Spangle: A Distributed In-Memory Processing System for Large-Scale Arrays

Sangchul Kim Bogyeong Kim Bongki Moon
Department of Computer Science and Engineering
Seoul National University, Seoul, Korea
stdio@snu.ac.kr, bgkim@dbs.snu.ac.kr, bkmoon@snu.ac.kr

***Abstract*—With increasing volumes of scientific data, a scalable and parallel computing framework is required for scientific analysis in computer simulations and experiments. Scientific data are commonly generated in multi-dimensional arrays, and the array data model is appropriate to store them for analysis, including for data mining and arithmetic computation. In this paper, we introduce an array processing system called *Spangle*. It is implemented on top of Apache Spark, a popular map-reduce framework for complex computation workloads. To support array data computation, we extended Resilient Distributed Dataset (RDD) based on the array data model named *ArrayRDD*. ArrayRDD is an inherently parallel data structure that provides fault-tolerance. In addition, by adopting the array data model, Spangle provides an interface for expressing machine learning algorithms, which heavily rely on linear algebra. We tailored two popular algorithms, PageRank and Stochastic Gradient Descent, for large-scale datasets in Spangle.**

I. Introduction

With the advancement of data collection and storage technologies, a broad range of fields, such as climatology, geology, and astronomy, produce large-scale arrays. For instance, the Large Synoptic Survey Telescope (LSST), built to survey the night-time sky, collects 60PB of images over ten years. In these fields, the complex analysis of scientific raster data plays an important role in understanding scientific phenomena. As the volume of such data increases, the analysis becomes heavy work that entails a long running time, which requires a scalable and parallel framework such as Apache Spark for the desirable performance and productivity.

Moreover, the growing demand for large-scale machine learning has been changing the landscape of data analysis. Many machine learning algorithms heavily rely on algebraic computations (*e.g.*, matrix calculations for logistic regression). There is a strong need for algorithms dealing with large-scale matrices that do not fit into a single machine memory. A large machine learning task may have to be distributed and processed in parallel to reduce the training times. It is crucial to provide a scalable and efficient programming model that domain scientists can easily adopt for their data processing needs.

We introduce Spangle, an array processing system implemented on top of Apache Spark. Spangle extends Resilient Distributed Dataset (RDD) for large-scale array processing. The parallel data structure, called ArrayRDD, is based on the array data model and inherently provides fault tolerance. ArrayRDD can represent multi-dimensional matrices, arrays as well as spatio-temporal data.

Spangle provides declarative interfaces to manipulate arrays. It allows us to organize tasks into a pipeline. Spangle manages sparse arrays containing null values or invalid cells with bitmasks. The bitmask is a series of bit vectors and provides non-trivial benefits. Without having to store invalid cells explicitly, it can compress an array thereby enabling to load larger ones into memory. Besides, it can reduce the computational volume by bypassing the invalid cells.

We can leverage not only the array indexing capability for performance but also a range of array processing APIs for programmability. Spangle can support processing large-scale raster data or large training data for machine learning algorithms. To evaluate the effectiveness of Spangle, we have customized and optimized the two popular machine learning algorithms, PageRank and stochastic gradient descent (SGD), for Spangle. Spangle achieves better performance than existing array processing systems. Overall, we make the following contributions.

- We have implemented an array processing system called Spangle for large-scale raster data analytics and machine learning.
- For sparse arrays, Spangle employs bitmasks in a few different modes so that it can compress large arrays.
- We have customized two machine learning algorithms, PageRank and SGD, for Spangle. It is demonstrated from these tailored algorithms that Spangle can facilitate complex analytics effectively.

This paper is organized as follows. We describe the background information in Section II. In Section III, we present our novel system, Spangle, and its architecture. Then, we describe the bitmask which is a Spangle component, and how it manages chunks and processes arrays in Section IV. Section V shows programming interfaces in Spangle. In Section VI, we optimize two machine learning algorithms for Spangle, and Section VII shows the performance of Spangle. Last, Section VIII reviews related work, and Section IX concludes this paper.

This work was partly supported by the National Research Foundation of Korea (2020R1A2C1010358 and 2016M3C4A7952633). The authors assume all responsibility for the content of the paper.

2375-026X/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00158

II. Background

In this section, we present the background of the array model and discuss null values in raster data. We then provide an overview of Apache Spark.

A. Array Data Model

An array, a collection of homogeneous elements and totally ordered, is a fundamental data model. It is composed of consecutive elements, named *cell*s. A cell can have multiple values, mapped into the same array index. For instance, a sensor can produce multiple values in a particular area (*e.g.*, temperature, precipitation, and pressure in climate data). This model is widely used to process multi-dimensional data, such as geospatial data, and to implement scientific and machine learning algorithms.

The raster data, generated by scientific observation and computer simulation, highly employ the array data model. These data have dimensions discretized regularly over time and space. Remote sensing, for example, processes spatial and temporal information and generates images. Values in the images are enumerated along with coordinates (*e.g.*, longitude and latitude), naturally represented in multi-dimensional arrays.

Another application of this model is to express linear algebra. In fact, many scientific algorithms are represented as matrices. For instance, two- or three-dimensional Fast Fourier Transforms are widely used in the field of image processing. Machine learning algorithms, such as matrix factorization and principal component analysis, are also key concerns with linear transformations. These are tightly related to vector spaces or specific matrix operations.

B. Null Value in Raster Data

The real array datasets contain *null values*, also called no-data or missing data, when data are lost or dropped. It represents the invalid state of data. This case occurs in, for example, satellite sensors observing objects which are often missed or unknown [1]. The objects in the universe such as stars or galaxies are sparsely distributed. That is, most astronomy image data are filled with a few empty regions [2]. Likewise, in monitoring shipping vessels, data around the coastline are empty because they congregate near major ports [3].

Each array-based system [4]–[7] adopts different methods to describe null values. The values can be encoded as NaNs, supported in most languages such as C++ and Java. The NaN is a straightforward representation of a null value. A mathematical operation (*e.g.*, addition) with a null value is not defined, which is the same as NaN. That is, the result of arithmetic operation is null (*e.g.*, $1 + null = null$), which is the same as that of NaN (*e.g.*, $1 + NaN = NaN$). It is unnecessary to define a character for a null value. However, it can have a limitation because the value is not specified for all general types, such as Int and Long. Another method to represent a null value is to use specific values such as the minimum or maximum value of a primitive type (*e.g.*, int_max or float_min). However, this could be cumbersome because a cell can have any real value. If a value is int_max, not a null value, a system cannot determine whether the value is a real value or null value.

Alternatively, the auxiliary data structure, bitmask, can be adopted to represent *null values*. Each bit vector in a bitmask can be either one or zero. If the data are a real value, a bit vector is set to one. Otherwise, the bit vector is set to zero. While the bitmask requires additional space, it is independent of any data type. A system can use all values and express the data status with the minimum size (*i.e.*, one bit per cell), compared with the above methods.

While a system processes raster data, values can be translated as null values. Suppose that scientists only focus on chlorophyll [8], where values are greater than a specific threshold below the sea surface. These negligible cells are considered to be invalid. Spangle treats a specific cell that is not of interest as a null value.

C. Overview of Apache Spark

Apache Spark [9] is a general-purpose framework for large-scale data processing with APIs in Scala, Java, and Python. More recently, a number of high-level APIs have been developed based on Spark. With built-in modules for streaming data analysis [10], SQL [11], machine learning [12], and graph processing [13], Spark enables advanced in-memory big data processing and analytics in many fields such as finance and e-commerce industries.

