A Query Processing Framework for Array-Based Computations

Leonidas Fegaras(✉)

University of Texas at Arlington, Arlington, USA
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 evaluate the effectiveness of our framework by measuring the performance improvement of matrix factorization when evaluated with inter-operator optimization.

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 these frameworks, the Map-Reduce programming model [3] 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. Currently, most programmers prefer to use a higher-level declarative language to code their Map-Reduce applications, such as Apache Hive [11] and PigLatin [18], instead of coding them directly in an algorithmic language, such as Java. For instance, Hive is

S. Hartmann and H. Ma (Eds.): DEXA 2016, Part I, LNCS 9827, pp. 240–254, 2016.
DOI: 10.1007/978-3-319-44403-1_15

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 [17], 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. 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 [17], 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 [1,8,20,22] 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 it is used for the next matrix operation. 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 [7,23] 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 been 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 into algebraic forms and then normalize and rewrite these forms into these algorithms using algebraic rewrite rules. We believe that this approach will be very effective when applied, not only to mathematical operations, but also to a wide spectrum of queries whose functionality is in essence equivalent to these mathematical operations.

The contribution of this work can be summarized as follows:

- We introduce a new 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 (Sect. 5).
- We provide an efficient implementation of GroupByJoin in Map-Reduce based on an algorithm that generalizes the SUMMA parallel algorithm for matrix multiplication (Sect. 6).
- We have extended the query optimization framework in MRQL to generate physical plans that use this 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 number of Map-Reduce operations for queries that contain consecutive matrix operations (Sect. 7).

– We report on a prototype implementation of our framework using MRQL running on top of Hadoop Map-Reduce. We show the effectiveness of our method through experiments on two queries, a simple query that combines matrix multiplication with matrix transpose, and the very complex query for matrix factorization, that is both iterative and contains many matrix operations in every iteration (Sect. 8).

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, which 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 use distributed memory for inter-node communication, such as the main memory Map-Reduce (M3R [21]), Apache Spark [19], Apache Flink [6], and distributed GraphLab [15]. Another alternative framework to the Map-Reduce model is the Bulk Synchronous Parallelism (BSP) programming model [23]. The best known implementations of the BSP model for data analysis on the cloud are Google's Pregel [16] and Apache Hama [10].

Most other array-processing systems use special storage techniques, such as regular tiling, to achieve better performance on certain array computations. SciDB [22] is a large-scale data management system for scientific analysis based on an array data model with implicit ordering. The SciDB storage manager decomposes arrays into a number of equal sized and potentially overlapping chunks, in a way that allows parallel and pipeline processing of array data. Like SciDB, ArrayStore [20] 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 [8] 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 [1] 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. Finally, MLlib, which is part of MLbase [13], is a machine learning library built on top of Spark and includes algorithms for fast matrix manipulation.

3 Background: The MRQL Query Language

Apache MRQL [17] is a query processing and optimization system for large-scale, distributed data analysis. MRQL was originally developed by the author [4,5], 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 [9], in BSP mode (Bulk Synchronous Parallel model) using Apache Hama [10], in Spark mode using Apache Spark [19], and in Flink mode using Apache Flink (previously known as Stratosphere) [6]. 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 [11] and PigLatin [18], 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]
```

where Points is the input data set of points on a plane, centroids is the current set of centroids (k cluster centers), and distance is a function that calculates the distance between two points. The initial value of centroids (the ... value) can be a bag of k random points. The select-query in the group-by part assigns the closest centroid to a point s (where [0] returns the first tuple of an ordered list). The 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
```

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.

