Ph.D. DISSERTATION

Large-Scale Array Processing and Management in Distributed Systems

분산시스템에서 대규모 배열 처리와 관리

AUGUST 2022

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Sangchul Kim

Ph.D. DISSERTATION

Large-Scale Array Processing and Management in Distributed Systems

분산시스템에서 대규모 배열 처리와 관리

AUGUST 2022

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Sangchul Kim

Large-Scale Array Processing and Management in Distributed Systems

분산시스템에서 대규모 배열 처리와 관리

지도교수 문 봉 기
이 논문을 공학박사 학위논문으로 제출함

2022년 4월

서울대학교 대학원

컴퓨터 공학부

김 상 철

김상철의 공학박사 학위 논문을 인준함

2022년 6월

위 원 장:	박 근 수	(인)
부위원장:	문 봉 기	(인)
위 원:	심 규 석	(인)
위 원:	엄 현 상	(인)
위 원:	정 연 돈	(인)

Abstract

Scientific observation, simulation, and experiments produce large amounts of scientific data. With such scientific data represented as multi-dimensional arrays, there has been a need to enhance complex analytics. The array data model is appropriate for scientific data management because it can physically cluster data close to each other, which ensures the locality of coordinate systems. In this dissertation, we focus on large-scale array processing and management based on the array data model. We present *Spangle* implemented on top of Apache Spark, a popular map-reduce framework for complex computation workloads. By adopting the array data model, Spangle facilitates scientific analysis using raster data and machine learning algorithms heavily relying on linear algebra. Moreover, we employ SciDB, a popular array-based DBMS, to improve array processing and usability. First, we present an efficient approach to query processing in the *Filter* operator, which examines attributes and coordinates with a full scan regardless of given conditions. This approach enables the filter operator to perform a selective scan using array indexes for given spatial information. Next, we propose the scalable loader, *SLS*. It streamlines the conversion process and modifies the distribution method in the loading stages. Also, it eliminates two heavy-duty steps: sort and redistribution, which account for a dominant portion of data loading. Last, we propose *SDF*, which facilitates data sharing and exchange to support complex analytics with minimal integration overhead. By adopting the principles of a federated database system, SDF abstracts away integrating processes while retaining the primary authority of each database and preserving system features such as analytics libraries.

keywords: Array data model, Scientific data, Array Processing, Array Management
student number: 2015-30267

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the advancement of data collection and storage technologies, scientific technology development has made remote explorations effortless over the past few decades, which has led to the prevalence of satellite applications, such as a weather forecast system, red tide detection, and global warming estimation. Also, high precision and resolution sensors generate diverse types and a high volume of raster data in scientific experiments and observations. They produce large-scale raster data in various fields, including ecology, climatology, and astronomy. For instance, the Large Synoptic Survey Telescope (LSST), built to survey the night-time sky, collects 60PB of images over ten years [1]. Consequently, the need for managing and processing unprecedented sizes of scientific data, especially raster data, has emerged.

As scientific data have increased, scientists need sophisticated scientific analysis, which leads to high computational cost. For example, satellite imagery with various resolutions and coordinate systems is often transformed and rigid to investigate the relationship between multiple satellites. With those conversions, they employ mathematical models or domain-specific algorithms based on linear algebra operations (*e.g.*, matrix multiplication, covariance, and inverse). The analysis using scientific data plays an important role in understanding scientific phenomena.

Meanwhile, the growing demand for large-scale machine learning has been changing the landscape of data analysis. The size of training datasets has grown rapidly and affected the accuracy of machine learning. Given an efficient algorithm, increasing the training data size is one of the ways to minimize the bias in machine learning. Besides, many machine learning algorithms heavily rely on algebraic computations (*e.g.*, matrix calculations for logistic regression), and thus they are concerned with operations on matrices.

The array data model is considered appropriate to manage and process scientific data represented as multi-dimensional arrays rather than a traditional relational data model [2]. This dissertation focuses on array processing and management to enhance scientific analysis.

1.1 Motivation

The research in this dissertation has been motivated by the efficient support of data management and processing for large-scale arrays, especially multi-dimensional data such as raster data and matrices. As the raster data (*e.g.*, remote sensing imaginary) and matrices are widely used in scientific fields, improved array processing and management can contribute to the scientific analysis. Furthermore, a matrix can be seamlessly represented as two-dimensional arrays, which can be treated based on the array data model. Most machine learning algorithms highly employ linear algebra, which matrix operations can express. In traditional approaches, experts adopt domain-specific algorithms for data processing methods. However, the machine learning can be another option, learning rules based on scientific data for even non-experts. Recent studies [3, 4] show that the machine learning approach enhances the scientific analysis.

In order to support efficient scientific analysis, we have researched to improve large-scale array processing and management on distributed systems on Apache Spark and SciDB. Apache spark is highlighted as a scalable and parallel computing frame-

work. With increasing volumes of data, it is required for data-intensive large-scale analysis. In past decades, the array data model on Spark has not been studied in-depth, and few array-based systems [5, 6] have been introduced to support scientific data analysis. Moreover, as described above, the growing demand for large-scale machine learning has shifted methods for data analysis. There is a strong need for algorithms dealing with large-scale matrices that do not fit into 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.

Moreover, SciDB is one of the popular array-based DBMSes, which can facilitate large-scale scientific data management, but it is insufficient. We suggest approaches to leverage the performance and functionality for query processing, data loading, and database federation on SciDB. The filter operator of SciDB reads all data without considering features of array-based DBMSes and spatial information. It is especially inefficient when a user searches a small region in massive datasets. The other major hurdle is pre-processing before loading data, which includes extraneous file conversion from a raw data format to a software-specific format and array transformations for data locality.

In addition, scientific data are managed in multiple separate databases by different sources and organizations. When analyzing such distributed data together for further comprehensive understanding and prediction, scientists need to access data via multiple simultaneous connections or collect them in a single location. In the middle of the data analysis process, scientists usually employ aggregated results rather than the whole data. If SciDB provides this feature at the query level, scientists can extract the results from other databases without the cumbersome steps.

1.2 Contributions

The overall goal of this dissertation is to provide efficient large-scale array processing and management. In the research, we employ a processing framework, Apache Spark, and an array DBMS, namely SciDB. We implement an array processing system on top of Apache Spark. It extends Resilient Distributed Dataset (RDD) to support efficient array processing and reduces the training time in machine learning. In addition, we improve the query processing and loading performance on SciDB. We suggest the selective scan for filter operator and a scalable loader. The selective scan approach reduces the disk I/O cost, and the scalable loader mainly minimizes network communication overhead. We also suggest a feature which federates multiple databases at the query level.

Overall, the main contributions are as follows.

- 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.

- We present advanced database techniques to provide efficient scientific data processing and management. First, we provide a new implementation of a selective scan that retrieves data corresponding to a range that satisfies specific conditions. In our experiments, the selective scan approach is up to 30x faster than the original scan. Second, we propose a scalable loader, *SLS*, which reduces loading overhead. Our experiments show that SLS reduces the data loading time up to

65% compared with the vanilla loader of SciDB. Last, we present *SDF*, *Scientific Database in Federation*, which facilitates data sharing and exchange to support complex analytics with minimal integration overhead. *SDF* has two connection models, *master-to-master* and *cluster-to-master*, for a shared-nothing architecture.