For in-memory cluster computing, Spark introduced an efficient abstraction called Resilient Distributed Datasets (RDDs) [14]. Each of them is a distributed collection of objects partitioned over a cluster. To manipulate RDDs, Spark provides a functional programming interface with two types of RDD operations, namely, transformation (*e.g.*, `map` and `filter`) and action (*e.g.*, `reduce` and `collect`).

Spark processes data through the combination of stages which are a series of RDD operations. Each stage is lazily evaluated. In other words, transformations are not executed until the actions are triggered. It enables Spark to avoid executing unnecessary transformations. RDDs are fault-tolerant with a lineage graph which is information about how they were derived from other RDDs. Using this graph, Spark can reconstruct RDDs after a failure. In addition, they explicitly persist in memory or on disk to accelerate data access and reuse.

III. Architecture of Spangle

In this section, we present a novel system called Spangle. It is designed to process and analyze large-scale arrays such as raster data and images, as well as to compute large matrices. We introduce its architecture and then describe each component.

A. System Overview

Spangle is an in-memory distributed processing system that adopts an array data model, built on top of Apache Spark. It supports array operations to enable scientific or business analytics over multi-dimensional datasets. The architecture of

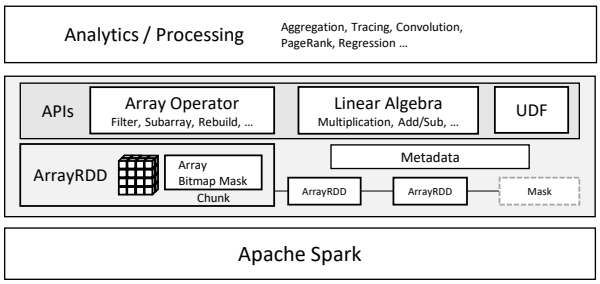


Fig. 1: The architecture of Spangle

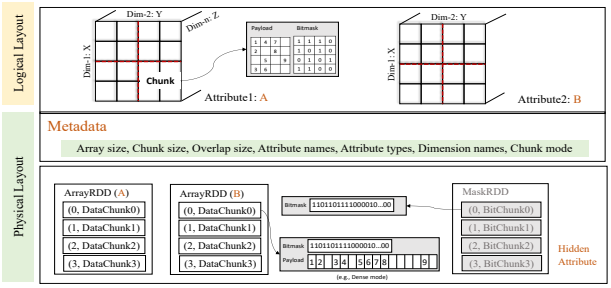


Fig. 2: The ArrayRDD and chunk layout

Spangle is described in Figure 1. It is composed of two main components: ArrayRDD and its metadata. The ArrayRDD, a dataset of Spangle, is an extension of the RDD and follows its properties, such as fault-tolerance and lazy evaluation. The other component is metadata. It is a description of an array and presents array specifications, such as the starting point and data types of attributes. Using this information, Spangle can manage arrays as physical and logical layouts, respectively.

Spangle is able to process arrays that can have two features: an irregular form and multiple attributes. The ragged data (irregular form) are usually sparse, even skewed, and the majority of them are empty cells (no-data) in scientific data, described in Section II-B. To represent null values, we employ bitmask, coupled with a payload in a chunk. A chunk clusters adjacent cells and preserves the data locality, which can easily access adjacent cells together. Without storing null values, Spangle can reduce the size of arrays, described in detail in Section IV-A.

Moreover, Spangle can manage multi-attribute arrays. While scientific data may have a number of attributes, most applications span the processing time within a few of them [15]. Considering this characteristic, we adopt a column-store manner to efficiently manage attributes by mapping an attribute into an ArrayRDD. The benefits of a column-store are to reduce the data size by compression and improve the cache hit ratio [16], suitable for in-memory processing systems. Even for machine learning, a column-store manner achieves high performance [17].

In Spangle, each cell can be identified by its array index. Spangle manages arrays by ArrayRDD, which consists of *chunk*s, and uses a special ArrayRDD, named *MaskRDD*. In particular, ArrayRDD inherits *PairRDD* (key-value RDD), where each record is composed of a key-value pair. To create ArrayRDD, Spangle first ingests data (*e.g.*, CSV and NetCDF) and assigns a unique ChunkID to each cell and groups cells with the same key. Afterward, cells are mapped into the payload, and the bitmask is set. These are performed as pipelining. When an operation, such as matrix multiplication, incurring data shuffling, or join is executed, Spangle re-distributes each of them and processes arrays in parallel based on chunks. The details are described below.

1) Chunk: An ArrayRDD is composed of non-overlapping blocks of an array, *chunk*s. An array is partitioned into chunks and distributed across multiple workers (*i.e.*, executors in Spark). A chunk contains geographically contiguous data, where co-located cells are clustered to ensure data locality. This is essential in the coordinate system because adjacent data are usually accessed and processed together. In a distributed environment, it reduces the network communication cost and brings out cache effects. However, several operators require data shuffling, which incurs significant overhead, such as in image processing. To minimize this overhead, a chunk can have extra cells by a given number in boundaries along each dimension, named *overlap* [18] which Spangle employs. It is useful when workers require adjacent cells in a chunk boundary (*e.g.*, blurring images). This technique can avoid data exchanges, which reduces the network overhead.

B. ArrayRDD

A chunk consists of two components, a `payload` and a `bitmask`, as shown in Figure 2. The payload is a collection of actual values, and the bitmask indicates their validity. The payload is physically stored in a one-dimensional array. In a distributed environment, each chunk must have a unique identifier to access cells. Specifically, ArrayRDD manages each record as a pair of a chunk and its ID. The ID is a single value, which stands for multiple values (*e.g.*, coordinates). Compared with multi-value representation, the single-value representation supports any arrays without concern for the number of dimensions and reduces the key length and lookup cost. Spangle assigns a chunk ID to every chunk, unless all cells in a chunk are empty. Consequently, Spangle does not create empty chunks, which also reduces the data size in memory.

1) MaskRDD: The *MaskRDD* is a hidden attribute that is internally used. It stores the global positions of null values and thus provides a global view for visible attributes. It is essential to achieve better performance when the number of attributes is greater than one. For instance, a cell filtered out in one attribute by the *Filter* operator, described in Section V, must be excluded from the other attributes. Spangle maintains this consistency in every operation; however, it is quite expensive. MaskRDD can reduce this cost, similar to the Spark strategy, *lazy-evaluation*. That is, every operation transforms only a MaskRDD, and Spangle evaluates all ArrayRDDs on-demand

from MaskRDD. This benefit in terms of the performance is described in Section VII-B.

C. Metadata and Mapper

To manage arrays, Spangle stores metadata, such as the starting and ending points of arrays, the interval of chunks, and data types. With this metadata, the *mapper* virtually translates a logical layout to the physical layout, and vice versa, as shown in Figure 2. It derives coordinates from a chunk ID, and Algorithm 1 shows how it works. This algorithm is used to create chunks and to retrieve cells within a specific range. We use this idea to optimize the performance by applying it to a mathematical operation in Section VI-C.

Algorithm 1: Computing a Chunk ID from Coordinates

```
input : metadata mt, coordinates pos
output: Chunk ID
chunkID = 0
length = 1
for i=0...mt.getNumDim-1 do
    chunkID += (pos(i) / mt.getChunkSize(i)) * length
    length = length * math.ceil(mt.getArraySize(i) /
      mt.getChunkSize(i))
end
chunkID
```

IV. Bitmask

This section introduces three different chunk management modes and describes how to compress arrays and access them for each mode in detail. Given the data distribution, a chunk is managed in three distinct modes: *Dense*, *Sparse*, and *Super-Sparse*. Under these modes, cells can be accessed in different approaches.