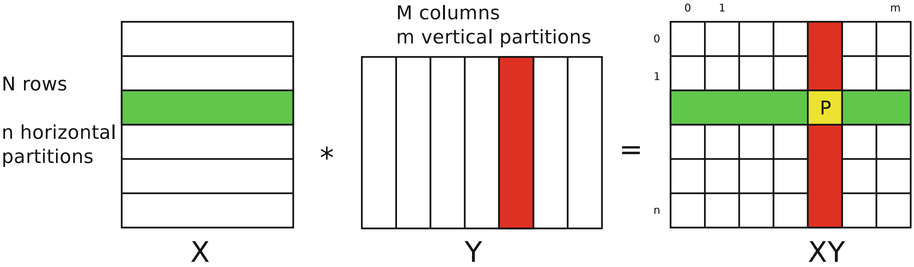


Fig. 1. Matrix multiplication: each partition P requires N/n rows from X and M/m columns from Y

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 cluster nodes for the group-by operation. Instead, one may use the SUMMA algorithm for matrix multiplication [7], which has been adapted for the BSP distributed model [23] and later for Map-Reduce [2]. 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 [12], 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 subtraction. Matrix factorization can be done using an iterative algorithm that repeatedly applies the following rules to minimize the error matrix E:

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

where γ is the learning rate and λ is 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, by generalizing these algorithms, one can optimize a wider spectrum of queries that resemble matrix multiplication, such as calculating the shortest distance between all pairs of nodes in a graph G:

```
repeat S = G
  step select (x,z,min(d))
           from (x,y,d1) in S, (y,z,d2) in S, z = d1+d2
          group by x, z
```

(assuming for simplicity that $(x, x, 0) \in G$ for every node x).

5 The GroupByJoin Operation

In this section, we generalize matrix multiplication using an algebraic operation, called a *Group-By Join*. Let X and Y be bags of types $\{\alpha\}$ and $\{\beta\}$, respectively, for arbitrary types α and β. The 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) )
```

which generalizes matrix multiplication, returns a value of type $\{\delta\}$, where

- jx is the left join key function of type $\alpha \rightarrow \kappa$,
- jy is the right join key function of type $\beta \rightarrow \kappa$,
- gx is the left group-by function of type $\alpha \rightarrow \kappa_1$,
- gy is the right group-by function of type $\beta \rightarrow \kappa_2$,
- h is the head function of type $((\kappa_1, \kappa_2), \gamma) \rightarrow \delta$.
- reduce(acc,zero,s) reduces the elements of a bag s of type $\{(\alpha, \beta)\}$ into a value of type γ, using an accumulator acc of type $((\alpha, \beta), \gamma) \rightarrow \gamma$ and a zero value of type γ. That is, $\mathsf{reduce}(\mathsf{acc}, \mathsf{zero}, \{z_1, z_2, \ldots, z_n\}) = \mathsf{acc}(z_1, \mathsf{acc}(z_2, \ldots, \mathsf{acc}(z_n, \mathsf{zero})))$.

To preserve bag semantics, we must have $\mathsf{acc}(x, \mathsf{acc}(y, s)) = \mathsf{acc}(y, \mathsf{acc}(x, s))$, for all x, y, and s.

The previous generic query is captured by the higher-order physical 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, matrix multiplication, which corresponds to the MRQL query

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

is captured by the operation:

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

6 The Implementation of GroupByJoin in Map-Reduce

The GroupByJoin operation distributes the data to 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.

Figure 2 shows the pseudo-code for the implementation of GroupByJoin in Map-Reduce, where flush(H) is:

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

```
mapLeft ( x ):
  for each i in 0..m−1
      emit ( ((hashCode(gx(x)) % n)∗m+i, jx(x), 1), (1,x) )

mapRight ( y ):
  for each i in 0..n−1
      emit ( ((hashCode(gy(y)) % m)+m∗i, jy(y), 2), (2,y) )

reduce ( ( partition ,joinkey ,tag ), values ):
  if ( partition != current_partition )
     flush (H)
     current_partition ← partition
  // (1,x) tuples arrive before (2,y) tuples in values
  for each leading (1,x) tuple in values
     insert x into xs
  for each (2,y) tuple in the rest of values
     for each x in xs
        key ← (gx(x),gy(y))
        if (H[key] is null)
           H[key] ← zero
        H[key] ← acc( (x,y), H[key] )

