. This nullifies the benefits of using a declarative query language and may result in platform-dependent, suboptimal, error-prone, and hard-to-maintain code. Furthermore, some of these languages are inappropriate for complex scientific and graph analysis applications, because they do not directly support iteration in declarative form and are not able to handle complex scientific data. But there are some recent query systems, such as Apache MRQL [8], which are powerful enough to express complex data analysis tasks.

In the past, large-scale data processing was mainly done in the realm of scientific computing. In recent years, the volume of data generated by scientists through experiments and simulations has been steadily increasing at an unprecedented rate. For example, the Large Hadron Collider at CERN and astronomy's Pan-STARRS5 array of celestial telescopes are capable of generating several petabytes of data per day, which need to be made available and analyzed by scientists on worldwide grids of computers. Data-intensive scientific computing shares some of the key ingredients of cloud computing. Just like in cloud computing, scientific computing is driven to use the most efficient computing techniques available, including high-performance computing and low-level data management. Many scientific data generated by scientific experiments and simulations come in the form of arrays, such as the results from high-energy physics, cosmology, and climate modeling. In addition, many algorithms for scientific data analysis and simulation are frequently expressed in terms of array operations. Furthermore, most scientific file formats used by scientists to store data, such as HDF5 [22] and NetCDF [35], are based on array structures. Since most of the data generated by scientists are in array form, current scientific applications must analyze enormous amounts of array data using complex mathematical data processing methods. Scientists are typically comfortable with numerical analysis tools, such as MatLab, but are not familiar with the intricacies of Big Data analysis and distributed computing. A declarative distributive query language capable of expressing complex mathematical operations on arrays could help them develop their data analysis applications without any prior knowledge of distributed computing.

The goal of this paper is to support large-scale scientific data analysis by 1) extending an existing distributed query language, namely Apache MRQL [8], with array operations that can capture most array-based computations in declarative form and 2) by developing a query processing framework that can optimize and evaluate these computations using state-of-the-art parallel processing algorithms. Other proposed systems [41,38,27,11] focus on storage structures and indexing techniques for arrays, such as chunking and tiling, to achieve better performance on certain parallel array computations. Although such storage layouts may speed up the processing of individual array operations, they produce results in a certain layout that may need to be restructured before they are used by the subsequent matrix operations. Furthermore, such schemes do not address inter-operation optimization, which is the focus of our work. Our approach is to accept any kind of array representation and storage but at the same time be able to recognize certain array operations in a query and translate them into efficient parallel array processing algorithms. For example, matrix multiplication $X \times Y$ between two sparse matrices X and Y can be implemented efficiently in a distributed environment using a 2D mesh of processors [24,44] by distributing the data to worker nodes in the form of a grid of partitions, where each partition contains only those rows from X and those columns from Y needed to compute a single grid partition of the resulting matrix. If a query language were to adopt a certain matrix representation and provide a fixed number of matrix operations in the form of predefined operators or library functions, then the task of recognizing these operations and mapping them to efficient algorithms would have become easy. Such an approach though does not leave many opportunities of inter-operator optimization, such as fusing matrix transpose with matrix multiplication, because the resulting fused operation would have to be a new operation that requires the introduction of a new efficient algorithm on the fly. Instead of looking at concrete algorithms that implement specific mathematical operations, our objective is to generalize these algorithms by making them parametric in such a way that the same efficient implementations that apply to the concrete algorithms can also apply to their generic counterparts.

The most effective method of making an algorithm parametric is to make it higher-order by abstracting parts of its computations into its functional parameters. Such a higher-order operation must capture the essence of the concrete algorithm it generalizes by facilitating an equivalent data distribution and by supporting a similar parallel processing method. To generate such a higher-order operation from a query, a query evaluator must be able to recognize certain syntactic patterns in the query, in their most generic form, that can be mapped to this operation. This task can become more feasible if it is done at the algebraic operation level, rather than at the syntactic level. That is, instead of introducing source-to-source transformations to match parts of a query with certain generic syntactic patterns that correspond to a generic operation, our approach is to translate queries to algebraic forms and then normalize and rewrite these forms into these algorithms using algebraic rewrite rules. We believe that this approach is very effective when applied, not only to matrix operations, but also to a wide spectrum of queries whose functionality is in essence equivalent to these matrix operations.

The contribution of this work can be summarized as follows:

- We define a higher-order operator, called *GroupByJoin*, that generalizes many algorithms that correlate two data sources using an equi-join followed by a group-by with aggregation.
- We provide an efficient implementation of GroupByJoin based on an algorithm that generalizes the SUMMA parallel algorithm for matrix multiplication on two Big Data frameworks: Map-Reduce and Spark.
- We provide an extension to the query optimization framework for MRQL to generate physical plans that use the GroupByJoin operator. This is accomplished with algebraic rewrite rules that recognize certain patterns in the algebraic terms derived from MRQL queries that are equivalent to a GroupByJoin operation. We show how these rewrite rules can be used, in conjunction with the existing algebraic optimization rules in MRQL, to minimize the amount of data shuffling in queries that contain consecutive matrix operations.
- We report on a prototype implementation of our framework using Apache MRQL running on top of three different Big Data platforms: Hadoop Map-Reduce, Apache Spark, and Apache Flink. We show the effectiveness of our methods through experiments on three queries, a matrix multiplication query, a simple query that combines matrix multiplication with matrix transpose, and a very complex query for matrix factorization, which not only is iterative but it also contains many matrix operations at every iteration step.

The rest of this paper is organized as follows. We compare our approach with related work in Section 2. We summarize our earlier work on query optimization for MRQL in Section 3. We highlight the basic ideas behind our approach in Section 4. We introduce the higher-order operator GroupByJoin in Section 5). We provide an efficient implementation of GroupByJoin based on the SUMMA algorithm on two Big Data frameworks: Map-Reduce (Section 6) and Spark (Section 7). We present our framework for translating and optimizing MRQL queries to GroupByJoin operations in Section 8. Finally, we evaluate our framework through experiments on three MRQL queries in Section 9.

2 Related Work

One of the major drawbacks of the Map-Reduce model is that, to simplify reliability and fault tolerance, it does not preserve data in memory between the map and reduce tasks of a Map-Reduce job or across consecutive jobs, which imposes a high overhead to complex workflows and graph algorithms, such as PageRank and matrix factorization, that require repetitive Map-Reduce jobs. To achieve better performance for such complex workflows, it is crucial to minimize the required number of Map-Reduce jobs, mostly because of the high overhead of dumping the intermediate results between consecutive Map-Reduce jobs to the HDFS. As an alternative solution, some recent systems for cloud computing go beyond Map-Reduce by maintaining dataset partitions in the memory of the compute nodes. These systems include the main memory Map-Reduce M3R [40], Apache Spark [9], Apache Flink [2], and distributed GraphLab [32].

There are also a number of higher-level languages that make Map-Reduce programming easier, such as HiveQL [42], PigLatin [36], SCOPE [12], and Dryad/Linq [28]. Apache Hive [42,43] provides a logical RDBMS environment on top of the Map-Reduce engine, well-suited for data warehousing. Using its high-level query language, HiveQL, users can write declarative queries, which are optimized and translated into Map-Reduce jobs that are executed using Hadoop. HiveQL does not handle nested collections uniformly: it uses SQL-like syntax for querying data sets but uses vector indexing for nested collections. Unlike MRQL, HiveQL has many limitations. It does not allow query nesting in predicates and select expressions, but allows a table reference in the from-part of a query to be the result of a select-query. Apache Pig [23] resembles Hive as it provides a user-friendly scripting language, called PigLatin [36], on top of Map-Reduce, which allows explicit filtering, map, join, and group-by operations. Like Hive, PigLatin performs very few optimizations based on simple rule transformations. PACT/Nephele [10] is a Map-Reduce programming framework based on workflows, which consist of high-order operators, such as map and reduce. These workflows are converted to logical execution plans for Nephele, a general distributed program execution engine. Even though PACT/Nephele workflow programs are very flexible and are not limited to rigid Map-Reduce pairs, they are hard to program, since programmers have to construct low-level workflows. SCOPE [12], an SQL-like scripting language for large-scale analysis, does not support sub-queries but provides syntax to simulate sub-queries using outer-joins. Like Hive, because of its limitations, SCOPE provides syntax for user-defined process/reduce/combine operations to capture explicit Map-Reduce computations. DryadLINQ [46] is a programming model for large scale data-parallel computing that translates programs expressed in the LINQ programming model to Dryad, which is a distributed execution engine for data-parallel applications. Unlike MRQL, the LINQ query syntax is very limited and has limited support for query nesting.