Assuming that the time complexity of the one-bit count for a word is constant, the computation for random access in n-words takes O(n). If an operation scans all elements in a chunk, then it takes O(n^2) time. To reduce the long-running time, we optimize it by distinguishing two access patterns, described in Section IV-B.

A. Chunk Management

As the density of an array, Spangle manages chunks in three different modes: dense, sparse, and super-sparse, described in Figure 3. It is not necessary to compress a dense array, whereas a sparse or super-sparse array needs to be compressed, which physically removes invalid cells and reduces the size of an array. In matrix operations, zero is treated as invalid. Similar to managing raster data, compression methods can be applied to a sparse or super-sparse matrix. Especially, in matrix operations such as matrix multiplication, the data size impacts on the network overhead which is the major factor of the performance. In addition, the bitmask can represent the static graph, described in Section VI-B.

Dense. The dense mode is a straightforward method. A payload is filled with almost valid cells, and bit vectors in a bitmask are almost set. That is, the size of every chunk is equal. Spangle can directly access a cell using an array index.

Fig. 3: Three modes for data distribution

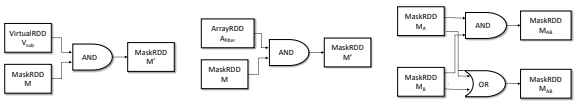
Sparse. Spangle drops invalid cells in the sparse mode. While it can reduce the payload size, all cells lose their initial positions (*i.e.*, indices), which cannot directly access cells with array indexes. To solve this challenge, we use a bitmask to identify valid cells and inform the original position. For the point query, for example, Spangle first searches for a corresponding chunk and scans a bitmask. By counting ones in a bitmask from the beginning to a given position, Spangle can derive the initial position.

For a sparse matrix, bitwise operations such as `and` and `or` are used between two matrices to reduce the computation overhead. For example, the element-wise product needs a bitwise `and` (&) operation. If a bit is unset (0), the corresponding cell must be zero (null); therefore, the computation between two cells is not required. By the `and` operation over two bitmasks, Spangle does not multiply the element-pair when at least one element in the pair is zero.

Super-Sparse. The super-sparse mode is that few valid cells are in a chunk, and the bitmask size accounts for the majority size of a chunk, where most bits in the bitmask are zero. This case often arises when valid cells which are evenly distributed (*e.g.*, a normal distribution) rarely exist. Due to our chunk management policy, if a chunk has more than one valid cell, Spangle creates the chunk. If then, the bitmask size can be larger than the payload size. Thus, to reduce the bitmask size, we set two levels, called the *hierarchical bitmask* method. If a bit at the upper level is zero, the corresponding word at the lower level must be filled with all zeros, which leads to removing the word.

B. Bitmask Operations

When arrays are managed in the sparse mode, the computation to obtain the number of one-bits, called *population count*, is required. In the literature, the algorithms [19], [20] introduce the population count in a word, whereas a built-in function achieves the best performance in practice. In Java, the function (`java.lang.Long.bitCount()`) is a native function, which can be treated by the JVM as intrinsic and be interpreted with a single machine code instruction. However,



(a) Subarray (b) Filter (c) Join

Fig. 4: Generate a MaskRDD for each operator

the population count over several words takes a longer time. For instance, assuming that a word is 64-bits in sparse mode, accessing the 67th cell must compute the population count of the first and second words. To reduce the processing time, we propose a hybrid approach that combines sequential and random access.

1) Sequential Access: The operators (*e.g.*, Filter and Aggregator, described in Section V), which read all cells, have a sequential access pattern. Considering the pattern where cells are accessed from the beginning (*e.g.*, scanning all cells), Spangle simply counts set bits from the previous position to the current position, called the *delta count*. That is, the number of bits for the following position is the count of the current bits plus the delta count until the next position, which avoids redundant computation.

2) Random Access: Few, but most frequently used operators have a random access pattern, such as subarray. In the sparse mode, Spangle counts the set bit to access a cell. To leverage the performance of the population count, we use SIMD operations, which can accelerate the counting in parallel with vectorization. Assuming that a computer supports AVX2 instructions, 256 bits (four words) can be computed in parallel. To use AVX2 in the JVM, we employ JNI to call native libraries because the AVX2 instruction library is called in the C language.

In addition, we employ a population count algorithm [21] to count the number of ones over 64 words in constant time. It is important to determine the chunk size, which is related to the algorithm and parallel processing. In our experiments, the appropriate chunk size is approximately 4,096 (64 words) - 65,536 (1,024 words). In the case of the chunk size larger than 64 words, we locate milestones that store population counts for every 64 words. As the SIMD operations are fast, however, the overhead of JNI is incurred. Due to the overhead, we do not use it for the sequential access pattern.

V. Programming Interface

In order to process multi-dimensional arrays, Spangle offers declarative interfaces to support high-level programming, which facilitates the implementation of both iterative algorithms and interactive analysis. The array operators such as filter and subarray are based on array algebra [22]–[24] and map algebra [25], which are adopted in array-based or geospatial systems [4], [5], [26]. Based on these algebras, Spangle provides the following core operators: Subarray, Filter, Join, and Aggregator. In addition to these operators, we add the matrix operators to support linear algebra on which machine learning strongly relies. Of the operators, Aggregator

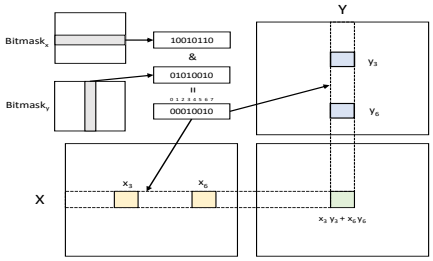


Fig. 5: Matrix multiplication using bitmasks

requires aggregation functions, such as sum, min, and average, and Spangle provides an abstraction to create user-defined functions. Internally, bitwise operations are performed while the operators process data. If the MaskRDD is used, bitwise operations are more critical. In this section, we describe the process of operators with bitwise operations.

A. Operators using Bitmasks

1) Subarray: Figure 4a shows the process of the MaskRDD for Subarray. It retrieves cells in the rectangular sub-region of an input array with the given top-left and bottom-right coordinates. If there is a MaskRDD, chunks are selected in the MaskRDD. Within the given range, bits in the virtual bitmask of each chunk are set. Then, the bitwise-and operation is executed between the virtual bitmask and the bitmask of each chunk.

2) Filter: It filters out all cells that are not satisfied with a given condition in a function. This function, specified by users, returns `true` or `false` by examining every cell. If the function evaluates a cell as false, Spangle treats the cell as invalid. Spangle sets bit-vectors as false for a cell that shifts to invalid. Then, it computes bitwise-and operation between both bitmasks in a MaskRDD and an ArrayRDD (*i.e.*, a selected attribute), similar to Subarray, as shown in Figure 4b.

3) Join: It joins two arrays based on dimensions. In Spangle, the operator takes two arrays as inputs and returns a new array. The number of attributes in the new array is equal to the total number of attributes between the two input arrays. The join operator has two sub-operators: and-join and or-join. The and-join operator collects valid cells of the same position in both two arrays, while or-join allows cells to be valid if either of the cells is valid in the same position. In Figure 4c, if two input arrays use the MaskRDD, it combines all ArrayRDDs and executes *AND* or *OR* between two MaskRDDs (*i.e.*, M_A and M_B).

4) Matrix Operators: Matrix operators are used for linear algebra, and one of the fundamental operations is matrix multiplication. They use customized math operators, described in Section VI, and library operators (*i.e.*, Breeze). Figure 5 shows matrix multiplication using bitmasks. The bitwise-and operation is performed between two bitmasks. Spangle extracts candidate elements from two matrices and multiplies them, but the multiplication is avoided if one of them is zero (null). This is more effective when one of the two matrices is sparse.