1.3 Outline

The rest of the dissertation is organized as follows. Chapter 2 describes the background knowledge of the array data model and two systems, SciDB and Apache Spark. Also, we present the related work in this dissertation. Chapter 3 introduces the processing system, Spangle, for large-scale arrays on top of Apache Spark. Spangle provides advanced operators regarding raster data processing and machine learning algorithms. We describe two machine learning algorithms with customization, which can demonstrate efficient matrix operations. In Chapter 4, we present an approach to improve query processing using spatial information and data loading in SciDB. Also, we describe a federated database system, which enables data integration between different databases. Last, we conclude with our contributions and future directions in Chapter 5.

Chapter 2

Background

In this chapter, we describe background knowledge and underlying two systems, Apache Spark and SciDB. First, we describe the array data model. Next, we present the overview of SciDB and Apache Spark.

2.1 Array Data Model

An array, a collection of homogeneous elements and totally ordered, is a fundamental data model. Figure 2.1 shows terms in array data model. It is composed of consecutive elements, named *cell*s. A *dimension* is ordered, represented by a range of contiguous integer values, and a cell can have multiple values mapped into the same array index, which we call *attribute*. 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

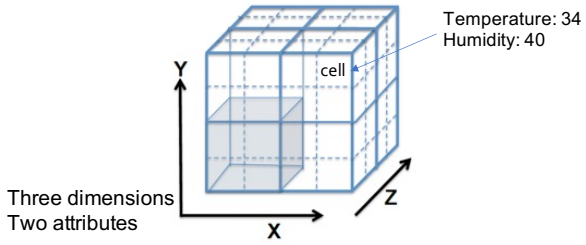


Figure 2.1: Array data model

(*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.

Array Based System

There are many array based systems to process and manage multi-dimensional data based on array data model. In this section, we describe array DBMSes and array processing systems. The systems related to Apache Spark are dealt with in another section.

HDF5 [7] is a popular file format for storing multi-dimensional data. It has two main structures: a group and a dataset. A group is a collection of objects in an HDF5 file, and a dataset consists of homogenous elements, which essentially follow the multi-dimensional array format. The attribute of HDF5 is metadata which describes the group and dataset. It is a portable and extensible format using a hierarchical structure by self-describing structures. In addition to HDF5, netCDF [8] is widely used, which provides a portable file format and an application programming interface (API) to store and retrieve netCDF files over heterogeneous platforms.

Titan [9], a parallel shared-nothing architecture, manages remote-sensing raw data. Its coordinator stores metadata using an R-tree index. The range queries, which retrieve spatio-temporal data, require input data to map into a grid by an aggregation query in a relational database. As each worker has a plan to read chunks for an aggregate query, it only considers the task distribution for load balancing related to the communication and I/O overhead.

T2 [10] is a customizable parallel database that provides the retrieval and processing of multi-dimensional data. It can manage both uniform and non-uniform datasets and support the most general form of array operations. Users can specify the functions of the system at query time.

RasDaMan [11] is a DBMS for multi-dimensional arrays. It has a client-server architecture that a server runs on a single machine and is a middleware on top of a relational database with BLOB storage. It provides a declarative query language for the general purpose. Because RasDaMan does not have native array storage, it processes array operations at the middleware.

SciDB [12] is a popular array database system based on a multi-dimensional array model and can manage ragged arrays. Section 2.3 describes it in more detail. ArrayDB [13] is an array database system prototype that implements the AML language. The plan of AML execution pipelines data and generates results at a time. We describe the AML language in more detail in the next section.

RAM [14, 15] implements the RAM array algebra on MonetDB. The array algebra operators can be assigned to relational algebra expressions. However, the array operators do not fit into the relation models, which incurs extra costs. Further, RAM is extended to a parallel setting [16]. It has two simple rules, partitioning and aggregation, to allow query decomposition. Partitioning distributes arrays and enables the query processing in parallel if there is no dependency between indexes. Similarly, in aggregation, RAM can process queries for partitioned arrays in parallel because they are not dependent on each other.

ArrayStore [17] is a storage manager that supports complex and diverse operations on sparse arrays. It can process queries in parallel in a shared-nothing architecture. ArrayStore presents three chunking strategies: regular chunking, irregular chunking, and a two-level combination of those. Regular chunking divides an array by the same shape, and irregular chunking splits an array into a chunk that has the same number of points. However, regular chunking takes a long time for data loading, but irregular chunking also requires explicit indexing in a parallel environment. It is more desirable when queries need to be processed in balance. Consequently, the two-level chunking strategy, which merges the two chunking strategies, is suggested to overcome the pitfall.

TileDB [18] is a storage manager for multi-dimensional arrays and can support both dense and sparse arrays. It organizes elements of an array into fragments which are ordered collections. The fragments allow transforming random writes into sequential writes, leading to very efficient reads. TileDB also supports parallelism via both multi-threading and multi-processing, providing thread safety and process safety, respectively, with lightweight locking. However, it does not support any array processing functions and distributed processing. ChronosDB [19] is a distributed geospatial multi-dimensional array DBMS, which supports in-situ processing. The data model of ChronosDB represents diverse raster data types and formats, and ChronosDB has a coordination layer on top of components, which leverages performance and functionality. SciHadoop [20] uses multiple partitioning to array data in HDFS and supports the optimized computation for holistic aggregates such as the median. However, each partition is bounded to process its corresponding chunk. A bottom-up partitioning is employed when fine-grained sub-arrays are created and merged into chunks. It minimizes the number of partitions across chunk boundaries. Also, SciHadoop reduces the overhead such as data transmission, remote reads, and unnecessary reads.

SIDR [21], implemented on SciHadoop, employs locality-aware access to scientific data in its original format. It extends the functionality that Reducers can determine

when the Mappers finish data processing for their result, which starts data processing much earlier. SCIMATE [22] enables the scientific data processing in different formats with a MapReduce like API. The API consists of functions submitted together with multiple optimizations. It improves the transformation process and accelerates the execution.

ArrayUDF [23] presents the User-Defined Function (UDF) mechanism for multi-dimensional arrays based on SDS [24] framework. It provides generalized structural locality support. Compared with the MapReduce computing model, ArrayUDF uses a single step to finish tasks for both Map and Reduce operations. It allows in-situ processing to access raw scientific file formats directly. HAMA [25] is a framework implemented on top of Hadoop for matrix operations, not for raster data processing. It supports varied matrix operations, such as matrix multiplication and the conjugate gradient method to solve linear equation systems. Matrices are stored in HBase, and their operations are implemented as Map-Reduce programs.

Array Query Languages

There are several proposed languages to identify an array algebra. They are designed as similar to SQL or typically array extensions to SQL. In this section, we discuss multiple array query languages in detail.

AQL [26] is a declarative query language and abstracts multi-dimensional arrays as functions from indexes to values. It supports various types, such as a boolean, tuple, and array. The AQL abstracts array operations for a higher-level language and adequately expresses the nested relational operations on multi-dimensional arrays. To streamline programming and enhance query optimization, the AQL operator can consist of components, such as comprehension, pattern, and block. It is similar to the relational algebra, which is extended by rewriting pipelined functions instead of mapping a higher-level language.