Vertex-centric graph-parallel programming is a new popular framework for large-scale graph processing. It was introduced by Google's Pregel [33] but is now available by many open-source projects, such as Apache Giraph [6], Apache Hama [4], and Spark's GraphX [7]. Most of these frameworks are based on the Bulk Synchronous Parallelism (BSP) programming model [44]. VERTEXICA [26] and Grail [15] provide the same vertex-centric interface as Pregel but, instead of a distributed file system, they use a relational database to store the graph and the exchanged messages across the BSP supersteps. Unlike Grail, which can run on a single server only, VERTEXICA can run on multiple parallel machines connected to the same database server. Such configuration may not scale out very well because the centralized database may become the bottleneck of all the data traffic across the machines. Although MRQL is a general-purpose Big Data query system, graph queries in MRQL are expressed using SQL-like syntax since graphs are captured as regular distributed collections. These queries are translated to distributed self-joins over the graph data.

Many scientific data generated by scientific experiments and simulations come in the form of arrays, such as the results from high-energy physics, cosmology, and climate modeling. Many of these arrays are stored in scientific file formats that are based on array structures, such as, CDF (Common Data Format), FITS (Flexible Image Transport System), GRIB (GRid In Binary), NetCDF (Network Common Data Format), and various extensions to HDF (Hierarchical Data Format), such as HDF5 and HDF-EOS (Earth Observing System). HDF5 [22] is a data model and file format that enables the data to be organized into hierarchical structures, called groups and datasets. NetCDF [35] is a self-describing data format that supports the creation, access, and sharing of scientific data. It is commonly used in climatology, meteorology, and GIS applications. Many array-processing systems use special storage techniques, such as regular tiling, to achieve better performance on certain array computations. TileDB [37] is an array data storage management system that performs complex analytics on scientific data. It organizes array elements into ordered collections called fragments, where each fragment is dense or sparse, and groups contiguous array elements into data tiles of fixed capacity. Unlike our work, the focus of TileDB is the I/O optimization of array operations by using small block updates to update the array stores. SciDB [41,39] 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 [38] 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. SystemML [27] is an array-based declarative language to express large-scale machine learning algorithms, implemented on top of Hadoop. It supports many array operations, such as matrix multiplication, and provides alternative implementations to each of them. SciHadoop [11] is a Hadoop plugin that allows scientists to specify logical queries over arrays stored in the NetCDF file format [35]. 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 [25] 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 MapReduce. 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. Unlike MRQL, SciHive implements matrix multiplication operations using relational joins, instead of group-by-joins. SciMATE [45] 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. Finally, MLlib [34], which is part of MLbase [30], is a machine learning library built on top of Spark and includes algorithms for fast matrix manipulation based on native (C++ based) linear algebra libraries. Unlike MRQL, which supports ad-hoc data analysis based on arbitrary vector and matrix representations, MLlib provides a uniform rigid set of high-level APIs that consists of several statistical, optimization, and linear algebra primitives that can be used as building blocks for data analysis applications. Like all API-based frameworks, MLlib does not support inter-operation optimizations, which is the focus of our approach.

The monoid algebra described in Section 8.1 has been introduced in our previous work on algebras for distributed big data analysis [18,17], which in turn was based on our early work on monoid algebras and calculi for object-oriented databases [20,21]. The work reported in this paper extends our previous work on array-based computations for Map-Reduce [16] by providing an evaluation framework for Spark and implementations and performance results for both the Spark and Flink platforms.

3 Background: The MRQL Query Language

Apache MRQL [8] is a query processing and optimization system for large-scale, distributed data analysis. MRQL was originally developed by the author [18,19], but is now an Apache incubating project with many developers and users worldwide. MRQL (the Map-Reduce Query Language) is an SQL-like query language for large-scale data analysis on computer clusters. The MRQL query processing system can evaluate MRQL queries in four modes: in Map-Reduce mode using Apache Hadoop [3], in BSP mode (Bulk Synchronous Parallel model) using Apache Hama [4], in Spark mode using Apache Spark [9], and in Flink mode using Apache Flink (previously known as Stratosphere) [2]. The MRQL query language is powerful enough to express most common data analysis tasks over many forms of raw in-situ data, such as XML and JSON documents, binary files, and CSV documents. MRQL is more powerful than other current high-level Map-Reduce languages, such as Hive [5] and PigLatin [36], since it can operate on more complex data and supports more powerful query constructs, thus eliminating the need for using explicit procedural code. With MRQL, users are able to express complex data analysis tasks, such as PageRank, k-means clustering, matrix factorization, etc, using SQL-like queries exclusively, while the MRQL query processing system is able to compile these queries to efficient Java code.

For example, the following MRQL query that calculates the k-means clustering algorithm (Lloyd's algorithm), by deriving k new centroids from the old (the stopping condition has been omitted):

```
repeat centroids = ...
step select < X: avg(s.X), Y: avg(s.Y) >
     from s in Points
     group by k: (select c from c in centroids
                    order by distance(c,s ))[0]
```

Query 1. k-Means Clustering

where Points is a dataset of points on the X-Y plane, centroids is the current set of centroids (k cluster centers), and distance is a function that calculates the Euclidean distance between two points. The repeat query syntax '**repeat** $R = i$ **step** s' defines the dataset R as the fixpoint of the step s by starting with $R = i$ and reassigning R to s at each iteration step, where s is a query that depends on R. (For brevity, the stopping condition of the repeat query has been omitted.) Here, the initial value of centroids (the ... value) is a bag of k random points. The inner select-query in the group-by assigns the closest centroid to a point s (where [0] returns the first tuple of an ordered list). The outer select-query in the repeat step clusters the data points by their closest centroid, and, for each cluster, a new centroid is calculated from the average values of its points.

4 Our Framework

One of the objectives of our work is to accept any kind of array representation but at the same time be able to recognize certain array operations in a query and translate them into efficient parallel array processing algorithms. Sparse vectors and matrices can be captured as regular collections in MRQL. For example, a sparse matrix M can be represented as a collection of triples, (v, i, j), for $v = M_{ij}$. Then, the matrix multiplication between two sparse matrices X and Y can be expressed as follows in MRQL:

```
select ( sum(z), i, j )
  from (x,i,k) in X, (y,k,j) in Y, z = x*y
 group by i, j
```

Query 2. Matrix Multiplication Query

that is, we retrieve the values $X_{ik} \in X$ and $Y_{kj} \in Y$ for all i, j, k, and we set $z = X_{ik} * Y_{kj}$. The group-by operation in MRQL lifts each non-group-by variable defined in the from-part of the query from some type T to a bag of T, indicating that each such variable must now contain multiple values, one for each group. Consequently, after we group by the indexes i and j, the variable z will be lifted to a bag of numerical values $X_{ik} * Y_{kj}$, for all k. Hence, **sum**(z) in the query header will sum up all these values, deriving $\sum_k X_{ik} * Y_{kj}$ for the ij element of the resulting matrix.

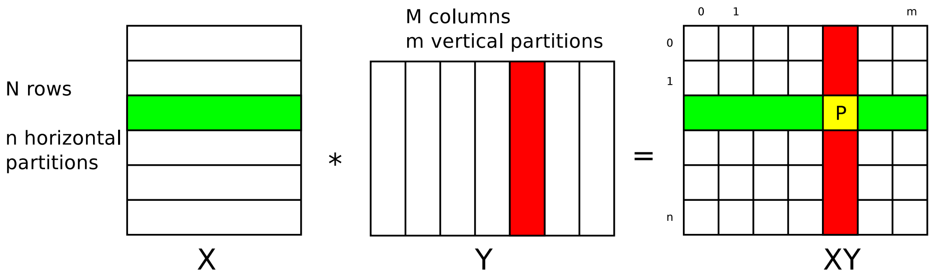

Fig. 1. Matrix Multiplication: each partition P requires N/n rows from X and M/m columns from Y

Matrix multiplication is an important operation, used frequently in scientific computations and machine learning. Suppose that X is an $N * K$ matrix and Y is an $K * M$ matrix. If the previous matrix multiplication query for $X \times Y$ is evaluated naively using an equi-join followed by a group-by, the intermediate result of the join would have been of size $N * K * M$, which would have to be shuffled to the compute nodes of the cluster for the group-by operation. Instead, one may use the SUMMA algorithm for matrix multiplication [24], which has been adapted for the BSP distributed model [44] and later for Map-Reduce [13]. This algorithm distributes the data as a grid of $m * n$ partitions, so that each partition contains N/n full rows from X and M/m full columns from Y (Fig. 1). That is, the X elements are replicated m times and the Y elements are replicated n times. Then, each partition is assigned to a single node in a cluster, which must have enough free memory to multiply the associated submatrices of size $N/n*K$ and $K*M/m$. The goal of this method is to minimize replication (m and n) so that the memory of each worker node in the cluster is fully utilized by performing the submatrix multiplication in memory. When implemented using Map-Reduce, this algorithm requires only one Map-Reduce job: the map task replicates and distributes the data to reducers, while each reducer multiplies its submatrices in memory using a hash join.