The cost of matrix multiplication is relatively high because of the network overhead in the map-reduce environment. Assuming that two matrices are partitioned using the same method (*e.g.*, hash or range partitioning), embarrassingly parallel matrix operations, such as addition and subtraction, can be computed without data shuffling. Matrix multiplication, however, shuffles chunks across partitions, and it joins two matrices (*i.e.*, ArrayRDDs) using ChunkIDs. Spangle follows a distributed matrix computation process, similar to scatter and gather. It joins two ArrayRDDs by the row ChunkID of a left ArrayRDD and the column ChunkID of a right ArrayRDD. After the matrix computation is completed, Spangle collects corresponding chunks with the add operation to map the resulting chunk.

Spangle minimizes the overhead by an alternative data structure, either bitmask or offset array, only for matrix computation. The offset array is similar to the coordinate list format (*i.e.*, COO) but represents multidimensional coordinates as one-dimensional coordinates. The conversion from a bitmask to an offset array occurs only when the size of the bitmask is larger than the size of the offset array. This conversion is only applied to a static matrix that is hardly updated, for example, the matrix of training data.

B. Aggregate Framework

Spangle provides *Aggregator*, a small framework for executing aggregate functions (*e.g.*, sum, avg, min). It receives dimension names, such as x-axis and y-axis names, as parameters and summarizes an array into a value or values with given conditions. That is, while aggregating an array, Spangle generates the new schema determined by the given conditions.

The aggregate functions consist of four abstraction functions that users can specify: 1) creating specific states for each chunk and setting them to have a default value (Initialize); 2) gathering values in chunks into states along the given dimensions (Accumulate); 3) collecting all states of chunks and generates new states of the new schema (Merge); and 4) evaluating the new states and returns the result (Evaluate).

Spangle also provides *Accumulator* that uses the abstraction functions of Aggregator. Similar to Aggregator, it accumulates each value along with an axis direction or user-defined directions. Users can run this operator in parallel in either a synchronous or asynchronous manner. If there are cells involved in separate chunks in a direction, the value of a previous cell must be computed with the next cell. This would be slow since all chunks require synchronization in the chunk boundary at every step, and the steps proceed one by one. In contrast, in an asynchronous manner, every chunk computes its values internally and then synchronizes. It is only allowed when the application is insensitive to accuracy.

VI. Machine Learning

The array model can easily express a matrix without any data conversion. A two-dimensional array directly shifts to a matrix with the same dimensionality (*i.e.*, row and column). Spangle can seamlessly support statistical and machine learning algorithms, as most of them strongly rely on linear algebra, elegantly expressed by matrices.

In this section, we describe the optimization and customized machine learning algorithms in Spangle. To support linear algebra effectively, we focus on optimizing the process of matrix operations, especially for matrix multiplication. In addition, we tailor two popular machine learning algorithms, PageRank and stochastic gradient descent, for Spangle.

A. Local Join for Matrix Multiplication

Internally, the matrix multiplication includes `Join` and `Reduce` operations provided by Spark. The matrix multiplication consists of three stages: two Join stages and one Reduce stage. While joining two RDDs, Spark writes them to disk prior to data shuffling. If chunks with the same IDs of two matrices reside in the same partitions, Spangle can locally join them, avoiding the shuffling cost. Still, in our observation, sending chunks that have equal IDs to the same partition splits stages and incurs disk I/O for the shuffle read and write.

Spangle provides an RDD wrapper that combines three stages into one stage when two matrices use the same partitioner, and two chunks that have equal IDs are in the same partition. That is, if left and right matrices are partitioned by row IDs and column IDs, respectively, Spangle does not shuffle them.

B. Graph Representation and PageRank

A matrix can represent a graph, $G = (V, E)$, as an adjacency matrix. The elements of the matrix indicate whether pairs of vertices are adjacent. This matrix is efficient when a graph is dense (*i.e.*, $|E|$ is close to $|V|^2$). In Spangle, a chunk consists of a payload and a bitmask, and we make use of the bitmask to represent the adjacency matrix. Because a bit-vector only becomes zero or one, an adjacent matrix using the bitmask can represent an unweighted graph, where the existence of an edge is stored as one bit rather than eight bits (integer value). The weighted graph, however, is not directly represented as a bitmask. We describe how to represent a PageRank graph using a bitmask in Spangle.

PageRank can be expressed as a direct and weighted graph. The weight of each vertex is divided by the number of out-edges and propagated to the other connected vertices. There are variants of PageRank; however, we use a basic algorithm in this paper to clearly describe our architecture. We assume that rows are destination vertices, and columns are source vertices. Because the propagated values in each column are zero or specific equal values, we take the specific values out of the matrix and create a vector (*i.e.*, a one-dimensional array).

In specific, we use a power method, the equation of which is

$$p_i = \alpha \sum_j (a_{ij} \cdot p_j) + (1 - \alpha)/n$$

where a_{ij} is the (i, j)-entry of the transition matrix, p_j is the j-th element of the vector, and α is a damping factor. The transition matrix is derived from the connectivity matrix. Let

N be the total number of pages. The connectivity N×N matrix A is created by defining the (i, j)-entry as

$$c_{ij} = \begin{cases} 1, & \text{if there is a line from j to i.} \\ 0, & \text{otherwise.} \end{cases}$$

To create the transition matrix from the connectivity matrix, the j column is divided by the number of out-edges (w_j), defined as

$$a_{ij} = c_{ij} \cdot w_j \tag{1}$$

where **w** is an N×1 vector. From PageRank and Equation (1), the following equation is derived.

$$p_i = \alpha \sum_j (a_{ij} \cdot p_j) + (1-\alpha)/n = \alpha \sum_j (c_{ij} \cdot w_j \cdot p_j) + (1-\alpha)/n$$

Thus, the original PageRank algorithm at the k-th iteration,

$$\mathbf{p_k} = \alpha A \cdot \mathbf{p_{k-1}} + (1-\alpha)/n$$

is equivalent to

$$\mathbf{p_k} = \alpha A' \cdot (\mathbf{w} \circ \mathbf{p_{k-1}}) + (1-\alpha)/n$$

where the symbol, ∘, is the element-wise product (Hadamard product). The transition matrix A can be decomposed into a matrix $\mathbf{A}'$ and vector **w**. In this decomposition, the bitmask can represent the matrix $\mathbf{A}'$, reducing the matrix size. We implement the customized PageRank in Spangle using this equation and evaluate its performance in Section VII.

C. Stochastic Gradient Descent

The gradient descent algorithm is one of the most well-known and straightforward methods for solving convex optimization problems [27]. It can be reformulated as a quadratic minimization problem (*e.g.*, least squares error) to solve a system of linear equations. It is an iterative algorithm that finds an optimal solution in convex and differentiable functions.

Formally, given the minimizing a function f associated with the i-th observation in the data set, the equation is as follows:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \theta \frac{1}{n} \sum_i \nabla f_i(\mathbf{x}_t)$$

where **x** is updated for each step, t ≥ 0. $\mathbf{x}_0$ starts at an arbitrary point, and it iteratively moves in the direction $\Delta\mathbf{x}_t$ with step size θ. However, computing the gradient every time for all training data is costly. Alternatively, the stochastic gradient descent (SGD) is widely used to reduce the cost. It randomly selects a single training sample instead of the whole. The SGD algorithm is expressed as

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \theta \nabla f_i(\mathbf{x}_t)$$

where i is randomly selected. For distributed computational architectures, especially in the map-reduce framework, stochastic parallel algorithms have been actively studied [28], [29], and we adopt the idea of the parallel SGD algorithm [30]. Similarly, the mini-batch gradient descent is also a common method that selects a few training samples, known for its fast convergence rate with statistical stability. In Spangle, users can assign a parameter, α, to configure how many training samples are used for each step. We present how to customize the SGD algorithm below.