Rasdaman provides RasQL [27] based on its array algebra, similar to a SQL query.

It has three main constructs that can express all array operations: MARRAY, COND, and SORT. The MARRAY specifies each cell position of a spatial domain to create a new array, and the COND combines cell values and results in a scalar value. The SORT re-arranges cells for a specific order along the selected dimension of corresponding hyper-slices. It shuffles cells by the given order-generating function, which concatenates dimensional hyper-slices without transforming values and the spatial domain. These three operators are basic operators, used to create other operators, and new operators can be added to the algebra on demand.

The Array Manipulation Language (AML) [28] takes bit patterns as parameters, which can evaluate arrays. It consists of three operators to manipulate dense arrays, such as SUB, MERGE, and APPLY, and we describe the details below. Given a range query, the SUB, which is commutative across dimensions, retrieves corresponding cells. In two distinct dimensions, the subsequent SUB applications can produce the same result regardless of their order. The MERGE combines two arrays over the same domain, and the APPLY supports a user-defined function, which can produce a new array. Users can manipulate arrays by APPLY as their own function. Those three operators can be organized as nested, i.e., pipelined as arguments between upper-level operators.

RAM and SRAM [14, 15] support array processing on top of the MonetDB [29], which is a columnar relational database. RAM and SRAM, typically used for information retrieval applications, manage dense and sparse arrays, respectively. The SRAM array algebra improves RAM specification, which presents additional structural operators. Basically, As both systems do not support arrays but represent arrays as relations, they must process arrays over relational algebra operators, which leads to inefficient query processing.

SciQL [30] employs SQL-like syntax and declarative query language, which can abstract the data model from the physical data layout. SciQL can seamlessly integrate set, sequence, and array semantics and extend the value-based grouping to struc-

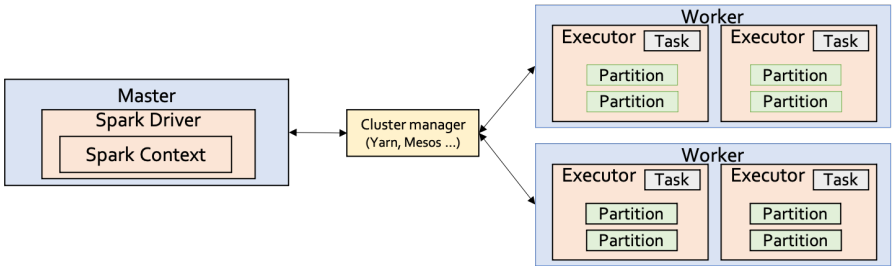


Figure 2.2: The architecture of Spark

tural grouping based on positional relationships for aggregation by selective access to groups of cells.

2.2 Overview of Apache Spark

Apache Spark [31] is a general-purpose framework for large-scale data processing with APIs in Scala, Java, and Python. It supports different computation types, such as job batches, iterative algorithms, interactive queries, and streaming. More recently, a number of high-level APIs have been developed based on Spark. With built-in modules for streaming data analysis [32], SQL [33], machine learning [34], and graph processing [35], Spark enables advanced in-memory big data processing and analytics in many fields such as finance and e-commerce industries.

Figure 2.2 shows Spark architecture. It uses the Resilient Distributed Datasets (RDDs) [36], each of which 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, a series of RDD operations. The transformations in a stage are not executed until the actions are triggered (*i.e.*, lazy evaluation). It enables Spark to optimize 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.

Apache Spark is natively designed for not only iterative processing, commonly well suited for complex analysis, but also interactive analysis, which can evaluate results step by step. It is a unified engine for different applications, which does not require learning and maintaining third-party tools. Moreover, it is a general-purpose framework for any purpose of applications from laboratory to production. Consequently, it can lead to higher productivity, especially when it is used for complex algorithms and pipelines.

2.3 Overview of SciDB

In this section, we describe the terminologies and architecture of SciDB. Then, we present the query language, operator, and data loading of SciDB.

Terminology

As SciDB is based on array data model, it uses its terms described in Section 2.1. An `instance` of SciDB is an independent worker, and a computing node can have one or more instances. A SciDB suite consists of a single master instance with several slave instances, and we call the node containing master instance a `master node`. A `chunk` is a physical unit of I/O for processing and inter-node communication [12]. It allows overlap between chunks to encourage region queries (e.g., Gaussian Smoothing).

System Architecture

SciDB [12] is a parallel DBMS based on a multi-dimensional array model which supports *ragged* arrays because the system does not require that arrays be rectangular. The model makes data objects that are close to each other to be stored together. SciDB uses

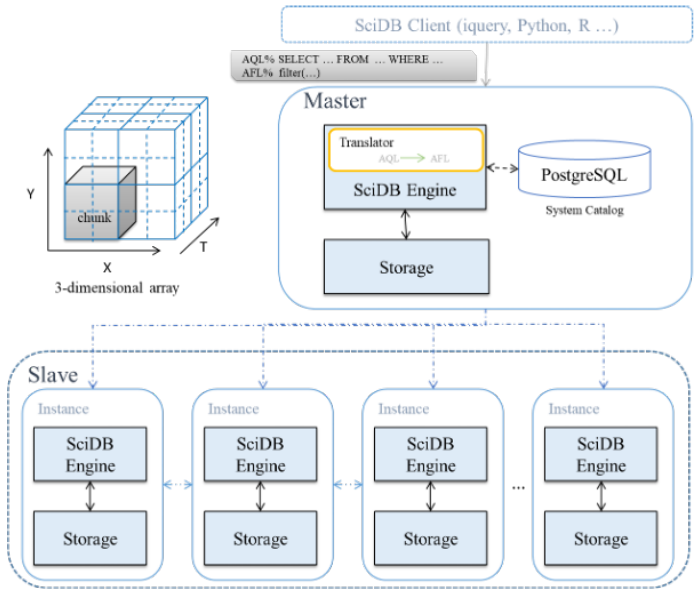


Figure 2.3: The architecture of SciDB

a shared-nothing architecture, where each node connected to the network can communicate with every other node and has its own independent storage.

Figure 2.3 illustrates the architecture of SciDB. The master node stores a system catalog, metadata of SciDB, in PostgreSQL [37]. SciDB partitions multi-dimensional arrays into chunks and stores the chunk metadata in the catalog, including boundaries and chunk lengths. SciDB does not store indexes explicitly because it stores the coordinates for every cell in the system catalog. Before processing queries, SciDB exploits the relevant information from the catalog, and the master broadcasts a query to SciDB instances responsible for managing a subset of arrays.

SciDB has a fixed size block model for a chunk because it is an advantage to managing and performing better between disk and main memory. The buffer pool with fixed-size slots collects data in main memory. If a chunk is too small, it could not avoid many reads. On the other hand, if a chunk is too big, cache miss and redundant I/O would occur because unnecessary data may move to read. Basically, chunks are partitioned and distributed to every instance using hash.