cleanup ( ):
  flush (H)
```

Fig. 2. Map-Reduce pseudo-code for GroupByJoin(jx, jy, gx, gy, acc, zero, h, X, Y)

which applies the function h to each key-value pair in the key-value map H and emits the results to the output. Similar to a regular reduce-side join on Map-Reduce [14], our group-by join uses two mappers, mapLeft and mapRight, for each of the inputs, X and Y, respectively. Both mappers emit pairs of key-values. 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 is a triple (partition,joinkey,tag), where partition is one of the $n * m$ partitions, and joinkey is the join key value, jx(x) or jy(y). The partition number of a partition (i, j) in the grid of $n*m$ partitions is equal to $i*m+j$. The two mappers replicate the X values m times and the Y values n times (associated with different partition numbers). A value x $\in$ X is sent to all the row partitions (gx(x) mod n, $*$) and a value y $\in$ Y is sent to all the column partitions ($*$, gy(y) mod m). Hadoop Map-Reduce supports custom partitioning, grouping, and sorting functions that control the shuffling of the map results to the reducers. 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 (sub-minor order).

That is, each partition will contain multiple reduce groups, one for each join key. For each partition p and for each different join key value v, the grouping values in the reducer method, reduce, will contain all the tuples from $\mathsf{x} \in \mathsf{X}$ and $\mathsf{y} \in \mathsf{Y}$ that are shuffled to this partition and satisfy $\mathsf{jx(x)} = \mathsf{jy(y)} = \mathsf{v}$. 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), while the size of hash table H will be $(N * M)/(n * m)$. The number of partitions may be larger than the number of worker nodes (the reducers). That is, each reducer may receive multiple partitions, and each partition may contain multiple groupings. Each grouping is handled separately by the reduce method, and the results of processing each partition is emitted by flush(H) at the end of each partition (when the partition number changes). The result of processing each partition are stored in the hash table H, of maximum size $(N * M)/(n * m)$. That is, we must select n and m to be the minimum values so that H can fit in memory. That is, if there is available memory to fit $\mathcal{T}$ tuples, then $(N * M)/(n * m) = \mathcal{T}$. Our goal is to minimize data replication, which is equal to $N * K * m + K * M * n$. That is, we want to minimize $N/n + M/m$ (if we divide by the constants K and $n * m$). This is possible, when $N/n = M/m = \sqrt{\mathcal{T}}$. Internally though, done implicitly by Hadoop Map-Reduce, each reducer node sorts and groups its entire partition (which contains $N * K/n + K * M/m$ tuples) before reduction, which is done with external sorting at each reducer.

7 Translating Queries to GroupByJoin Operations

Based on the discussion in the Introduction, it would be hard to use source-to-source transformations to put queries, such as matrix multiplication and shortest distance, into an algebraic form, such as GroupByJoin, 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 into their default algebraic forms and then normalize and rewrite these forms using algebraic rules.

The MRQL algebra used in this section has already been described in our previous work [4]. 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 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:

$$\text{cMap}(f, X) \;=\; \{\, z \,|\, x \in X,\, z \in f(x)\,\} \tag{1}$$

Given an arbitrary type κ that supports value equality (=), an arbitrary type α, and a bag X of type $\{(\kappa, \alpha)\}$, the operation 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 $\alpha \rightarrow \kappa$ and j_y of type $\beta \rightarrow \kappa$, and combines the joining values using the function h of type $(\alpha, \beta) \rightarrow \gamma$, deriving a bag of type $\{\gamma\}$. Finally, aggregations are captured by the operation 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 $(\alpha, \beta) \rightarrow \beta$ and a zero value $zero$ of type β. For example, 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:

$$\mathrm{cMap}(f, \mathrm{cMap}(g, S)) \rightarrow \mathrm{cMap}(\lambda x.\, \mathrm{cMap}(f, g(x)), S) \tag{2}$$

that 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 Eq. (1):

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

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

$$\begin{aligned}
&\mathrm{join}(\, j_x,\, j_y,\, h,\, X,\, \mathrm{cMap}(\lambda y.\, \{f(y)\}, Y)\,)\\
&\qquad \rightarrow\ \mathrm{join}(\, j_x,\, \lambda y.\, j_y(f(y)),\, \lambda(x,y).\, h(x, f(y)),\, X,\, Y\,) \qquad (3)\\
&\mathrm{cMap}(\lambda v.\, \{f(v)\},\, \mathrm{join}(\, j_x,\, j_y,\, h,\, X,\, Y\,))\\
&\qquad \rightarrow\ \mathrm{join}(\, j_x,\, j_y,\, \lambda(x,y).\, f(h(x,y)),\, X,\, Y\,)) \qquad (4)
\end{aligned}$$

In our framework, GroupByJoin operations are derived from algebraic forms 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 an equi-join followed by a group-by to a GroupByJoin. For example, the MRQL query 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
```