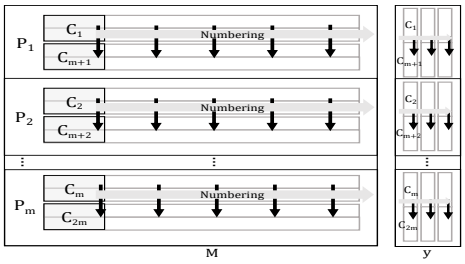


Fig. 6: Numbering chunk IDs

Mapping chunk IDs in parallel: In raster data, Spangle first assigns chunk IDs to all cells, described in Section III-C, and then collects them to create chunks (map and reduce) in each partition. In the SGD algorithm, we can minimize the overhead incurred by data shuffling. This is possible because it is independent between the training samples. Figure 6 shows how to assign them to training samples in parallel.

The M is a matrix based on training data, and y can be a feature or labeled vector. A single row can be split by a specific interval (*i.e.*, the chunk size) along with columns. Given the number of partitions (*nP*), a partition ID (*pID*), and a row chunk ID (*rID*), the equation for chunk ID (C_n) is described as follows:

$$C_n = nP \cdot rID + pID \tag{2}$$

The rID is initially zero and increases up to the number of chunks, derived from the number of rows divided by the chunk interval in a partition. After this equation is evaluated at the first chunk in each partition, such as C_1 and C_m, generating chunk IDs can be continued. The equation does not guarantee consecutive chunk IDs when pID does not monotonically increase. Nevertheless, the equation is valid because Spangle only requires a unique ID for a chunk.

To compute SGD in parallel, Spangle distributes chunks (*e.g.*, hash partitioning) over partitions. However, this causes the network overhead because a few `reduce` step is required to evaluate $\mathbf{x}_{t+1}$. To minimize this overhead, every partition randomly selects a couple of samples at each step using Equation (2) reversely. It searches chunks in parallel, which ensures the linearly scalable performance, according to the parallel SGD algorithm [30]. From the above equation, each partition can retrieve chunks by evaluating the *rID*. This method is independent of partitioners, such as the hash and range.

Customized SGD algorithm: For simplicity, we consider a simple example that uses the gradient descent algorithm, *logistic regression*. The logistic regression is a statistical model widely used for analyzing data, where one or more independent variables determine a dependent variable classified as either zero or one. Suppose that the loss function is provided, the linear equation is

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \theta M_t^T (h(M_t \cdot \mathbf{x}_t) - \mathbf{y}_t)$$

where h(x) is a hypothesis such as a sigmoid, M_t is a partial matrix that consists of randomly selected samples, and x and

Query	Description
Q1 Aggregation	For all images in local coordinate space, compute the average value of selected cells in a specific range. This query might simulate, for example, finding the average, background noise in the raw imagery.
Q2 Regridding	In a specific range, regrid the raw data for the images. Compute the average value of adjacent cells. Gridding of the raw data values may be used for an interpolation function.
Q3 Aggregation	For the observation, select cells in a specific range. Then, compute the average values that match a given condition.
Q4 Polygons	For the observations, select cells in a specific range, and filter the cells that match a given condition. Compute the observations whose values satisfy a given condition.
Q5 Density	For the observations, select cells in a specific range and group the observations spatially into a specific range. Find cells containing more than given observations.

TABLE I: Queries for raster data processing

y are vectors. In this equation, transposing a matrix is quite expensive, consuming O(n/p), where n is the number of cells, and p is the number of executors. Before adapting the SGD algorithm to Spangle, we optimize the above equation to

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \theta((h(M_t \cdot \mathbf{x}_t) - \mathbf{y_t})^T M_t)^T \quad (3)$$

Because the vector size is significantly smaller than the matrix size in general, and the vector is one-dimension (*i.e.*, 1×n), Spangle does not need to transpose the physical layout of the vector. Instead, it only replaces metadata (*e.g.*, from 1×n to n×1).

VII. Experiments

We evaluate the performance of Spangle using real datasets. Spangle was compared with three systems, which can process raster data: SciSpark [31], RasterFrames[1], and SciDB [6]. Afterward, we compared five systems to evaluate the performance of matrix operations: Spangle, SciDB, Spark (COO), MLlib (CSC), and SciSpark. Then, we evaluated Spangle over two machine learning algorithms, PageRank and logistic regression, using Spark built-in modules.

A. Experimental Setup

We ran our experiments on nine nodes, composed of one master node and eight slave nodes. Each node has an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz (six cores with hyper-threading), 32GB DDR4 RAM, and 2 TB 7200RPM HDD. In our environment, we used Ubuntu with 4.4.0 Linux kernel version, OpenJDK 1.8.0_191 64bit, Hadoop 2.7.3, and Spark 2.3.3. We ran Spark from the master node in yarn-client mode, with 24 executors, 2 GB driver memory, 10 GB executor memory, and three threads per executor. We used 19.11 version and set SciDB as the default settings with 24 instances.

[1]https://rasterframes.io/

B. Raster Data Processing

Datasets: We used two different raster datasets: *SDSS* from astronomy surveys [2] and SeaWiFS L3 Chlorophyll, *CHL* [8]. As the SDSS website offers the images encoded in the FITS file format [32], we converted them into a different data format that comparison systems can understand, such as NetCDF, CSV, and TIFF. Spangle can load the NetCDF and CSV format, and we used NetCDF format in this experiment. Each scanline has observed five filters (broad bands), called *u g r i z*, with each consisting of 2048 by 1489 pixels basic units. CHL has three dimensions (*longitude, latitude, and time*) and one attribute (*chlorophyll*). Each cell is an eight-day average of *chlorophyll* of the earth with a resolution of 9 km × 9 km.

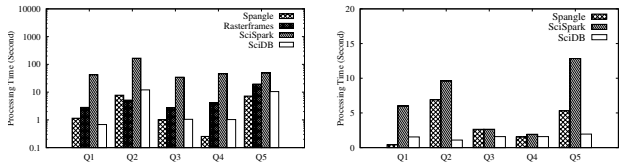
Query: Table I includes benchmark queries for raster data processing. We refer to a scientific data processing benchmark [33], designed with collaboration from domain experts. This benchmark consists of nine queries submitted on: recooking on the raw data, observation data, and the observation groups. Because comparison systems, SciSpark and RasterFrames, do not execute all queries by using their provided APIs, we select five queries and rewrite them to fit into the systems. All queries select cells in a specific area of interest to users, determined by a range $[X_1, Y_1]$ and $[X_2, Y_2]$ such that $X_1 \leq X_2$ and $Y_1 \leq Y_2$. We measure the processing time with actions such as count() since Spark evaluates map operations when calling actions.

Result: We first evaluate the performance of the systems for the raster data processing. Comparing Spangle with other systems, we consider their ability to process the benchmark queries. To load NetCDF files, SciSpark first loads data in a dense format and then splits them, which makes it difficult to hold large-scale arrays. If the size of an array is too large, it can fail to load data before distribution. In addition, SciSpark supports only a few APIs to process images (raster data). We implemented functions that process benchmark queries based on APIs of SciSpark. Besides, we transformed data from FITS to TIFF because RasterFrames understands the TIFF file format. We added an extra dimension for images, where the results of the four systems were equal.

We implemented a method to load data in Spangle with Unidata libraries[2]. SciSpark and RasterFrames cannot load all images. SciSpark manages data as dense, which requires more memory than Spangle. RasterFrames can reduce data by compression for sparse data, but it reads them in the master node and spread them to workers. Because of these limits and fair comparison, we used the maximum amount of data commonly processed in our environment for four systems and excluded the data ingesting time.