Given dimensionality (d) and a prime number (P), SciDB uses following the hash function to distribute data, H_d,

$$H_{i+1} = ((H_i \times P) \oplus (V_i - m_i))/l_i \quad \text{(A)}$$

where l is a chunk length and, V and m are coordinates of a chunk and the smallest coordinates for each dimension, respectively. Initially, $H_0 = 0$ and SciDB uses 1,013 as P. A hash value is calculated iteratively in the range $0 \leq i < d$. Then, the instance ID (I_ID) becomes the remainder divided by the number of instances (N).

$$I\_ID = H_d \,\%\, N \quad \text{(B)}$$

For instance, the hash value of coordinates $\{1, 1, 10\}$ is four when each starting coordinate is $\{1, 1, 1\}$, and each chunk size is $\{2, 2, 2\}$.

Query Language and Operator

There are two distinct query languages in SciDB: Array Query Language (AQL) and Array Functional Language (AFL). AQL is based on the grammar of SQL [26] and compiled into AFL at the query preparing phase. AFL is optimized for array calculations, derived from APL (A Programming Language). The primary data type of APL is a multi-dimensional array. Users can access SciDB operators through AFL, and operators such as $filter()$, $between()$ and $aggregate()$ can be cascaded to obtain desired results. For example, a query can be represented as *join(B, filter (A, x = value))*, where A and B are arrays, and x is an attribute of A. This example presents spatial join, which overlaps two arrays and merges attributes of two input arrays when dimensionality should be equal. In the query, the intermediate results of *filter* are delivered to a *join* operator.

SciDB has a unique operator, $redimension()$, which transforms dimension values to attribute values, and vice versa. This operator reshapes an array (for example, 100 by 100 to 1000 by 10) and transforms one or more dimensions. This step is compa-

rable to both indexing and clustering. While redimensioning, SciDB partitions multi-dimensional arrays into chunks with a specific size. It is imperative to determine the optimized chunk size because it affects the data size and data locality.

SciDB provides user-defined operators (UDO) for extensibility. UDO is useful for writing new operators accepting one or more arrays as arguments. Users employ UDO in various ways, including linear algebra, such as matrix multiply, curve fitting, and linear regression. They can implement algorithms composed of sequential or iterative logic with it. Many calculations for an algorithm consist of a few steps, and complex computations may proceed with various kinds of core calculations.

Data Loading in SciDB

The overall SciDB data loading process is explained in Figure 2.4. The master node converts raw data format (e.g., HDF5) to intermediate format (e.g., TXT) and then a Comma-Separated Value (CSV) format. The master node distributes data to all SciDB instances when the conversion finishes. We describe the detail of the steps below. Before sending data, the master node splits data for instances by a round-robin and sends them to instances. Then, SciDB instances convert CSV to Dense Loading Format (DLF), and DLF files are loaded into a one-dimensional array. Since the initial array format is a one-dimensional array after loading, redimension operator transforms the array to an appropriate format. These steps are described below in more detail.

Loading. To load data into SciDB, we must convert raw files into an input file format, which the SciDB loader can support. The first task is data split. We divide a file using round-robin and store them to separate files. The master node divides the file, and the SciDB loader distributes and loads them, alleviating I/O overhead in the master node. Data are distributed to instances using SSH and pipes, and each instance converts them into DLF format, which SciDB understands. After the conversion, SciDB instances load DLF files to a one-dimensional array.

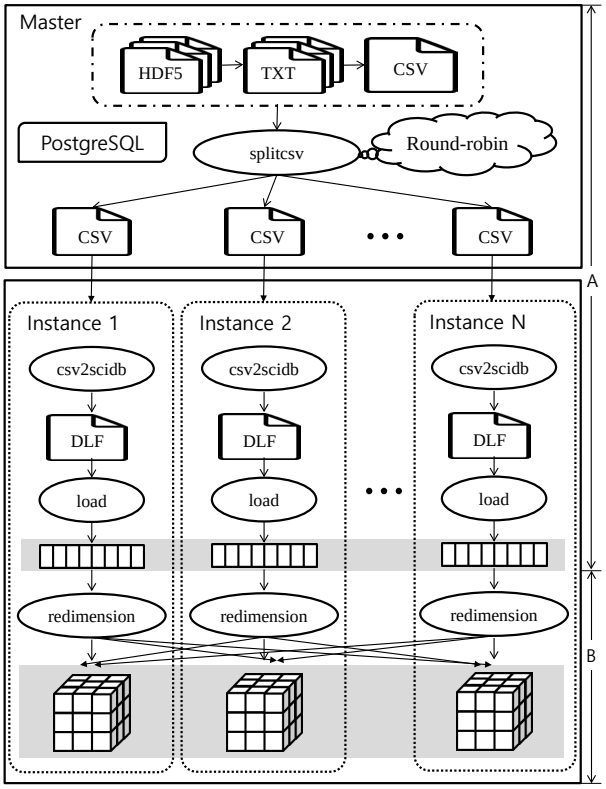


Figure 2.4: Data loading in SciDB: (A) Loading stage, (B) Redimensioning stage

Redimensioning. There are four major steps in the redimensioning stage. 1) data preparation: the first step involves converting the array into a one-dimensional form and matching each attribute to a corresponding coordinate. This conversion is pre-processing for the next step. 2) sorting all cells: this step accounts for half of the total redimensioning time. 3) preparing distribution: SciDB aggregates sorted elements into a temporal array before distributing data to actual target instances.4) distribution and synchronization: data are redistributed based on hash values and stored in corresponding instances. All instances communicate with each other to synchronize the final step.

2.4 Related Work

In this section, we describe related work to process and manage raster datasets. In Spark, there have been data-intensive computing platforms for scientific analytics and machine learning. In addition, several database management systems are proposed based on array data model. We present those systems and database techniques to improve data management.

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

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 [11] and SciDB [12]. 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 [6] 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 sci-

entists.

ClimateSpark [5] uses a multi-dimensional array-based data model on Spark and can organize climate datasets, commonly represented as multi-dimensional arrays. It generates a spatio-temporal index to avoid pre-processing steps and store the index to an RDBMS. ClimateSpark is designed for climate data analytics, but it does not provide the native array operators. As the configuration of this system is complicated and does not maintain now, we fail to evaluate its performance and identify its features. H5spark [38]can load HDF5 data natively into Spark, which provides a parallel I/O interface using an MPI-like independent I/O. It depends on the Lustre file system striping to achieve high I/O bandwidth. GeoSpark [39] 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 [34], SystemML [40], and MLI [41]. 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 [42], MADlib [43], and H2O [44]. SystemML specifies machine learning algorithms at a high-level with a declarative machine learning language. Several systems also follow this style: OptiML [45] and DMaC [46]. Furthermore, MLI provides linear algebra APIs to build machine learning algorithms. This type is widely used, such as TensorFlow [47] and PyTorch [48]. Spangle also follows this design, but it mainly provides array and matrix operators optimized by bitmasks. In particular, it supports bitmask-based matrix operations.

Advanced Database Techniques for Large-Scale Arrays

This section describes the previous work related to the systems and our three different approaches.

Optimizing Query Performance on Array DBMS. Improving query processing performance has been researched over the past decades. The common approach is to use a specific index, which allows reading data selectively. We describe previous research to process raster datasets by avoiding full scan.