How can such algorithm be incorporated into the evaluation engine of a query language? One solution is to provide a library of predefined functions for various matrix operations, using their most efficient implementation. But such an approach does not leave any opportunities for inter-operation optimization. Consider, for example, Matrix Factorization using Gradient Descent [29], used in machine learning applications, such as for recommender systems. The goal of this computation is to split a matrix R of dimension $n \times m$ into two low-rank matrices P and Q of dimensions $n \times k$ and $k \times m$, for small k, such that the error between the predicted and the original rating matrix $R - P \times Q^T$ is below some threshold, where $P \times Q^T$ is the matrix multiplication of P with the transpose of Q and '$-$' is cell-wise matrix subtraction. Matrix factorization can be implemented using an iterative algorithm that repeatedly applies the following rules to minimize the error matrix E:

$$E \leftarrow R - P \times Q^T \tag{1}$$
$$P \leftarrow P + \gamma(2E \times Q^T - \lambda P) \tag{2}$$
$$Q \leftarrow Q + \gamma(2E \times P^T - \lambda Q) \tag{3}$$

where γ is the learning rate and λ is the the normalization factor used in avoiding overfitting. But matrix transpose and cell-wise operations can be fused with matrix multiplication, because they both correspond to a map operation, which can be incorporated into the map stage of the Map-Reduce operation that implements matrix multiplication, thus avoiding the extra map stage all together. That is, instead of defining matrix operations as opaque library functions, we can express them using sufficiently generic algebraic operations (i.e., higher-order functions) and use algebraic rewrite rules to fuse them, thus minimizing the number of processing stages and eliminating intermediate results. That way, in addition to offering more opportunities for optimization, application developers will not be forced to represent their data matrices in the single fixed representation used by the underlying implementation of the concrete matrix algorithms. Instead, they will be free to use any representation, thus focusing only on the computation logic.

In addition to array operations, by generalizing these algorithms, one can optimize a wider spectrum of queries that resemble matrix multiplication, such as calculating the shortest path distances of all node pairs in a graph. If we represent a graph G as a dataset of edges (i, j, d), where d is the distance between the graph nodes i and j, then this dataset is equivalent to a matrix G such that the distance d is the matrix value G_{ij}. Then, the shortest distance D_{ij} between i and j can be calculated by initially setting $D_{ij} = G_{ij}$ and $D_{ii} = 0$ and by repeatedly improving D_{ij} as follows:

$$D_{ij} \leftarrow \min(D_{ij}, \min_k(D_{ik} + G_{kj}))$$

which indicates that the shortest distance between a pair of nodes i and j in a graph G is the minimum $D_{ik}+G_{kj}$ among all graph nodes k, where D_{ik} is the shortest distance between i and k and G_{kj} is the distance between k and j. The operation $\min_k(D_{ik}+G_{kj})$ is similar to the matrix multiplication $D \times G$, but with addition instead of multiplication and minimum instead of addition. The MRQL query that expresses the shortest path distance algorithm is as follows:

```
1  repeat D = G union ( select (i, i ,0)
2                       from (i, j ,d) in G
3                       group by i )
4    step select ( i , j ,min(d))
5          from (i ,k,d1) in D, (k, j ,d2) in G, d = d1+d2
6          group by i, j
```

Query 3. Shortest Distance Query

As explained in Section 3, the repeat query syntax '**repeat** $D = i$ **step** s' starts with $D = i$ and reassigns D with the result of s at each iteration step. The initial value of D (lines 1 through 3) contains, in addition to the edges in G, the edges $(i, i, 0)$ (that is, $D_{ii} = 0$) so that the minimum calculation in line 4 includes the case for k = i, giving d $= G_{ij}$. The select query in the repeat step (lines 4-6) looks very similar to the matrix multiplication query (Query 2).

5 The GroupByJoin Operation

In this section, we generalize matrix multiplication using an algebraic operation, called a *Group-By Join*. Given two arbitrary bags X and Y, the following generic MRQL query:

```
select h( k, reduce(acc,zero,z) )
  from x in X, y in Y, z = (x,y)
 where jx(x) = jy (y)
 group by k: ( gx(x), gy(y) )
```

Query 4. Generic GroupByJoin Query

generalizes matrix multiplication, where

- the function jx is the left join key function,
- the function jy is the right join key function,
- the function gx is the left group-by function,
- the function gy is the right group-by function,
- the function h is the result function, and
- reduce(acc,zero,s) reduces the elements of a bag s using an accumulator acc, such that $\text{reduce}(\text{acc}, \text{zero}, \{z_1, z_2, \ldots, z_n\}) = \text{acc}(z_1, \text{acc}(z_2, \ldots, \text{acc}(z_n, \text{zero})))$ and $\text{reduce}(\text{acc}, \text{zero}, \{\,\}) = \text{zero}$.

This query joins the bags X and Y using the join keys jx and jy, then groups the join result by the group-by keys gx and gy, then aggregates the pairs z=(x,y) in each group using the reduce function, and finally transforms each result tuple using the function h. Note that, although the zero value in reduce(acc,zero,s) is not needed for a group-by aggregation since it is not possible for a group to be empty, it is needed for total aggregations on datasets that may be empty. To preserve bag semantics, function acc in reduce(acc,zero,s) must satisfy $\text{acc}(x, \text{acc}(y, z)) = \text{acc}(y, \text{acc}(x, z))$ and $\text{acc}(x, \text{zero}) = x$, for all x, y, and z.

The previous generic MRQL query is captured by the higher-order operation:

```
GroupByJoin( jx, jy, gx, gy, acc, zero, h, X, Y )
```

which generalizes the SUMMA algorithm by distributing X and Y into a grid of $n * m$ partitions based on their group-by and join key functions.

For example, the matrix multiplication in Query 2 is captured by the operation:

```
GroupByJoin( λ(x,i,k). k,                // the join key jx
             λ(y,k,j). k,                // the join key jy
             λ(x,i,k). i,                // the group-by key gx
             λ(y,k,j). j,                // the group-by key gy
             λ((x,y),c). c+x*y,          // the accumulator acc
             0,                          // the zero element
             λ((i,j),c). (c,i,j),        // the header h
             X, Y )                      // the input datasets
```

where $\lambda p.\, e$ is an anonymous function such that, if $f = \lambda p.\, e$, then $f(p) = e$. For example, for $f = \lambda$((i,j),c).(c,i,j), we have f((i,j),c) = (c,i,j).

Another example, is the select query in the repeat step of Query 3:

```
select ( i , j , min(d))
 from (i ,k,d1) in D, (k, j ,d2) in G, d = d1+d2
 group by i, j
```

which is captured by the operation:

```
GroupByJoin( λ(i,k,d1). k,                  // the join key jx
             λ(k,j,d2). k,                  // the join key jy
             λ(i,k,d1). i,                  // the group-by key gx
             λ(k,j,d2). j,                  // the group-by key gy
             λ((d1,d2),d). min(d,d1+d2),    // the accumulator acc
             0,                             // the zero element
             λ((i,j),d). (i,j,d),           // the header h
             D, G )                         // the input datasets
```

6 The Implementation of GroupByJoin in Map-Reduce

A straightforward implementation of the GroupByJoin operation is a join followed by a group-by with aggregation. In this section, we implement the GroupByJoin operation in the Map-Reduce framework using the SUMMA algorithm for matrix multiplication [24], which is, as we will see, more efficient than the straightforward implementation. Our implementation is based on the reduce-side join algorithm for Map-Reduce (described below) but it uses special partition, grouping, and sorting functions to capture and optimize the SUMMA algorithm. Before we describe the GroupByJoin implementation, let's see how the following join query between two datasets R and S:

```
select r.C, s.D
from r in R, s in S
where r.A = s.B
```

can be implemented in Map-Reduce. If one of the two datasets, such as R, is small enough to fit in the memory of every worker node in the cluster, then we can broadcast R to all worker nodes. This algorithm, called the map-backed join, is a Map-Reduce job that consists of a map stage only, without a reduce stage. Before the Map-Reduce job, the dataset R is broadcast to all worker nodes and each worker node creates a built hash table from R. Then the map stage of each worker node joins its input split of S with the hash table by R by probing the hash table. This join is very efficient but it requires that one of the datasets can fit in memory. If neither R nor S can fit in the memory of a worker node, then the join can be implemented using the reduce-side join algorithm [31], shown in Fig. 2.

```
mapLeft ( r ):
  emit(r.A,(1,r))

mapRight ( s ):
  emit(s.B,(2,s))

reduce ( key, values ):
  for each (1,r) in values
      for each (2,s) in values
          emit(key,(r.C,s.D))
```

Fig. 2. Map-Reduce pseudo-code for the Reduce-Side Join Between the Datasets R and S

The Map-Reduce job shown in Fig. 2 has two mappers, one for each dataset:

- mapLeft: a mapper for the dataset R that generates key-value pairs where the key is the join key R.A
- mapRight: a mapper for the dataset S that generates key-value pairs where the key is the join key S.B

The two mappers send the R and S values associated with the same keys R.A=S.B to the same reducer, where they are reduced together. The mapLeft mapper tags the R tuples with 1 and the mapRight mapper tags the S tuples with 2 so that the reducer can tell them apart. Since the reduce method is evaluated for each different key, the values are all the tuples from R and S that correspond to the same key, which is equal to R.A and S.B. The nested loop in the reduce method separates the R from the S tuples by looking at the tag of the value: a value with tag 1 is an R tuple and a value with tag 2 is an S tuple. Finally, the R tuples are combined with the S tuples using a nested loop, since these are the tuples that correspond to the same join key and must be joined together.