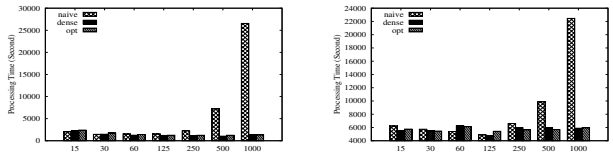
Figure 7a shows the results of queries without a range query using 100 images, while Figure 7b includes a range query using 1,000 images. In this experiment, we set the chunk size to 128×128×1. RasterFrames can support geometry operations with geometry libraries; however, it often yields incorrect results. Because we do not entirely trust the results of queries

[2]https://www.unidata.ucar.edu/software/netcdf-java/



(a) Without range query (b) Using 1,000 images

Fig. 7: Comparing raster data processing systems



(a) Filter (b) Aggregate

Fig. 8: The processing time along with the chunk size

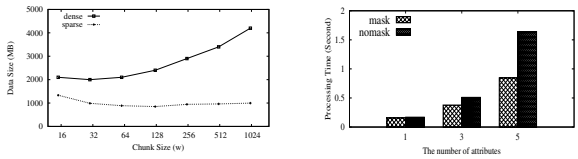
for that reason, we do not use only a range query to compare the four systems in Figure 7a.

Figure 7a shows that Spangle achieves a great performance, except for Query 2. Spangle effectively manages and processes sparse arrays without converting them into dense arrays. It minimizes the network overhead incurred by the aggregation operation, as managing sparse data at the optimal size helps reduce the overhead. Similarly, operations (*e.g.*, windowing) that compute values with adjacent cells may lead to network overhead, as those cells at the boundary are distributed across workers. Spangle supports overlap, described in Section III-A, which avoids data exchange for those operations. We used this function for the second and fifth queries. As SciDB is implemented from the scratch based on C++, we expected it to be the fastest. Because SciDB pushes down queries, it can reduce disk I/O overhead. However, queries such as Q2 and Q5, which require computations, are relatively slow.

In Query 2, Spangle is slower than RasterFrames. When loading data for regridding, RasterFrames must previously fit the chunk size into the target grid (*e.g.*, 3×3). This is not flexible for other operators but beneficial because it does not need to reshape the chunks. In contrast, Spangle can set the chunk size regardless of the target grid. It reads a specific range of a chunk for the target grid, which incurs computation overhead for data access. In Figure 7b, Spangle outperforms SciSpark. SciSpark manages sparse arrays as dense. It requires more memory and may incur overhead when shuffling data.

Next, we evaluate the data size and processing time with a sparse dataset between the dense and sparse modes described in Section IV-A. Since the chunk size has an impact on the performance, we varied it in this experiment. CHL stores the average value every eight days. We fixed one at the time dimension and set the latitude and longitude as the given length, w. That is, the chunk size is $w \times w \times 1$, where w is varied from 16 to 1000. To measure the processing time, including data access time, we used Filter and Aggregator that access all valid cells.

Figure 8 shows that the processing time in dense and sparse



(a) Data size by modes (b) MaskRDD performance

Fig. 9: The data size and processing time

modes. The `naive` is the sparse mode that counts bits until a specific position that operates each time. The `dense` is the dense mode that manages an array as dense, and accesses a cell in constant time by using an array index, whenever the chunk size increases. The `opt` indicates the optimal bitmask operations in the sparse mode, described in Section IV-B. With the increasing chunk size, naive takes a substantial time, compared with others. It reads the whole words for each position, which becomes the main factor in accessing cells and degrades the performance. In contrast to dense, opt does not outperform but shows the comparable performance. However, all methods do not achieve the best performance if the chunk size becomes smaller. It seems that splitting a large array as small chunks can have higher parallelism, but the scheduling overhead affects the total processing time more than the parallel processing.

Figure 9a shows the data size in memory for each mode with increasing the chunk size. The dense mode substantially increases along with the chunk size, but the sparse mode maintains a relatively similar size regardless of the chunk size. In the dense mode, it is necessary to store invalid cells in a payload, which increases the data size. In both modes, with a small chunk size, the data size decreases. If a chunk is empty, Spangle drops it, which leads to reducing memory space. However, as the probability of existing empty chunks decreases before 64, the number of chunks increases accordingly. Overall, while the optimal chunk size varies with the data distribution and density, the sparse mode has a significant contribution to reducing the data size.

Figure 9b shows the effect of the MaskRDD using the Q5 query. The x-axis is the number of attributes. In this experiment, we used five bands *u, g, r, i*, and *z* as attributes. When employing the MaskRDD, the changes are evaluated lazily. On the other hand, without the MaskRDD, all changes are evaluated eagerly. That is, all masks in each attribute are collected, and the *AND* operation is executed between bitmasks. When using an attribute, the performance between the two is similar. However, by increasing the number of attributes, the difference is distinguishable. With the MaskRDD, the processing time increases linearly, but without the MaskRDD, it takes a significant amount of time to evaluate all attributes. The MaskRDD consumes memory space, but improves the performance when Spangle has attributes more than one attribute.

C. Machine Learning

Datasets: Table II shows the datasets used in this experiment. In Table IIa, we used relatively sparse matrices for

Dataset	Matrix Size	Density
Covtype [34]	581K×54	0.218
Mouse [35]	45K×45K	0.014
Hardesty [35]	8M×8M	6.4e-7
Mawi [35]	129M×129M	9.3e-9

(a) Matrix Datasets

Dataset	Edge	Vertex
Enron [36]	367K	36K
Epinions [36]	508K	75K
LiveJournal [36]	69M	4.9M
Twitter [37]	1,468M	61.6M

(b) Graph datasets

Datasets	# of Rows		Feature
	Training	Test	
URL reputation [34]	1.9M	479K	3.2M
KDD Cup 2010 [38]	8.4M	510K	20M
KDD Cup 2012 [38]	120M	30M	55M

(c) Logistic regression datasets

TABLE II: Machine learning datasets

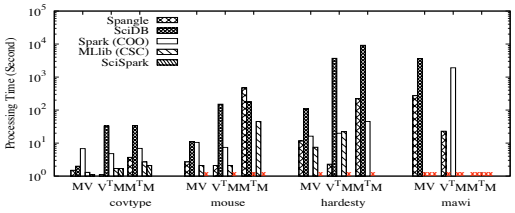
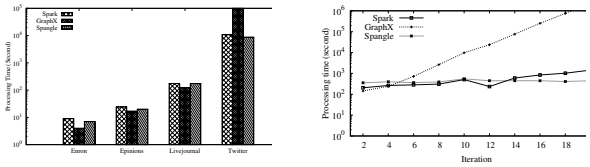


Fig. 10: Machine learning core operations



(a) Three Systems (b) Twitter

Fig. 11: The processing time in PageRank

ML core operations and generated vectors that consist of random values for matrix-vector multiplications. In Table IIc, we randomly split the datasets 80-20 to evaluate the test score for logistic regression.

Result: In this section, we compare the systems in ML core operations. Then, we evaluate Spangle with built-in modules on Spark using PageRank and logistic regression.

ML algorithm core operations. To understand the performance of machine learning algorithms, we first compared five systems which support linear algebra operations: Spangle, SciDB, Spark (COO), MLlib (CSC), and SciSpark. These are variants of machine learning core operations, and we summarize them as follows: matrix-vector multiplications (M×V and V^T×M) and transpose-self matrix multiplication (M^T×M). We exclude element-wise and scala-matrix operations because they incur embarrassingly parallel workloads in the map-reduce environment. Most machine learning algorithms, such as logistic regression and support vector machine, include M×V or V^T×M. The algorithms are often expressed as M^T×M (*e.g.*, principal component analysis).