SDS/Q [24] uses bitmap indexing to HDF5, which improves performance for highly selective queries. COMPASS [49, 50] supports value-based filter operations on SciDB. It maps elements into a number of bins specified by users and encodes bin-based indices, which enables access to cells by avoiding full scan. BitFun [51] provides bitmap indexing using a hierarchical bitmap index to continuously re-index arrays during queries. k^2-raster [52] is the variant of k2-tree [53]. It represents raster datasets and uses compressed space, offering indexing capabilities. This is not based on array data model, but is on top of the raster data to improve query and processing performance.

Our approach identifies spatial information in a query to use the positional index of an array, which can be accessed directly by coordinates. It first filters data by the spatial information and then retrieves data by other conditions.

Loading Data into Database in Parallel. Many existing database systems support parallel data loading. Greenplum [54] built on PostgreSQL supports parallel data loading with its external tables used with the extract, load, and transform process. Since the table supports SQL operations such as *SELECT*, *SORT*, and *JOIN*, the data can be extracted, loaded, and transformed (ETL) simultaneously. That process finally makes the data available in the target table. In addition, by using external tables and *gpfdist* that enables servers to connect directly to the external sources, Greenplum can give maximum parallelism and load bandwidth.

The loading process in Oracle [55] occurs in four main phases: 1) data loading with

external tables, 2) preparation, 3) loading, and 4) post-load activities. To begin with, an external table is employed, and a read-only table with metadata is stored in a database, which uses external data as a virtual table. By using a virtual table, queries are directly submitted, and data are loaded in parallel without loading the external data in the database. Before data loading, several tables and objects need to be prepared. The external data are processed in parallel to support the scalable loading in this phase. The third phase is loading data by a SQL command. Due to the second phase, a database environment is set for sufficient parallelism. Then, the statistics of data are computed to give an optimizer that relies on accurate statistics.

Like Oracle, Vertica [56] can also load data in parallel. A user submits a *COPY* statement, including a list of nodes to execute the loading operation. A single host controls the loading process on multiple nodes in the cluster. Raw files have a delimiter, and the Vertica loader splits the input data files on a delimiter. Then, the loader creates a new tuple for each line and redistributes the tuple to all nodes using hash. Once the data are loaded, the columns are sorted and compressed, which follows the physical design of the database.

Federated Database System. A federated database system was researched in depth in the 1980s and 1990s. From the study, several systems adopt the concept of the system for data integration. IBM suggested database federation through an early prototype named Garlic [57], whose architecture and optimization for cross-query have become fundamental components. A *wrapper* is used to federate both libraries and data, which supports multi-server integration by configuring metadata of multi-severs. In *SDF*, metadata of remote databases is managed by a link table in PostgreSQL, retrieved by *Address Manager*.

Several systems [58, 59] were researched for a join operation between Hadoop and a DBMS that has shared-nothing architecture. JEN [59], a join execution engine, has a similar connection model to our cluster-to-cluster model, but the cluster-to-cluster model is not restricted by the number of instances involved in both local and remote

clusters. Also, BigDAWG [60] is a system to integrate data from heterogeneous engines. This is a middleware solution because data go through islands, a middleware for a link between engines. These engines should communicate with each other via islands and have pre-defined schemas for federated data. However, *SDF* integrates data by its operators directly and generates schema by itself. BigDAWG federates heterogeneous engines (*e.g.*, from the relational model to the array model) but does not support data federation between array models.

The plugged operator or engine was developed to support database data federation, similar to *SDF*. PostgreSQL [37] has an extension, foreign data wrapper (FDW), which provides the database federation. FDW connects PostgreSQL with various kinds of data depicted in a tabular format, such as Oracle database tables, CSV files, and process lists. In MySQL [61], a local server has a federated table whose schema matches the table a local server attempts to access on the remote server. If a result set is produced, each column must be converted for the schema format of a federated table. While both two DBMSs require a federated schema to be defined in advance, *SDF* tailors it during the query execution time.

Chapter 3

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

With increasing volumes of scientific data, scientific analysis becomes heavy work that entails a long running time, which requires a scalable and parallel framework such as Apache Spark for desirable performance and productivity. Moreover, the growing demand for large-scale machine learning has changed the data analysis trend. Many machine learning algorithms heavily rely on algebraic computations (*e.g.*, matrix calculations for logistic regression). Thus, it is increasingly required to manage large-scale matrices, which are difficult to be processed in single machine memory. Many domain scientists need a scalable and efficient programming model for complex scientific analytics.

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 or-

ganize 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 chapter is organized as follows. In Section 3.1, we present Spangle and its architecture. Then, we describe the bitmask which is a Spangle component, and how it manages chunks and processes arrays in Section 3.2. Section 3.3 shows programming interfaces in Spangle. In Section 3.4, we optimize two machine learning algorithms for Spangle, and Section 3.5 shows the performance of Spangle. Last, Section 3.6 summarizes this chapter.

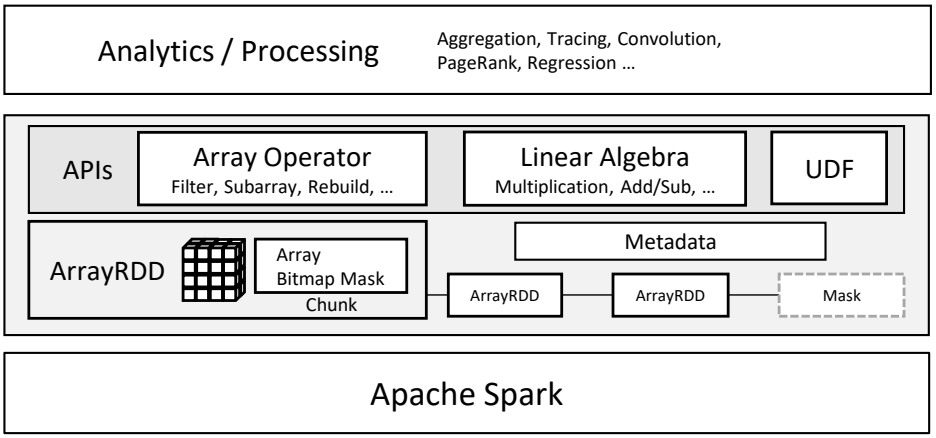


Figure 3.1: The architecture of Spangle

3.1 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.

3.1.1 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 Spangle is described in Figure 3.1. 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 3.2.1. 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 3.2.2.

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 [62]. 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 [29], suitable for in-memory processing systems. Even for machine learning, a column-store manner achieves high performance [63].

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. Then, it 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.

To support full parallelism, it prefers not to have the relationship between data. In array processing, however, the relationship between cells, which stands for an array index, is important. The adjacent cells in an array are often co-operated and aggregated, such as average and summation (e.g., convolution). Consequently, we give the Chunk ID to each chunk to identify its location. Spangle manages the pairs of (ChunkID, chunk) as a PairRDD. The PairRDD is a native data structure in Spark, which provides easy extension and compatibility for every Spark version. ArrayRDD inherits

PairRDD on Spark, and its element is (key, chunk), where the key is a single value representing the location of the chunk. By this feature, Spangle manages large-scale arrays or matrices partitioned across nodes and provides distributed array operations and linear algebra. The key has a similar role in map-reduce processing but has three different aspects for our design: First, the chunk ID explicitly represents the chunk location in an array. It provides information of its value (chunk). It is a kind of a chunk index. Second, for scalable dimensionality, the chunk ID should represent all dimension metadata. The single value representation is simple but gets a benefit from this scalability. Last, as the dimension is scaled, the size of metadata may grow. We need to reduce the metadata size because Spangle is an in-memory processing system.