The GroupByJoin operation is based on the reduce-side join but it uses special replication and partitioning techniques, as required by the SUMMA algorithm. It distributes the data to the worker nodes in the form of a $n * m$ grid of partitions, where each partition contains only those rows from X and those columns from Y needed to compute a single partition of the resulting matrix.

```
 1  mapLeft ( x ):
 2    for each i in 0..m−1
 3        emit ( ((hashCode(gx(x)) % n)∗m+i, jx(x), 1), (1,x) )
 4
 5  mapRight ( y ):
 6    for each i in 0..n−1
 7        emit ( ((hashCode(gy(y)) % m)+m∗i, jy(y), 2), (2,y) )
 8
 9  reduce ( ( partition ,joinkey ,tag), values ):
10    if ( partition != current_partition )
11       flush(H)
12       current_partition ← partition
13    xs ←∅
14    // (1,x) tuples arrive before (2,y) tuples in values
15    for each leading (1,x) tuple in values
16       insert x into xs
17    for each (2,y) tuple in the rest of values
18       for each x in xs
19          key ← (gx(x),gy(y))
20          if (H[key] is null)
21             H[key] ← zero
22          H[key] ← acc( (x,y), H[key] )
23
24  cleanup ( ):
25    flush(H)
```

Fig. 3. Map-Reduce pseudo-code for GroupByJoin(jx, jy, gx, gy, acc, zero, h, X, Y)

Fig. 3 shows the pseudo-code for the implementation of GroupByJoin in Map-Reduce. The rest of this section explains the code in detail. Each of the $n * m$ partitions is ideally assigned to a single worker node (a reducer), but in general each reducer may receive multiple partitions, and each partition may contain multiple groupings. Each grouping is handled separately by the reduce method. Similar to a regular reduce-side join on Map-Reduce, our GroupByJoin uses two mappers, mapLeft and mapRight, one for each input dataset, X and Y. Unlike a regular reduce-side join though, the mapLeft replicates each tuple m times while the mapRight mapper replicates each tuple n times, all under different join keys (lines 1-3 and 5-7 in Fig. 3). Both mappers emit (key,value) pairs. A mapper value takes the form (tag,data), where data is the input data and tag is the source number 1 or 2, to specify the input source (X or Y). A mapper key on the other hand is a triple (partition,joinkey,tag), where partition is one of the $n * m$ partitions, and joinkey is the join key value, jx(x) for the left mapper and jy(y) for the right mapper. More specifically, the partition number of a partition (i, j) in the grid of $n * m$ partitions is equal to $i * m + j$ (based on row-major numbering in the partition grid). The two mappers replicate the X and Y values under different partition numbers. A value x $\in$ X is sent to all the row partitions (gx(x) $\bmod n$, $*$) (n partitions) while a value y $\in$ Y is sent to all the column partitions ($*$, gy(y) $\bmod m$) (m partitions). Hadoop Map-Reduce supports custom partitioning, grouping, and sorting functions that control the shuffling of the map results to the reducers. These custom functions are adjusted in such a way that this Map-Reduce job implements the SUMMA algorithm efficiently. They are directly derived from the partition, joinkey, and tag components of the mapper key. In our Hadoop Map-Reduce implementation,

- the partition function returns the partition value of the mapper key,
- the grouping function returns the pair (partition,joinkey), and
- the sorting is based on partition (major order), joinkey (minor order), and tag (subminor order).

Each call to the reduce method is associated with a certain partition number and a certain joinkey, by means of the grouping function. Consequently, the values parameter of the reduce method contains all tuples from both X and Y whose join key is equal to joinkey. For matrix multiplication, when X is an $N * K$ matrix and Y is an $K * M$ matrix, the size of values will be $N/n + M/m$, one column from the X horizontal partition and one row from the Y vertical partition (Fig. 4).

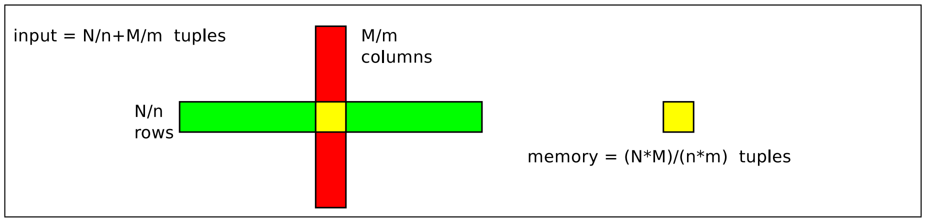

Fig. 4. Each reducer receives $N/n + M/m$ tuples per call and stores $(N * M)/(n * m)$ tuples

In Map-Reduce, the values argument of the reduce method can only be accessed once in a stream-like fashion, because these values are directly read from the merged runs stored on the reducer's local disk. But if we sort the values so that the X tuples appear before the Y tuples, then it would not be necessary to store the values in a vector to do the nested-loop join between the matched X and Y tuples. The reduce method in Fig. 3 stores the X tuples in a vector xs (lines 13-16) and then does a cross product between xs and the Y tuples (lines 17-22), followed by a group-by with aggregation. This is possible because the X tuples come before the Y tuples, by means of the sorting function, and all results in a group are guaranteed to occur within the same partition. The reduce method uses a key-value map H (implemented as a hash table) that holds the partial results of the aggregation after the join and group-by. Each reducer must have its own copy of H (a static variable in Java). It has to be maintained across multiple calls to the reduce method within the same partition. The body of the nested-loop join (lines 19-22) makes partial contributions to the group-by with aggregation by using the group-by key (gx(x),gy(y)) to accumulate the new pair (x,y) to the existing value associated with this key using the accumulator acc.

The code in lines 10-12 in Fig. 3 checks whether the reducer has seen the end of a partition. Function flush(H) applies the function h to each key-value pair in H and emits the results to the output file:

```
flush ( H ):
  for each (key,value) in H
    emit h(key,value)
  clear H
```

Given that a reducer may be assigned multiple partitions, the results of processing each partition are emitted by flush(H) at the end of each partition (when the partition number changes). This is possible because the input pairs arrive to a reducer sorted by a partition number (major order). The hash table H needs to be maintained across a partition only since each partition contains all the required data to complete the join. Thus, when the current partition is finished, the hash table H is flushed and the next partition is ready to be processed with an empty H. Thus, H should be large enough to fit the largest resulting partition. Ideally, if there is no data skew, the size of H should be the total size of the resulting dataset divided by $n * m$. In our matrix multiplication case, when X is an $N * K$ matrix and Y is an $K * M$ matrix, the size of the hash table H should be $(N * M)/(n * m)$ (see Fig. 4). Larger n and m requires more data replication and thus more network traffic, while smaller n and m may require a hash table that is too large to fit in memory. Thus, we must select the smallest n and m so that H can fit in memory. If there is available memory at each reducer to fit $\mathcal{T}$ tuples, then $(N * M)/(n * m) = \mathcal{T}$. Our goal is to minimize data replication, which is equal to $N * K * m + K * M * n$, which is equivalent to minimizing $N/n + M/m$ (if we divide by the constant $K * n * m$). Given that $(N/n) * (M/m) = \mathcal{T}$, the minimum $N/n + M/m$ is derived when $N/n = M/m = \sqrt{\mathcal{T}}$. That is, given the amount of available memory at each reducer ($\mathcal{T}$ tuples), the optimal grid size that minimizes the amount of shuffling is $n = N/\sqrt{\mathcal{T}}$ and $m = M/\sqrt{\mathcal{T}}$.