Figure 10 shows the processing time for each system. The **x** mark indicates that a system could not process an operation because it occurred out of memory error or did not finish in bounded time. As the transpose operation is expensive, most systems take a long time V^T×M, compared with M×V. Spangle is not always the fastest, but it achieves excellent performance and shows scalability. Especially, it can process the multiplication for the large-size matrix (*i.e.*, Mawi) by optimizing the local join and the transpose operation for a vector, described in Section VI. The COO format can process Hardesty, but it fails to process Mouse. The number of non-zero elements (*i.e.*, density) affects the performance rather than the matrix size.

Most systems fail to compute $M^T M$ because the operation is too expensive, including matrix transformation and matrix multiplication. As SciDB is the disk-based database system, it takes a long processing time to process a huge matrix because it incurs disk I/Os for temporal data that do not fit in memory. In the Mawi datasets, it did not complete in the bounded time.

In summary, the major factor of the performance in matrix operations is not only computational algorithms but also the data size. As the volumes of intermediate data generated during processing operations are enormous, the processing of large-scale data should be considered.

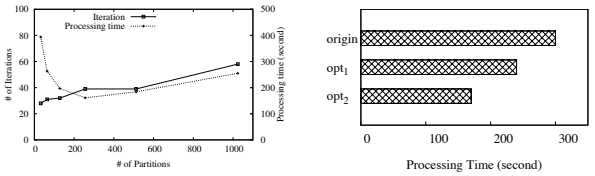
PageRank. Without a built-in module (*i.e.*, GraphX), we can implement PageRank using Spark APIs [39]. Figure 11 shows the performance of PageRank for Spangle, Spark, and GraphX. We ran 20 iterations and measured the processing time for both the end-to-end and each step, and all RDDs were cached. To achieve the best performance of Spangle, we apply the sparse mode to three datasets, Enron, Epinions, and Twitter, and the super-sparse mode to Livejournal. Compared with the graph model, the matrix may consume significant space, growing along with $O(n^2)$, where n is the number of vertices.

Owing to the partitioning strategy and sparse data management technique, Spangle shows a similar performance compared with two systems in relatively small data. Spangle is slightly faster than Spark, and slower than GraphX in three datasets, namely, Enron, Epinions, and Livejournal. In Twitter, however, Spangle achieves the best performance. In contrast to Spangle, the time for each iteration of GraphX increases. GraphX manages tripletRDD used to join the VertexRDD and EdgeRDD. Caching the EdgeRDD (large-scale data) causes increasing data size along with iterations, and GraphX occurs a shuffle to send a message from EdgeRDD to VertexRDD. During this process, a new RDD is created to reduce the data size for this shuffling. If Spark spills an RDD but needs to reuse it soon, it re-generates that RDD by lineage. This occurs at every iteration, which leads to doubles in the processing time.

SGD algorithm. In this experiment, we compared Spangle and MLlib which is a built-in module on Spark. Both systems are on Spark and provide the logistic regression function. In Spangle, we investigate and compare the performance by varying the number of partitions, a parameter of the distributed SGD algorithm. For both systems, variables for the algorithm, such as *tolerance and step size*, are equally set. The tolerance is 0.0001, and the step size is 0.6. We used two metrics,

Datasets	URL Reputation		KDD CUP 2010		KDD CUP 2012	
	time (s)	acc (%)	time (s)	acc (%)	time (s)	acc (%)
Spangle	193.7	94.26	32.9	86.62	1776.1	95.55
MLlib	185.6	94.21	-	-	-	-

TABLE III: The performance comparison using three datasets



(a) Iterations and processing time (b) Optimization for SGD

Fig. 12: The processing time of logistic regression

accuracy (in short, *acc*) and training time, to evaluate the two systems. The accuracy denotes the percentage of correct answers, and the training time does not include the data ingestion.

Table III shows the performance of Spangle and MLlib. We expected that two systems can process all cases, but Spangle only completed. MLlib fails to ingest two larger datasets, incurring out of heap memory. For this reason, we can only compare the performance of the two systems using URL reputation. To fairly compare them, we fixed the accuracy above 94.2% and measured the training time, as both two systems can frequently obtain the accuracy in this dataset. This result shows that the customized algorithm for Spangle is comparable. However, Spangle has the challenge of achieving more precise accuracy, as we do not yet implement a highly optimized algorithm, such as the *Adagrad* algorithm.

Figure 12a shows the relationship between the number of partitions and the processing time. In this experiment, we used the URL reputation dataset. A small number of partitions leads to a low parallelism. Meanwhile, a large number of partitions incur network overhead for the reduce operation. Figure 12b shows the performance of Spangle using our optimization technique.

We investigate the two-step optimization introduced in Section VI-C. The $\texttt{opt}_1$ is a customized equation (3). The $\texttt{opt}_2$ is not to execute the transpose operation for a vector but to replace the description, instead. Specifically, the opt_1 optimization reduces the computational cost. This decreases the processing time by 20%. Instead of transposing a training set, the transposing vector, which is relatively small and transformed for each iteration, has less overhead. The opt_2 optimization avoids the computation cost, as Spangle only replaces the description, not the physical layout. It improves the performance of Spangle, by approximately 30%. Overall, this optimization enhances the SGD algorithm and improves its performance by approximately 43%.

VIII. Related Work

There have been database systems and data-intensive computing platforms for scientific analytics and machine learning. Several systems are based on the array data model, such as RasDaMan [5] and SciDB [6]. SciDB is a popular array database system based on a multi-dimensional array model and can manage ragged arrays. SciDB is not entirely designed to store sparse arrays, as it does not support specialized techniques or data structures for them. Since array database systems are disk-based systems using SQL or SQL-like language, they are less flexible than map-reduce. The ML library is also insufficient for the community version. On the other hand, Spangle mainly executes in-memory computations in a map-reduce environment with sparse array management.

In addition, several systems based on Apache Spark have emerged for scientific datasets in recent research. SciSpark [31] provides APIs that abstract scientific data (*i.e.*, NetCDF and HDF) using linear algebra libraries. However, it has the mapping information inside a wrapper class and provides few array operations. Users need to understand the system and implement their operations. Especially in linear algebra, SciSpark does not provide the matrix multiplication in a distributed environment. RasterFrames can process spatial data based on Dataframe of Spark SQL. Using DataFrame, it can deliver a set of functionalities, horizontally scalable for general analysts and data scientists. GeoSpark [40] is a spatial data processing system on top of Spark. It can process geometric type queries using three abstracted RDDs. However, Spangle is a general-purpose array based system that can process scientific datasets as well as a large-scale matrix. It supports declarative APIs to manipulate arrays for ease of use and effectively manages sparse arrays with a bitmask.

As many scientists have employed arrays to represent matrices, a few systems have supported linear algebra and machine learning. There are numerous systems on top of Apache Spark, such as MLlib [12], SystemML [41], and MLI [42]. MLlib is an embedded library that provides machine learning, but a pre-canned distributed implementation for machine learning. This style is also a common approach in systems such systems: Mahout [43], MADlib [44], and H2O [45]. SystemML specifies machine learning algorithms at a high-level with a declarative machine learning language. Several systems also follow this style: OptiML [46] and DMaC [47]. Furthermore, MLI provides linear algebra APIs to build machine learning algorithms. This type is widely used, such as TensorFlow [48] and PyTorch [49]. Spangle also follows this design, but it mainly provides array and matrix operators optimized by bitmasks. In particular, it supports bitmask-based matrix operations.