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 ) ) )
```

while the MRQL query that captures matrix transpose Y^T:

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

is translated into the following algebraic form:

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

Hence, using Eq. 3, the two cMaps in the composition $X \times Y^T$ are fused into:

```
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 ) ) )
```

which 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 )
```

that combines matrix multiplication with matrix transpose.

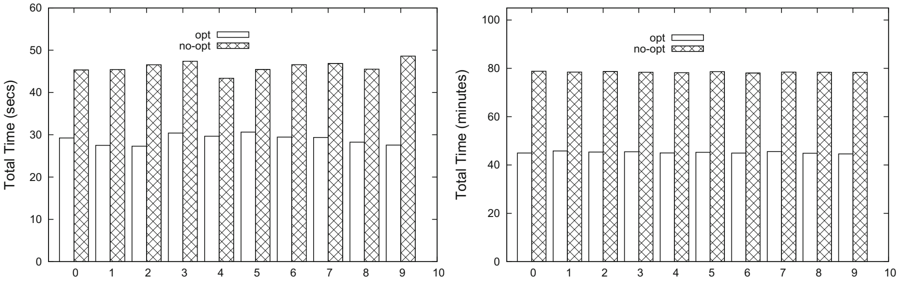
8 Performance Evaluation

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 4 GB RAM and 2 VCPUs at 2.3 GHz. For our experiments, we used Hadoop 2.6.0 (Yarn) and MRQL 0.9.6. The cluster frontend was used exclusively as a NameNode/ResourceManager, while the rest 8 compute nodes were used as DataNodes/NodeManagers. For our experiments, we used all the available 16 VCPUs of the compute nodes for Map-Reduce tasks.

We have experimentally validated the effectiveness of our methods using two MRQL queries: Matrix factorization using gradient descent, shown in Fig. 3, and the simple query: multiply(Pmatrix,transpose(Qmatrix)), where multiply and transpose are also given in Fig. 3. Given a matrix R, our matrix factorization query in Fig. 3 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, Rmatrix, was an $n \times m$

```
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
};
let (E,P,Q) = factorize (Rmatrix,Pmatrix,Qmatrix)
 in multiply (P,transpose(Q));
```

Fig. 3. Matrix factorization using gradient descent in MRQL



(A) A simple query data with/without optimization

(B) Matrix factorization with/without optimization

Fig. 4. Evaluation of a simple matrix query (A) and matrix factorization (B)

sparse matrix with random integer values between 1 and 5 (resembling the 5-star rating in Netflix) in which only the 10 % of the elements were provided (the rest were zero). The size of m was always kept equal to $10 * n$, while $n * m$ was equal 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, Pmatrix and Qmatrix, had sizes $n * k$ and $m * k$, respectively, where $k = 10$ for all experiments (a low rank), and were initialized with random values between 1 and 5.