7 The Implementation of GroupByJoin in Spark and Flink

In addition to Map-Reduce, the GroupByJoin operation has been implemented on Apache Spark [9] and Apache Flink [2] since these frameworks too are supported by the MRQL evaluation engine. These frameworks provide a similar API, although they have different performance characteristics. Hence, we describe the GroupByJoin implementation in Spark only but we evaluate GroupByJoin on both Spark and Flink.

The central Spark data abstraction is the Resilient Distributed Dataset (RDD), which is an immutable collection of values partitioned across multiple machines in a cluster.

These RDD partitions are typically stored in the memory of the compute nodes. An RDD is resilient to failures; a lost RDD partition can be reconstructed from its input RDD partitions (from the RDDs that were used to compute this RDD). The evaluation of RDD transformations in Spark is deferred until an action is encountered that brings data to the master node or stores the data into a file. Spark collects the deferred transformations into a DAG and divides them into subsequences, called stages. Data shuffling occurs between stages, while transformations within a stage are combined into a single RDD transformation. Despite this inter-stage optimization, Spark cannot perform non-trivial optimizations, such as moving a filter operation before a join, because the functional arguments of the RDD operations are written in the host language and cannot be analyzed for code patterns at run-time. Spark has addressed this shortcoming by providing two additional APIs, called DataFrames and Datasets [1]. A Dataset combines the benefits of RDD (strong typing and powerful higher-order operations) with Spark SQL's optimized execution engine. A DataFrame is a Dataset organized into named columns as in a relational table. SQL queries in DataFrames are translated and optimized to RDD workflows at run-time using the Catalyst architecture.

```
 1  def groupByJoin ( jx, jy, gx, gy, acc, zero, h, X, Y )
 2    = { val XS = X.flatMap{ x => (0 until m).map{
 3                              i => ( (gx(x).hashCode() % n)*m+i, x ) } }
 4        val YS = Y.flatMap{ y => (0 until n).map{
 5                              i => ( (gy(y).hashCode() % m)+m*i, y ) } }
 6        XS.cogroup(YS,n*m)
 7          .flatMap{ case (p,(xs,ys))
 8                    => val H = new HashMap
 9                       val hx = new MultiMap
10                       xs.foreach{ x => hx.addBinding(jx(x),x) }
11                       ys.foreach{
12                         y => hx(jy(y)).foreach{
13                                x => val key = (gx(x),gy(y))
14                                     if (!H.contains(key))
15                                        H += (( key, zero ))
16                                     H += (( key, acc( (x,y), H(key) ) ))
17                              }
18                       }
19                       H.map{ case (k,v) => h(k,v) }
20                  }
21    }
```

Fig. 5. Spark Scala code for GroupByJoin

Fig. 5 shows the code of the GroupByJoin method. The Scala types have been omitted for brevity. As it was done in the mapLeft and mapRight methods in Fig. 3, the code in lines 2-5 in Fig. 5 creates the RDDs XS and YS that replicate the elements of X m times and the elements of Y n times into a grid of $n*m$ partitions. (In Scala, X.flatmap(f) applies the function f to every element of the sequence X and concatenates the resulting lists into one list; while (0 until m) creates a new list $[0, \ldots, m-1]$.) Unlike mapLeft and mapRight though, Spark does not need to use a tag to separate the X from the Y values because it provides a special operation for reduce-side join, called cogroup. The n*m argument of cogroup indicates the number of reducers, which is also the number of the output partitions. Furthermore, the keys in XS and YS do not depend on the join keys jx and jy. This means that the join performed by cogroup is over the partition number only. But, as in Map-Reduce, each partition is self-sufficient to create its join result. That is, after the cogroup, there will be $n * m$ pairs (p,(xs,ys)), for each different partition p, ideally one pair for each reducer. This is not problematic because Spark keeps each RDD partition in memory in most cases. The flatMap in lines 7-20 performs a hash join between xs and ys. The partial results of the group-by aggregation are maintained in the hash table H, in the same way it was done in the reducer in Fig. 3. The flatMap functional constructs the built hash table hx for the input xs using a MultiMap (line 9), which is a hash table that maps a key to a set of values. Then, the foreach expression in lines 11-18 performs a loop over ys and probes the hash table hx using the join key jy(y), and in general returns a set of x values. Finally, the probed x is used along with y to add a new entry to the aggregation table with key (gx(x),gy(y)) (lines 14-16). After all values are aggregated, the result of the groupByJoin is calculated by applying the header function h to every value in the hash table H (line 19).

```
 1  val XS = X.flatMap{ case x@(v,i,j)
 2                        => (0 until m).map{ k => ( (i % n)*m+k, x ) } }
 3  val YS = Y.flatMap{ case y@(v,i,j)
 4                        => (0 until n).map{ k => ( (j % m)+m*k, y ) } }
 5  XS.cogroup(YS,n*m)
 6    .flatMap{ case (k,(xs,ys))
 7               => val H = new HashMap
 8                  val hx = new MultiMap
 9                  xs.foreach{ case x@(_,_,k)
10                               => hx.addBinding(k,x) }
11                  ys.foreach{ case (y,k,j)
12                               => hx(k).foreach{ case (x,i,_)
13                                                   => val key = (i,j)
14                                                      if (!H.contains(key))
15                                                         H += (( key, 0.0 ))
16                                                      H += (( key, x*y+H(key) ))
17                                                }
18                            }
19                  H.map{ case ((i,j),v) => (v,i,j) }
20             }
```

Fig. 6. Spark Scala code for the SUMMA Algorithm for $X \times Y$

To illustrate the GroupByJoin functionality better, the Spark code in Fig. 6 computes the matrix multiplication $X \times Y$ using the SUMMA algorithm. It is an instance of the GroupByJoin algorithm in Fig. 3. Every tuple (v,i,j) in X is replicated m times across all columns k in the row i of the $n * m$ grid, which corresponds to the partition (i % n)*m+k (lines 1-2). On the other hand, every tuple (v,i,j) in Y is replicated n times across all rows k in the column j of the grid, which corresponds to the partition (j % m)+m*k (lines 3-4). The XS.cogroup(YS,n*m) operation in line 5 performs a distributed join over $n * m$ partitions using the join keys (partition numbers) computed in lines 1-4. That is, the number of pairs returned by this join is $n*m$, one for each partition. Given that an RDD is in the distributed memory across the worker notes, each pair (k,(xs,ys)) corresponds to a different reducer, which ideally is a different worker node. The xs and ys collections have all the required data to derive the join result that corresponds to this partition. This computation is done in the hash join with group-by and aggregation in lines 7-19. The hash table H contains a partition of the resulting matrix and is populated during the final aggregation. The hash table hx is the built table from xs and is populated before the join (lines 9-10) using the key k for a tuple (x,i,k) in xs. The actual hash join is done in lines 11-18 by scanning ys (line 11) and probing the built hash table hx (line 12) using the hash key k from the tuple (y,k,j). The code in lines 13-16 updates the result table H by adding the product x*y to the exiting value. Finally, the code in line 19 converts the entries in the result table H into matrix values for the resulting matrix.

8 Translating Queries to GroupByJoin Operations

Based on the discussion in the Introduction, it would be hard to use source-to-source transformations to translate queries, such as matrix multiplication and shortest distance, to an algebraic form that contains GroupByJoin operations, because query syntax may take many different equivalent forms, which have to be recognized by these source-to-source transformations. Instead, our approach is to translate queries to their default algebraic forms and then normalize and rewrite these forms using algebraic rules.