IX. Conclusion

In this paper, we introduce Spangle, an array processing system that provides declarative interfaces. It facilitates high-level programming for both iterative algorithms and interactive analysis. This high-level programming system significantly increases the productivity of data scientists, as it is easy to manipulate arrays without considering the mechanisms involved. In addition, Spangle also supports machine learning. Most of them rely on linear algebra, and Spangle effectively

provides matrix operations. Our experiments show that Spangle is a competitive system compared with existing ones. It can accelerate scientific analysis and enhance scalable array processing on Spark with a massive amount of raster data. Furthermore, our optimization technique can accelerate the creation of a more accurate machine learning model.

REFERENCES

[1] L. Peng and Y. Diao, "Supporting Data Uncertainty in Array Databases," in *Proceedings of the 2015 ACM SIGMOD Conference*. Melbourne, Victoria, Australia: ACM, 2015, pp. 545–560.

[2] Sloan Digital Sky Survey Website, "Sloan Digital Sky Survey Data," https://www.sdss.org/, 2021.

[3] T. M. Website, "Vessel Traffic Data," https://marinecadastre.gov/ais, 2021.

[4] "ArcGIS hompage," https://www.esri.com/.

[5] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann, "The Multidimensional Database System RasDaMan," in *ACM SIGMOD Record*, vol. 27, ACM. NY, USA: ACM, 1998, pp. 575–577.

[6] P. G. Brown, "Overview of SciDB: Large Scale Array Storage, Processing and Analysis," in *Proceedings of the 2010 ACM SIGMOD Conference*. Indianapolis, indiana, USA: ACM, 2010, pp. 963–968.

[7] R. Rew and G. Davis, "NetCDF: an Interface for Scientific Data Access," *IEEE computer graphics and applications*, vol. 10, no. 4, pp. 76–82, 1990.

[8] NASA Goddard Space Flight Center, Ocean Ecology Laboratory, Ocean Biology Processing Group, "SeaWiFS Ocean Color Data," https://oceancolor.gsfc.nasa.gov/data/seawifs/.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[10] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," in *Proceedings of the 24th ACM SOSP Conference*. ACM, 2013, pp. 423–438.

[11] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark SQL: Relational Data Processing in Spark," in *Proceedings of the 2015 ACM SIGMOD Conference*. ACM, 2015, pp. 1383–1394.

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

[13] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph Processing in a Distributed Dataflow Framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 599–613.

[14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, 2012, pp. 2–2.

[15] A. Shoshani and D. Rotem, *Scientific data management: challenges, technology, and deployment*. CRC Press, 2009.

[16] P. A. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-Pipelining Query Execution," in *CIDR*, vol. 5, 2005, pp. 225–237.

[17] K. Kara, K. Eguro, C. Zhang, and G. Alonso, "ColumnML: Column-Store Machine Learning with On-The-Fly Data Transformation," *Proceedings of the VLDB Endowment*, vol. 12, no. 4, pp. 348–361, 2018.

[18] E. Soroush, M. Balazinska, and D. Wang, "Array Store: a Storage Manager for Complex Parallel Array Processing," in *Proceedings of the 2011 ACM SIGMOD Conference*, 2011, pp. 253–264.

[19] P. Wegner, "A Technique for Counting Ones in a Binary Computer," *Communications of the ACM*, vol. 3, no. 5, p. 322, 1960.

[20] D. E. Knuth, "Bitwise Tricks & Techniques, The Art of Computer Programming, 4," 2009.

[21] W. Muła, N. Kurz, and D. Lemire, "Faster population counts using AVX2 instructions," *The Computer Journal*, vol. 61, no. 1, pp. 111–120, 2017.

[22] P. Baumann and S. Holsten, "A Comparative Analysis of Array Models for Databases," in *Database theory and application, bio-science and bio-technology*. Springer, 2011, pp. 80–89.

[23] L. Libkin, R. Machlin, and L. Wong, "A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques," in *ACM SIGMOD Record*. NY, USA: ACM, 1996, pp. 228–239.

[24] A. P. Marathe and K. Salem, "A Language for Manipulating Arrays," in *Proceedings of the 23rd VLDB Conference*, San Francisco, CA, USA, 1997, p. 46–55.

[25] C. D. Tomlin, *Geographic Information Systems and Cartographic Modelling*. New Jersey, US: Prentice-Hall, 1990.

[26] R. Cornacchia, S. Héman, M. Zukowski, A. P. Vries, and P. Boncz, "Flexible and Efficient IR using Array Databases," *The VLDB Journal*, vol. 17, no. 1, pp. 151–168, 2008.

[27] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge university press, 2004.

[28] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent," in *Proceedings of the 17th ACM SIGKDD Conference*. ACM, 2011, pp. 69–77.

[29] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, K. Olukotun, and A. Y. Ng, "Map-Reduce for Machine Learning on Multicore," in *Advances in neural information processing systems*, 2007, pp. 281–288.

[30] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized Stochastic Gradient Descent," in *Advances in neural information processing systems*, 2010, pp. 2595–2603.

[31] R. Palamuttam, R. M. Mogrovejo, C. Mattmann, B. Wilson, K. Whitehall, R. Verma, L. McGibbney, and P. Ramirez, "SciSpark: Applying In-Memory Distributed Computing to Weather Event Detection and Tracking," in *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 2015, pp. 2020–2026.

[32] D. C. Wells and E. W. Greisen, "FITS-a flexible image transport system," in *Image Processing in Astronomy*, 1979, p. 445.

[33] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, S. Madden, M. Stonebraker, S. B. Zdonik, and P. G. Brown, "SS-DB: A Standard Science DBMS Benchmark," in *XLDB2010*, 2010.

[34] Center for Machine Learning and Intelligent Systems, "UCI Machine Learning Repository," https://archive.ics.uci.edu/ml/index.php, 2021.

[35] Texas A&M University, "SuiteSparse Matrix Collection," https://sparse.tamu.edu/, 2021.

[36] Stanford University, "SNAP: Stanford Network Analysis Project," http://snap.stanford.edu, 2021.

[37] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*. New York, NY, USA: ACM, 2010, pp. 591–600.

[38] SIGKDD, "KDD Cup," https://www.kdd.org/kdd-cup/, 2021.

[39] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark: Lightning-Fast Big Data Analytics*. " O'Reilly Media, Inc.", 2015.

[40] J. Yu, J. Wu, and M. Sarwat, "A Demonstration of GeoSpark: A Cluster Computing Framework for Processing Big Spatial Data," in *32nd IEEE ICDE Conference*. IEEE, 2016, pp. 1410–1413.

[41] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "SystemML: Declarative machine learning on MapReduce," in *27th IEEE ICDE Conference*. IEEE, 2011, pp. 231–242.

[42] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. E. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, "MLI: An API for Distributed Machine Learning," in *ICDM*, H. Xiong, G. Karypis, B. M. Thuraisingham, D. J. Cook, and X. Wu, Eds. IEEE Computer Society, 2013, pp. 1187–1192.

[43] "Apache Mahout," http://mahout.apache.org.

[44] J. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li *et al.*, "The MADlib Analytics Library or MAD Skills, the SQL," *Proceedings of the VLDB Endowment*, 2012.

[45] "H2O," https://www.h2o.ai/.

[46] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun, "OptiML: an implicitly parallel domain-specific language for machine learning," in *ICML*, 2011.

[47] L. Yu, Y. Shao, and B. Cui, "Exploiting matrix dependency for efficient distributed matrix computation," in *Proceedings of the 2015 ACM SIGMOD Conference*, 2015, pp. 93–105.

[48] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[49] "PyTorch," https://pytorch.org//.