For both MRQL queries, we perform our evaluations in two modes: with and without inter-operation optimization. With inter-operation optimization means that matrix operations were defined using macros so that compositions of operations are fused into one operation, thus avoiding the creation of intermediate results (which Hadoop Map-Reduce must store in the HDFS). Without inter-operation optimization means that the matrix operations were defined as opaque functions, which have to be evaluated as is, thus offering no opportunities for optimization. The results for the simple query multiply(Pmatrix,transpose(Qmatrix)) are shown in Fig. 4A. The results look very similar for different data sizes (100K through 145K tuples) because all matrices (including the intermediate results) are split into 16 files (one for each compute node in the HDFS) and each file can fit into one HDFS block (64MBs) regardless of its size. We can see in Fig. 4A that there is improvement even for just two operations: matrix multiplication and transpose. With inter-operation optimization, these two operations are fused into a single one, a GroupByJoin, which runs in about the same time as matrix multiplication alone. The results for matrix factorization are shown in Fig. 4B. Here, the improvement is even more substantial (the optimized query takes about half the time of the non-optimized one) since the results of all these optimizations are aggregated and repeated at each iteration step.

9 Conclusion

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.

Acknowledgments. This work is supported in part by the National Science Foundation under the grant CCF-1117369. Our performance evaluations were performed at the Chameleon cloud computing infrastructure, www.chameleoncloud.org, supported by NSF.

References

1. Buck, J., Watkins, N., Lefevre, J., Ioannidou, K., Maltzahn, C., Polyzotis, N., Brandt, S.A.: SciHadoop: array-based query processing in hadoop. In: SC 2011
2. Das, A., Afrati, F.N., Salihoglu, S., Ullman, J.D.: Upper and lower bounds on the cost of a map-reduce computation. In: VLDB 2013
3. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: OSDI 2004
4. Fegaras, L., Li, C., Gupta, U.: An optimization framework for map-reduce queries. In: EDBT 2012
5. Fegaras, L., Li, C., Gupta, U., Philip, J.J.: XML query optimization in map-reduce. In: International Workshop on the Web and Databases (WebDB) (2011)
6. Apache Flink. http://flink.apache.org/
7. Geijn, R.A., Watts, J.: SUMMA: scalable universal matrix multiplication algorithm. Concurrency: Pract. Experience **9**(4), 255–274 (1997)
8. Ghoting, A., Krishnamurthy, R., Pednault, E., Reinwald, B., Sindhwani, V., Tatikonda, S., Tian, Y., Vaithyanathan, S.: SystemML: declarative machine learning on MapReduce. In: IEEE International Conference on Data Engineering (ICDE) (2011)
9. Apache Hadoop. http://hadoop.apache.org/
10. Apache Hama. http://hama.apache.org/
11. Apache Hive. http://hive.apache.org/
12. Koren, Y., Bell, R., Volinsky, C.: Matrix factorization techniques for recommender systems. In: IEEE Computer, August 2009
13. Kraska, T., Talwalkar, A., Duchi, J., Griffith, R., Franklin, M., Jordan, M.I.: MLbase: a distributed machine learning system. In: Conference on Innovative Data Systems Research (2013)
14. Lin, J., Dyer, C.: Data-intensive text processing with MapReduce. Book pre-production manuscript, April 2010
15. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Distributed GraphLab: a framework for machine learning and data mining in the cloud. In: VLDB 2012
16. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: PODC 2009
17. Apache MRQL (incubating). http://mrql.incubator.apache.org/
18. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: a not-so-foreign language for data processing. In: SIGMOD 2008
19. Apache Spark. http://spark.apache.org/
20. Soroush, E., Balazinska, M., Wang, D.: ArrayStore: a storage manager for complex parallel array processing. In: SIGMOD 2011
21. Shinnar, A., Cunningham, D., Herta, B., Saraswat, V.: M3R: increased performance for in-memory Hadoop jobs. In: VLDB 2012
22. The SciDB Development Team: overview of SciDB: large scale array storage, processing and analysis. In: SIGMOD 2010
23. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)