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* [17] 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.

3.1.2 ArrayRDD

A chunk consists of two components, a `payload` and a `bitmask`, as shown in Figure 3.2. The payload is a collection of actual values, and the bitmask indicates their

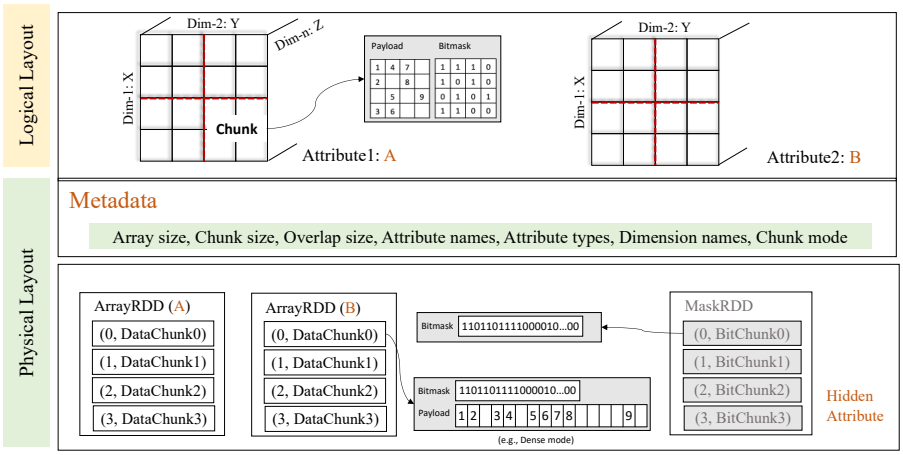


Figure 3.2: The ArrayRDD and chunk layout

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.

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 3.3, 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 3.5.2.

3.1.3 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 3.2. 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 3.4.3.