8.1 The MRQL Algebra

The MRQL algebra used in this section has already been described in our previous work [18,17]. The most important algebraic operation in the MRQL algebra is cMap (also known as concat-map or flatten-map in functional programming languages), which generalizes the select, project, join, and unnest operators of the nested relational algebra. Given two arbitrary types α and β, the operation $\mathrm{cMap}(f, X)$ maps a bag X of type $\{\alpha\}$ to a bag of type $\{\beta\}$ by applying the function f of type $\alpha \rightarrow \{\beta\}$ to each element of X, yielding one bag for each element, and then by merging these bags to form a single bag of type $\{\beta\}$. Using a set former notation on bags, it is expressed as follows:

$$\mathrm{cMap}(f, X) \;=\; \{\, z \,|\, x \in X,\, z \in f(x)\,\} \tag{4}$$

Given an arbitrary type κ that supports value equality ($=$), an arbitrary type α, and a bag X of type $\{(\kappa, \alpha)\}$, the operation $\mathrm{groupBy}(X)$ groups the elements of the bag X by their first component and returns a bag of type $\{(\kappa, \{\alpha\})\}$, where the first component of each tuple is a unique group-by key and the second is the group (a bag) that contains all values that correspond to this key. For example, groupBy({(1,“A”), (2,“B”), (1,“C”)}) returns {(1,{“A”,“C”}), (2,{“B”})}. Although any join $X \bowtie_{j_x(x)=j_y(y)} Y$ can be expressed as a nested cMap, to facilitate the creation of physical plans for joins, the MRQL algebra provides a special join operator:

$$\begin{aligned}&\mathrm{join}(j_x, j_y, h, X, Y)\\ &= \{\, h(x,y) \,|\, x \in X,\, y \in Y,\, j_x(x) = j_y(y) \,\}\\ &= \mathrm{cMap}(\lambda x.\, \mathrm{cMap}(\lambda y.\, \mathbf{if}\, j_x(x) = j_y(y)\, \mathbf{then}\, \{h(x,y)\}\, \mathbf{else}\, \{\,\},\, Y),\, X)\end{aligned}$$

where an anonymous function $\lambda x.\, e$ specifies a unary function (a lambda abstraction) f such that $f(x) = e$. This operation joins two bags, X of type $\{\alpha\}$ and Y of type $\{\beta\}$, using the join functions, j_x of type $\alpha \to \kappa$ and j_y of type $\beta \to \kappa$, and combines the joining values using the function h of type $(\alpha, \beta) \to \gamma$, deriving a bag of type $\{\gamma\}$. Finally, aggregations are captured by the operation $\mathrm{reduce}(acc, zero, X)$, which reduces the elements of a bag X of type $\{\alpha\}$ into a value of type β, using an accumulator acc of type $(\alpha, \beta) \to \beta$ and a zero value $zero$ of type β. For example, $\mathrm{reduce}(\, \lambda(x,s).\, x + s,\, 0,\, \{1,2,3\}\,) = 6$.

The algebraic terms derived from MRQL queries can be normalized using rewrite rules, such as:

$$\mathrm{cMap}(f, \mathrm{cMap}(g, S)) \to \mathrm{cMap}(\lambda x.\, \mathrm{cMap}(f, g(x)), S) \tag{5}$$

which fuses two cascaded cMaps into a nested cMap, thus avoiding the construction of the intermediate bag. This rule can be proven directly from the cMap definition in Equation (4):

$$\begin{aligned}\mathrm{cMap}(f, \mathrm{cMap}(g, S)) &= \{\, z \,|\, w \in \{\, y \,|\, x \in S,\, y \in g(x) \,\},\, z \in f(w) \,\}\\ &= \{\, z \,|\, x \in S,\, y \in g(x),\, z \in f(y) \,\}\\ &= \{\, z \,|\, x \in S,\, z \in \{\, w \,|\, y \in g(x),\, w \in f(y) \,\} \,\}\\ &= \mathrm{cMap}(\lambda x.\, \mathrm{cMap}(f, g(x)), S)\end{aligned}$$

In addition, a cMap can be fused with a join resulting to a new join:

$$\begin{aligned}&\mathrm{join}(\, j_x,\, j_y,\, h,\, X,\, \mathrm{cMap}(\lambda y.\, \{f(y)\}, Y)\,)\\ &\qquad \to\ \mathrm{join}(\, j_x,\, \lambda y.\, j_y(f(y)),\, \lambda(x,y).\, h(x, f(y)),\, X,\, Y\,) \qquad (6)\\ &\mathrm{cMap}(\lambda v.\, \{f(v)\},\, \mathrm{join}(\, j_x,\, j_y,\, h,\, X,\, Y\,))\\ &\qquad \to\ \mathrm{join}(\, j_x,\, j_y,\, \lambda(x,y).\, f(h(x,y)),\, X,\, Y\,)) \qquad (7)\end{aligned}$$

8.2 Translating Algebraic Terms to GroupByJoin Operations

In an earlier work [18], we have presented a general framework for translating MRQL queries to algebraic terms. This framework uses novel optimization techniques to map these algebraic forms to efficient workflows of physical plan operations that are specific to the underlying distributed platform.

The framework described in this paper extends our earlier work [18] by introducing a new algebraic operation, GroupByJoin, and by providing rules for deriving GroupByJoin operations from algebraic forms. By default, the generic MRQL query, Query 4:

```
select h( k, reduce(acc,zero,z) )
  from x in X, y in Y, z = (x,y)
 where jx(x) = jy(y)
 group by k: ( gx(x), gy(y) )
```

is translated to the following algebraic form:

```
cMap( λ(k,s).{ h(k,reduce(acc,zero,s)) },
      groupBy( join( jx, jy,
                     λ(x,y).( (gx(x),gy(y)), (x,y) ),
                     X, Y ) ) )
```

which joins X and Y via the join keys jx and jy (i.e., a value x in X is joined with a value y in Y if jx(x)=jy(y)), and produces pairs (key, (x,y)), where key is the group-by key (gx(x),gy(y)). The group-by operation collects all (x,y) pairs that are associated with the same key into a group and the outer operation, cMap, accumulates each group s to a single value by reducing this group using reduce(acc,zero,s) and then applying the result function h. These algebraic forms can be derived from queries that may look very different from the previous query, since algebraic forms are normalized to a canonical form and, thus, queries that correspond to a GroupByJoin will always be translated to the same canonical algebraic form. The GroupByJoin operation is derived with the help of the following rule:

```
cMap( λ(k,s).{ h(k,reduce(acc,zero,s)) },
      groupBy( join( jx, jy,
                     λ(x,y).( (gx(x),gy(y)), (x,y) ),
                     X, Y ) ) )
   → GroupByJoin( jx, jy, gx, gy, acc, zero, h, X, Y )
```

which rewrites the previously derived equi-join/group-by algebraic form to a GroupByJoin. Note that the variables in this rule, such as x, can be matched with any term, while term functions, such as gx(x), are terms that contain their arguments as subterms. For example, the term function gx(x) matches any term that depends on the variable x. The pattern that represents a join followed by the groupBy in the previous rule matches most terms with a groupBy after join. The cMap functional argument that evaluates the aggregation reduce(acc,zero,s) though is more restrictive as it requires that each group s formed after groupBy be reduced by an aggregation that matches reduce(acc,zero,s), for some acc and zero.

For example, consider the MRQL query, Query 2, that captures matrix multiplication $X \times Y$:

```
select ( sum(z), i, j )
  from (x,i,k) in X, (y,k,j) in Y, z = x*y
 group by i, j
```

This query is translated into the following algebraic form:

```
cMap( λ((i,j),s).{( reduce(λ(v,c).c+v, 0, s), i, j )},
      groupBy( join( λ(x,i,k).k, λ(y,k,j).k,
                     λ((x,i,k),(y,l,j)).( (i,j), x*y ),
                     X, Y ) ) )
```

which joins the matrices X and Y so that a matrix element (x,i,k) in X is joined with a matrix element (y,k',j) if k=k'. These joined values contribute the key-value pair ((i,j), x*y) to the join result, where the key (i,j) is used as the group-by key by the groupBy operation. Then, all x*y values that correspond to the same group key (i,j) form a group s, which is reduced by the outer cMap by calculating the sum of all these values using reduce(λ(v,c). c+v, 0, s) (i.e., it reduces s using the accumulator + and an initial value 0). This algebraic form matches the left-hand side of our translation rule, which rewrites the algebraic form to the term:

```
GroupByJoin( λ(x,i,k). k, λ(y,k,j). k, λ(x,i,k). i, λ(y,l,j). j, λ(v,c). c+v, 0, λ((i,j),c). {(c,i,j)}, X, Y )
```

which is equivalent to the term derived in Section 5.

8.3 Optimization of GroupByJoin Operations

Optimization of matrix operations in our framework is done before algebraic terms are transformed to GroupByJoin operations using the existing MRQL optimizer. That is, join, groupBy, and cMap operations are fused together using the rewrite rules presented in Section 8.1, and then the resulting terms are transformed to GroupByJoin operations using the rewrite rule presented in Section 8.2.