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
```

3.2 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 3.2.3.

3.2.1 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 [64]. 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 [65]. Likewise, in monitoring shipping vessels, data around the coastline are empty because they congregate near major ports [66].

Each array-based system [8, 11, 12, 67] 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

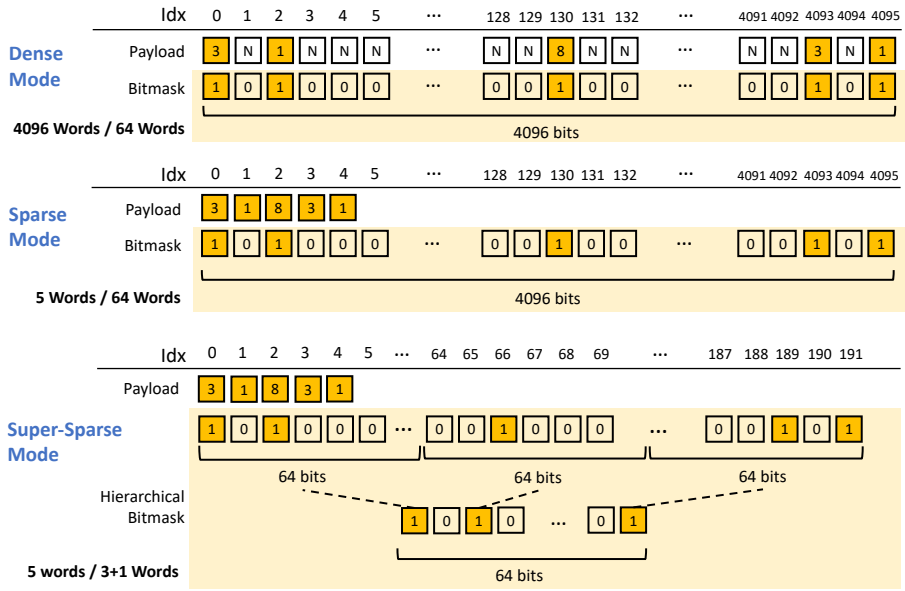


Figure 3.3: Three modes for data distribution

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 [68], 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.

3.2.2 Chunk Management

As the density of an array, Spangle manages chunks in three different modes: dense, sparse, and super-sparse, described in Figure 3.3. 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 in the performance. In addition, the bitmask can represent the static graph, described in Section 3.4.2.

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.

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 the bit-vectors 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. For instance, data can be super-sparse but follow a normal distribution. 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.

3.2.3 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 [69, 70] introduce the population count in a word, whereas a built-in function achieves the best performance in practice.

Bitmask Operation

Spangle employs bitmasks to represent states of cells, valid or not. As cells are compressed by removing null values, the population count is required [69, 70]. A succinct data structure provides rank and select queries with minimized space close to the theoretic lower bound using bit-vectors. Formally, in a vector, V[0..n-1] of the length n over an alphabet Σ of size σ,

$rank_c(V, i)$ returns the number of occurrences for symbol c in the prefix V[0..i-1].

$select_c(V, i)$ returns the position of the i-th occurrence for symbol c in V.

In Spangle, we are interested in the zero and one representation, Σ =0,1 and σ=2. For example, in the bit-vectors, `011011001`, $rank_1(V, 5)$=4 and $select_1(V, 2)$=2. The $rank_c(V, i)$ is implemented in population count, which counts the number of one bits on a word (V).

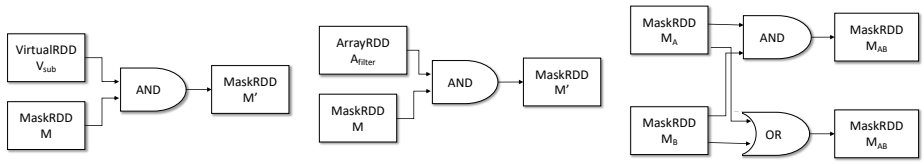
There are several algorithms [71] to compute rank and select in constant time. The previous research [72, 73, 74, 75] shows the rank and select are computed in constant time, using o(n) bits more space. It first divides V into large blocks of length, l=$(logn)^2$. Again, the large block is divided into the small blocks of length, s=(log n)/2. For the large block, we store L[i] = $Rank_1$(V, i·l - 1) and S[i] = $Rank_1$(V, i·s - 1) - $Rank_i$(V, (i - 1)·s - 1). Afterward, we build a P(x) table which stores the number of ones in x of length s. We can compute the rank in constant time by adding the values stored on those three tables. In practice, as CPUs support *popcount*, we do not need the lookup table.

The RRR algorithm [76, 77, 78] can replace V with an entropy compressed repre-

sentation without increasing the time bounds, used for sparse bit-vectors in practice. It is implemented as follows. Bit-vectors splits the sequence into blocks, u, which is (lgn)/2, and every block has a tuple (c_i, o_i). The c_i represents a class, which corresponds to its number of 1s, and o_i represents the offset of the block. The class of c_i is the number of ones in the i-th block; for example, if i-th block is 101, the c_i is 2. The offset o_i is the lexicographical rank among c_i. In addition, there are three tables defined as E, R, and S. Table E stores every combination of u bits, sorted by class, and it is sorted by offset within each class. It stores all candidates for rank at every position of each combination. Table R and S store the concatenation of all c_i combinations, using $\lceil \lg(u+1) \rceil$ bits per field, and o_i combinations, using $\lceil \lg\binom{u}{c_i} \rceil$ bits per field, respectively. This algorithm consumes $nH_0(V)$+o(n) bits.

SD-array [79] follows Elias-Fano representation [80, 81]. Let S[i] is the $select_1$(V,i) and t = lg(n/m). The n is length of V, and m is the number of set bits. L[i] is the leftmost t bits of S[i], and P[i] is the S[i]/2^t. We use H, which represents the unary codes of P[i]-P[i-1], to reduce the space, and 2+$\lfloor \log(n/m) \rfloor$ bits per element. The queries are select(V, i)=($select_1$(H,i)+1-i)·2^t + L[i] and rank(V,i) = linear search from 1+$rank_t$(H, $select_0$(H, i)). For example, we divide the bit-vectors by five bits, which are 6, 8, 9, and 17 (00000, 01011, 00000, 00100, 00000, 00000, 00). In this example, H=01011001 and L=2011. We can retrieve select(3) = 9.

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. While arrays are processed, bitmasks are also transformed, which is expensive to recreate auxiliary data structures. To minimize the cost, we employ the native function to count the ones in bitmasks without any auxiliary data structures. However, 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.



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

Figure 3.4: Generate a MaskRDD for each operator

Sequential Access

The operators (*e.g.*, Filter and Aggregator, described in Section 3.3), 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.

Random Access

Few, but most frequently used operators have a random access pattern, such as sub-array. 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 [82] 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

	CSR	Spangle
Space Complexity	O(2E + N)	O(E + N $\times$ N / W)
Time Complexity	O(lg k)	O(n / W)

Table 3.1: Comparison between CSR and Spangle

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.

3.2.4 Space and Time Complexity

Compressed sparse row (CSR) or compressed sparse column (CSC) format is widely used to store the sparse matrix. Table 3.1 shows the comparison of two.

In detail, for the space complexity, the CSR format is O(2E + N) where N $\times$ N matrix has the number of elements, E. The bitmask format in sparse mode is O(E + N $\times$ N/W) where W is a word (*e.g.*., 64-bits). However, we focus on large-scale matrices, and Spangle partitions matrices to process them in parallel. Section 3.5 shows that a chunk (*i.e.*, matrix) is effective around 128 - 256 (65,536), and we recommend the chunk size is multiple of 64. In this case, the bitmask format can be smaller than the CSR format, but if non-zero cells rarely exist, it wastes the space. Then, we suggested super-sparse mode. The lookup time using hierarchical bitmask can be slow because Spangle must access two bitmasks. However, depending on operation characteristics, the data size affects the performance rather than computational time because of network I/O. Thus, we optimize the performance by reducing the data size in memory (e.g., the bitmask-matrix for PageRank) and data shuffling, described in Section 3.4.

With space complexity, a cost function to select a suitable mode is as follows. We assume that a word is 64-bits. Theoretically, in terms of space complexity, the sparse mode is desirable if

$$the\ number\ of\ valid\ cells + \lceil the\ number\ of\ cells/64 \rceil < the\ number\ of\ cells$$

which is related to the compression ratio. However, this is not a strict rule because the data skewness and the processing method for arrays affect the performance besides the data size. We do not describe the in this paper by page limitation, but in our experiment using uniform distribution datasets, the sparse mode is adequate in terms of both data size and the processing time when the density is under 0.1. Likewise, when the density is under 0.025, the super-sparse mode is effective.