For example, consider the composition of matrix multiplication with matrix transpose $X \times Y^T$, where matrix transpose Y^T is expressed as follows in MRQL:

```
    select (y,j,i) from (y,i,j) in Y
```

which flips the two indexes, thus transposing the matrix Y. This query is translated to the following algebraic form:

```
cMap( λ(y,i,j). { (y,j,i) }, Y )
```

Therefore, from Section 8.2, the composition $X \times Y^T$ is:

```
    cMap( λ((i,j),s). {( reduce(λ(v,c). c+v, 0, s), i, j )},
            groupBy( join( λ(x,i,k). k, λ(y,k,j). k,
                           λ((x,i,k),(y,l,j)). ( (i,j), x*y ),
                           X,
                           cMap( λ(y,i,j). {(y,j,i)}, Y ) ) ) )
```

(The only difference from the matrix multiplication algebraic form in Section 8.2 is the last line that transposes the matrix Y.) Using Equation 6, the join is fused with the inner cMap giving:

```
cMap( λ((i,j),s). {( reduce(λ(v,c). c+v, 0, s), i, j )},
        groupBy( join( λ(x,i,k). k,
                       λ(y,j,k). k,
                       λ((x,i,k),(y,j,l)). ( (i,j), x*y ),
                       X, Y ) ) )
```

This term is translated to the following algebraic operation:

```
GroupByJoin( λ(x,i,k). k, λ(y,j,k). k, λ(x,i,k). i, λ(y,j,l). j, λ((x,y),c). c+x*y, 0, λ((i,j),c). (c,i,j),
             X, Y )
```

which combines matrix multiplication with matrix transpose into a single GroupByJoin operation.

9 Performance Evaluation

```
macro transpose ( X ) {          /* matrix transpose */
  select (x,j,i)
    from (x,i,j) in X
};
macro multiply ( X, Y ) {        /* matrix multiplication */
  select (sum(z),i,j)
    from (x,i,k) in X, (y,k,j) in Y, z = x*y
   group by (i,j)
};
macro mult ( a, X ) {            /* multiplication by a constant */
  select ( a*x, i, j )
    from (x,i,j) in X
};
macro Cadd ( X, Y ) {            /* cell-wise addition */
  select ( x+y, i, j )
    from (x,i,j) in X, (y,i,j) in Y
};
macro Csub ( X, Y ) {            /* cell-wise subtraction */
  select ( x-y, i, j )
    from (x,i,j) in X, (y,i,j) in Y
};
macro factorize ( R, Pinit, Qinit ) {    /* matrix factorization */
  repeat (E,P,Q) = (R,Pinit,Qinit)
    step ( Csub(R,multiply(P,transpose(Q))),
           Cadd(P,mult(a,Csub(mult(2,multiply(E,transpose(Q))),mult(b,P)))),
           Cadd(Q,mult(a,Csub(mult(2,multiply(E,transpose(P))),mult(b,Q)))) )
   limit 10
};
```

Fig. 7. Matrix Factorization using Gradient Descent in MRQL

The platform used for our evaluations is a small cluster of 9 nodes, built on the Chameleon cloud computing infrastructure, `www.chameleoncloud.org`. This cluster consists of nine m1.medium instances running Linux, each one with 4GB RAM and two VCPUs at 2.3GHz. For our experiments, we used Hadoop 2.6.0 (Yarn), Spark 2.1.0, Flink 1.0.3, and MRQL 0.9.8. The cluster frontend was used exclusively as a Name-Node and ResourceManager, while the remaining 8 compute nodes were used as DataNodes and NodeManagers. There was a total of 16 VCPUs and a total of 28.5GB of RAM available for compute tasks. The HDFS file system was formatted with the block size set to 128MB and the replication factor set to 3. Each dataset used in our experiments was stored in a single HDFS file. Our experiments were run using MRQL on three evaluation modes: Hadoop Map-Reduce mode, Spark mode, and Flink mode. Each experiment was evaluated 5 times under the same data and configuration parameters. Each data point in the plots in Figs. 8, 9, and 10 represents the mean value of 5 experiments while the vertical error bar at each line point represents the minimum and maximum values among these 5 experiments.

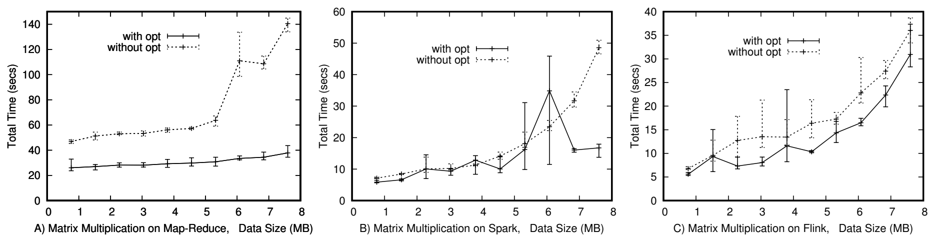

Fig. 8. Evaluation of Matrix Multiplication on A) Map-Reduce, B) Spark, and C) Flink

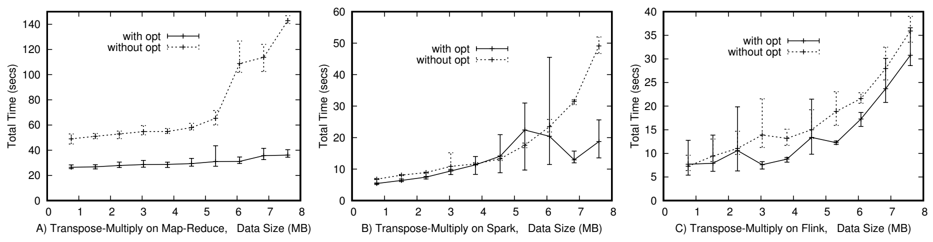

Fig. 9. Evaluation of Multiply-Transpose on A) Map-Reduce, B) Spark, and C) Flink

We have experimentally validated the effectiveness of our methods for three MRQL queries, based on operations that are defined in Fig. 7: 1) a matrix multiplication query, multiply(X,Y), 2) a simple query multiply-transpose, multiply(X,transpose(Y)), and 3) a matrix factorization query using gradient descent, factorize(R,P,Q).

The matrices X and Y used in the evaluation of the first two queries (matrix multiplication and multiply-transpose) were square matrices with dimensions $(i * 40) * (i * 40)$ and size $i * 0.38$ MB, for $i \in [1, 10]$. Consequently, the largest matrix used has dimensions 400*400 while the maximum total input size is 3.8*2=7.2 MB. The type of the matrix elements is (float, int, int), where the two integers are the row and column indexes, and the float is the value. Each matrix used in our experiments was dense (i.e, all matrix elements were provided) and filled with random values between 0.0 and 10.0,

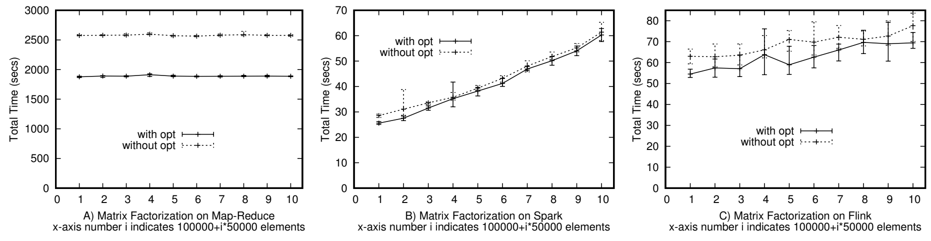

Fig. 10. Evaluation of Matrix Factorization on A) Map-Reduce, B) Spark, and C) Flink

and the matrix elements were placed in random order. Fig. 8 shows the results of evaluating the matrix multiplication query on Map-Reduce, Spark, and Flink, with and without using our optimization framework. That is, the "with opt" line is from evaluating the matrix multiplication query using a GroupByJoin operation and the "without opt" line is from the straightforward evaluation plan that consists of a join followed by a group-by with aggregation. In the latter case, the join is a simple reduce-side join. We can see that the performance improvement for Map-Reduce is more pronounced than that for Spark and Flink, especially for matrices larger than 280*280 (which corresponds to a total size of 5.32MB). We get similar performance results for the multiply-transpose query in Fig. 9. The results for multiply-transpose are very similar to those for matrix multiplication because, the optimized version of the former is exactly the same as the optimized version of the latter query, while the non-optimized versions differ only in the extra map needed for the transpose operation, which can be performed efficiently by all three frameworks since it does not require any data shuffling. We also believe that, in the case Map-Reduce, the reason for the peak at 5MB is that the size of the intermediate data between the reduce-side join and the group-by with aggregation has reached a critical point where sorting and merging must be external (on the mappers local disk), rather than in-memory. More specifically, since the intermediate data produced by the join between two matrices of size N^2 is of size N^3, the group-by workload after the join will be very high. The group-by though is implemented by partitioning the data and sorting the partitions at the mappers, before the data are shuffled and merged at the reducers. When the data are larger than the amount of memory allocated for sorting at a mapper, the mapper will necessarily use external sorting, which may explain the peak in the diagram.

Fig. 10 shows the results of evaluating matrix factorization. Given a matrix R, our matrix factorization query in Fig. 7 calculates the error matrix $E = R - P \times Q^T$ and the factor matrices P and Q, so that R is approximately equal to $P \times Q^T$. For our experiments, we set this query to iterate 10 times and used the learning rate $a = 0.002$ and the normalization factor $b = 0.02$. The matrix to be factorized, R, was an $n \times m$ sparse matrix with random integer values between 1 and 5 in which only the 10% of the elements were provided (the rest were implicitly zero). The size of m was always kept equal to $10 * n$, while $n * m$ was set to $100000 + i * 50000$ elements, for $i \in [0, 9]$. That is, $n * m$ took the following values: 100*1000, 122*1220, 141*1410, 158*1580, 173*1730, 187*1870, 200*2000, 212*2120, 223*2230, 234*2340. The initial factor matrices, Pinit and Qinit, had sizes $n * k$ and $m * k$, respectively, where $k = 10$ for all experiments (which is a low rank), and all their elements were initialized to 2.5. Fig. 10 shows the results of evaluating the matrix factorization query on Map-Reduce, Spark, and Flink, with and without optimization. We can see that the improvement for the Map-Reduce evaluation is substantial (the optimized query is about 27% faster than the non-optimized one) mostly because the benefits of all the optimizations used in MRQL are accumulated and repeated at each iteration step. The results from the Map-Reduce evaluation look very similar for different data sizes (100K through 550K tuples) because all matrices (including the intermediate results) are split into 16 partitions in the HDFS (one for each compute node) and each partition can fit into one HDFS block (128MBs) regardless of its size.

10 Conclusion and Future Work

We have presented a general framework for optimizing SQL-like queries that capture array-based computations on sparse arrays. In contrast to related work, we do not provide a library of predefined array operations. Instead, we are letting programmers express their array operations using normal SQL-like syntax, but, at the same time, we provide an optimization framework that translates these queries into efficient distributed array operations. That way, we are able to achieve inter-operation optimization that would be infeasible if these operations were expressed as black boxes. Our framework has been tested on three popular Big Data platforms that exhibit different functionality and performance characteristics. As a future work, we will apply our framework to more Big Data platforms, such as Apache Storm. In addition, we are planning to capture more data-parallel algorithms that have already been used in high-performance computing to implement matrix operations or other general linear algebra parallel algorithms. Furthermore, we are planning to experiment with a generalization of the group-by-join algorithm to capture n-way matrix multiplications. That is, instead of using chains of binary matrix operations, complex terms that involve multiple matrix operations would be translated into a single n-way group-by-join operation thus avoiding creating the intermediate matrices between the binary matrix operations. To accomplish this, the grid of partitions will have to be multi-dimensional, where each dimension corresponds to a different matrix. Then, the replication of the elements of a matrix will be done across the rest of the $n - 1$ grid dimensions. That way, any complex term that consists of multiple matrix multiplications and other operations (such as, cell-wise operations and transpose), such as the body of matrix factorization, will be fused to a single n-ary group-by-join, thus achieving optimal performance.

Acknowledgments: Our performance evaluations were performed at the Chameleon cloud computing infrastructure, `www.chameleoncloud.org`, supported by NSF.

References

1. M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark.

In *SIGMOD'15*.
2. Apache Flink. `http://flink.apache.org/`. 2018.
3. Apache Hadoop. `http://hadoop.apache.org/`. 2018.
4. Apache Hama. `http://hama.apache.org/`. 2018.
5. Apache Hive. `http://hive.apache.org/`. 2018.
6. Apache Giraph. `http://giraph.apache.org/`. 2018.
7. GraphX: Apache Spark's API for Graphs and Graph-Parallel Computation. `https://spark.apache.org/graphx/`. 2018.
8. Apache MRQL (incubating). `http://mrql.incubator.apache.org/`. 2018.
9. Apache Spark. `http://spark.apache.org/`. 2018.
10. D. Battre, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *1st ACM Symposium on Cloud computing (SOCC'10)*, pp 119–130, 2010.
11. J. Buck, N. Watkins, J. Lefevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. A. Brandt. SciHadoop: Array-based Query Processing in Hadoop. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.
12. R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *Proceedings of the VLDB Endowment (PVLDB)*, 1(2):1265–1276, 2008.
13. A. Das, F.N. Afrati, S. Salihoglu, and J.D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. In *VLDB'13*.
14. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04*.
15. J. Fan, *et al*. The case against specialized graph analytics engines. In *CIDR*, 2015.
16. L. Fegaras. A Query Processing Framework for Array-Based Computations. In *27th International Conference on Database and Expert Systems Applications (DEXA)*, 2016.
17. L. Fegaras. An Algebra for Distributed Big Data Analytics. Journal of Functional Programming, Special issue on Programming Languages for Big Data, Volume 27, 2017.
18. L. Fegaras, C. Li, and U. Gupta. An Optimization Framework for Map-Reduce Queries. In *EDBT'12*.
19. L. Fegaras, C. Li, U. Gupta, and J. J. Philip. XML Query Optimization in Map-Reduce. In *International Workshop on the Web and Databases (WebDB)*, 2011.
20. L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. In *International Conference on Management of Data (SIGMOD)*, pp 47–58, 1995.
21. L. Fegaras and D. Maier. Optimizing Object Queries Using an Effective Calculus. In *ACM Transactions on Database Systems (TODS)*, 25(4):457–516, 2000.
22. M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the HDF5 technology suite and its applications. In *EDBT/ICDT Workshop on Array Databases*, 2011.
23. A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a High-Level Dataflow System on top of Map-Reduce: the Pig Experience. In *Proceedings of the VLDB Endowment (PVLDB)*, 2(2):1414-1425, 2009.
24. R. A. Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. In *Concurrency: Practice and Experience*, 9(4):255–274, April 1997.
25. Y. Geng, X. Huang, M. Zhu, H. Ruan, and G. Yang. SciHive: Array-based query processing with HiveQL. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications (Trustcom)*, 2013.
26. A. Jindal, *et al*. Vertexica: Your Relational Friend for Graph Analytics! In *PVLDB*, 7(13): 1669–1672, 2014.

27. A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative Machine Learning on MapReduce. In *IEEE International Conference on Data Engineering (ICDE)*, 2011.
28. M. Isard and Y. Yu. Distributed Data-Parallel Computing Using a High-Level Programming Language. In *ACM SIGMOD International Conference on Management of Data*, pp 987–994, 2009.
29. Y. Koren, R. Bell, and C. Volinsky. Matrix Factorization Techniques for Recommender Systems In *IEEE Computer*, August 2009.
30. T. Kraska, A. Talwalkar, J. Duchi, R. Griffith, M. Franklin, and M.I. Jordan. MLbase: A Distributed Machine Learning System. In *Conference on Innovative Data Systems Research*, 2013.
31. J. Lin and C. Dyer. Data-Intensive Text Processing with MapReduce. Morgan & Claypool Publishers, 2010.
32. Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. In *VLDB'12*.
33. G. Malewicz, M. H. Austern, A. J.C Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a System for Large-Scale Graph Processing. In *ACM SIGMOD International Conference on Management of Data*, pp 135–146, 2010.
34. X. Meng, J. Bradley, B. Yavuz, *et al*. MLlib: Machine Learning in Apache Spark. In *Journal of Machine Learning Research*, 17:1-7, 2016.
35. NetCDF: Network Common Data Form.
`https://www.unidata.ucar.edu/software/netcdf/`.
36. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-Foreign Language for Data Processing. In *ACM SIGMOD International Conference on Management of Data*, 2008.
37. S. Papadopoulos, K. Datta, S. Madden, and T. Mattson. The TileDB array data storage manager. In *PVLDB*, 10(4), 2016.
38. E. Soroush, M. Balazinska, and D. Wang. ArrayStore: A Storage Manager for. Complex Parallel Array Processing. In *ACM SIGMOD International Conference on Management of Data*, 2011.
39. E. Soroush, M. Balazinska, S. Krughoff, and A. Connolly. Efficient Iterative Processing in the SciDB Parallel Array Engine. In *27th International Conference on Scientific and Statistical Database Management (SSDBM)*, 2015.
40. A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat. M3R: Increased performance for in-memory Hadoop jobs In *VLDB'12*
41. The SciDB Development Team. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *ACM SIGMOD International Conference on Management of Data*, 2010.
42. A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Antony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a Warehousing Solution over a Map-Reduce Framework. In *Proceedings of the VLDB Endowment (PVLDB)*, 2(2):1626–1629, 2009.
43. A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive: A Petabyte Scale Data Warehouse Using Hadoop. In *IEEE International Conference on Data Engineering (ICDE)*, pp 996–1005, 2010.
44. L. G. Valiant. A bridging model for parallel computation. In *CACM*, 33(8):103-111, August 1990.
45. Y. Wang, W. Jiang, and G. Agrawal. SciMATE: A novel MapReduce-like framework for multiple scientific data formats. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2012.
46. Y. Yu, *et al*. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.