In terms of time complexity for the data access (random access), CSR can be O(lg k) time for column (binary search), where k is the number of non-zero elements of the specified row. The time complexity of bitmask format is O(n/64), where n is the number of cells. To overcome this overhead, we optimize the performance by using bitwise operations. In Section 3.5, we evaluate the efficiency of bitmask-based calculation by an additional experimental result. We compared its performance with other formats (COO, CSC).

3.3 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 [83, 26, 28] and map algebra [84], which are adopted in array-based or geospatial systems [11, 15, 67]. 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 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.

3.3.1 Operators using Bitmasks

Subarray

Figure 3.4a 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.

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 bitvectors 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 3.4b.

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 suboperators: 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 3.4c, if two input arrays use the MaskRDD, it concatenates all ArrayRDDs and executes *AND* or *OR* between two MaskRDDs (*i.e.*, M_A and M_B).

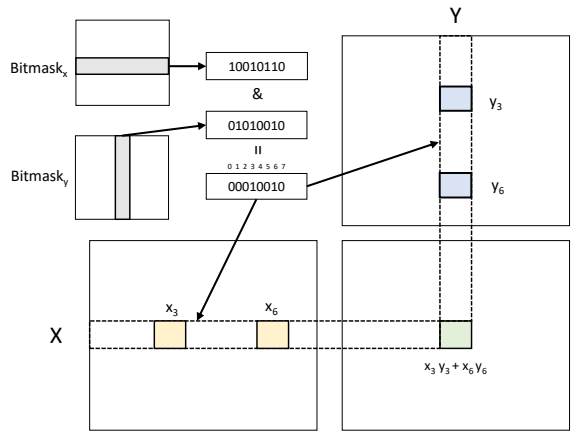


Figure 3.5: Matrix multiplication using bitmasks

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 3.4, and library operators (*i.e.*, Breeze). Figure 3.5 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.

3.3.2 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 generating 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 ap-

plication is insensitive to accuracy.

3.4 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.

3.4.1 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.

3.4.2 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 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 jth 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

$$a_{ij} = c_{ij} \cdot w_j \tag{3.1}$$

where $\mathbf{w}$ is an N$\times$1 vector. From PageRank and Equation (3.1), 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,

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

is equivalent to

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

where the symbol, $\circ$, is the element-wise product (Hadamard product). The transition matrix A can be decomposed into a matrix $\mathbf{A}'$ and vector $\mathbf{w}$. 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 3.5.

3.4.3 Stochastic Gradient Descent

The gradient descent algorithm is one of the most well-known and straightforward methods for solving convex optimization problems [85]. 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 $\mathbf{x}$ is updated for each step, t $\geq$ 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 [86, 87], and we adopt the idea of the parallel SGD algorithm [88]. 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.

Mapping Chunk IDs in Parallel

In raster data, Spangle first assigns chunk IDs to all cells, described in Section 3.1.3, 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 training samples are independent of each other. Figure 3.6 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:

$$C_n = nP \cdot rID + pID \tag{3.2}$$

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

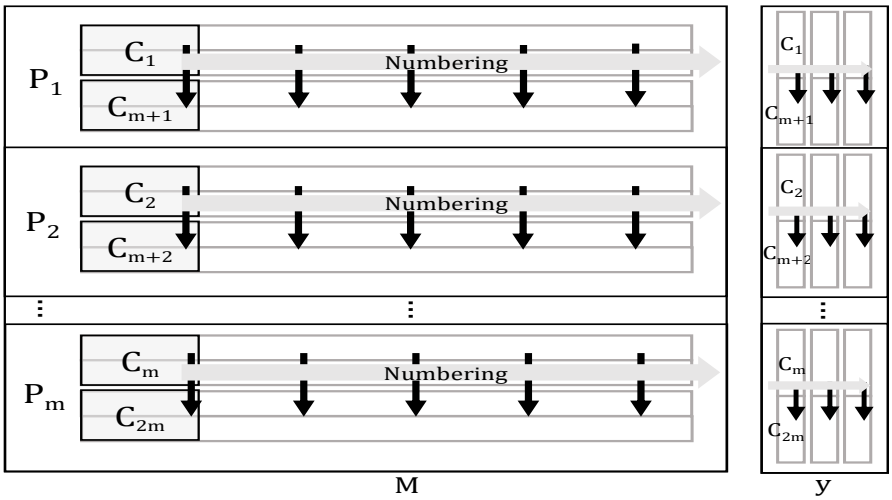


Figure 3.6: Numbering chunk IDs

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 (3.2) reversely. It searches chunks in parallel, which ensures the linearly scalable performance, according to the parallel SGD algorithm [88]. 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

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

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 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

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \theta((h(M_t \cdot \mathbf{x}_t) - \mathbf{y_t})^T M_t)^T \tag{3.3}$$

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).

3.5 Experiments

We evaluate the performance of Spangle using real datasets. Spangle was compared with three systems, which can process raster data: SciSpark [6], RasterFrames[1], and SciDB [12]. 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.

3.5.1 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 hyperthreading), 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

[1] https://rasterframes.io/

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 3.2: Queries for raster data processing

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.

3.5.2 Raster Data Processing

Datasets

We used two different raster datasets: *SDSS* from astronomy surveys [65] and SeaWiFS L3 Chlorophyll, *CHL* [68]. As the SDSS website offers the images encoded in the FITS file format [89], 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

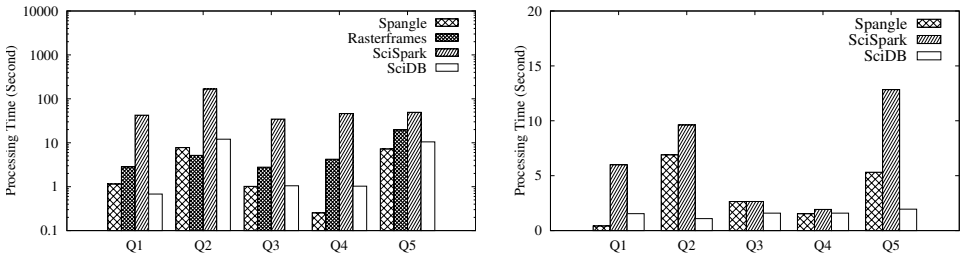
scanline has observed five filters (broad bands), called *u g r i z*, with each consisting of 2048 by 1489 pixels basic units. CHL has three dimensions (*longitude, latitude, and time*) and one attribute (*chlorophyll*). Each cell is an eight-day average of *chlorophyll* of the earth with a resolution of 9 km $\times$ 9 km.

Query

Table 3.2 includes benchmark queries for raster data processing. We refer to a scientific data processing benchmark [2], 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 SciSpark and RasterFrames do not execute all queries by using their provided APIs, we select five queries (*i.e.*, Q1, Q3, Q4, Q5, and Q6) and rewrite them to fit into the systems. The Q2 extracts clusters from one image using a different threshold. It requires a user-defined operator and is similar to the third and fifth queries, which need a regridding or windowing function, but RasterFrames does not provide either function. The Q7 finds those whose centers fall in a specified rectangle. It computes the average of specific dimensions, but SciSpark does not provide it. The Q8 finds centers of observations in a group, and the Q9 finds a sequence of polygons in the observation group. Both Q8 and Q9 need cross join for specific dimensions and the average of specific dimensions. Especially, the Q9 requires an operator to convert attributes to dimensions. SciSpark and RasterFrames do not support those operators.

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.



(a) Without range query

(b) Using 1,000 images

Figure 3.7: Comparing raster data processing systems

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 spreads 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 3.7a shows the results of queries without a range query using 100 images,

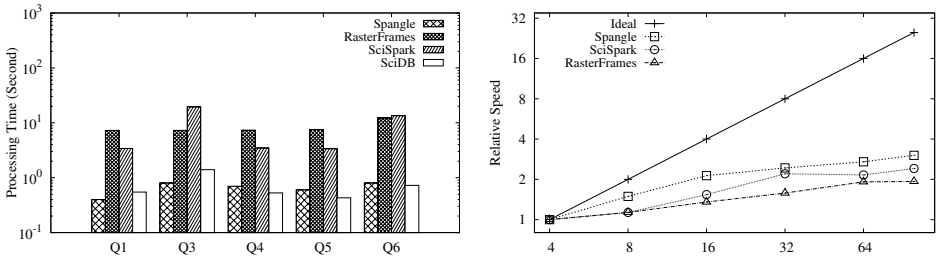
[2]https://www.unidata.ucar.edu/software/netcdf-java/

while Figure 3.7b includes a range query using 1,000 images. In this experiment, we set the chunk size to $128 \times 128 \times 1$. RasterFrames can support geometry operations with geometry libraries; however, it often yields incorrect results. Because we do not entirely trust the results of queries for that reason, we do not use only a range query to compare the four systems in Figure 3.7a.

Figure 3.7a 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 3.1.1, 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 3.7b, Spangle outperforms SciSpark. SciSpark manages sparse arrays as dense. It requires more memory and may incur overhead when shuffling data.

In addition to the experiment using ten nodes, we employ 100 nodes to investigate the performance and scalability. Each node has the 4 cores and 8GB RAM. In this experiment, we run an executor in each node. Figure 3.8a shows that the trend of the performance is similar in 10 nodes, but the relative performance gap between systems



(a) Evaluation in 100 nodes

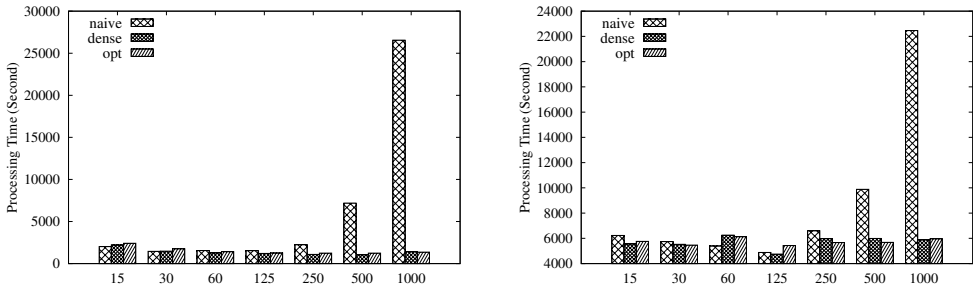
(b) Varying nodes

Figure 3.8: Comparing raster data processing systems in 100 nodes

is reduced. As we use more nodes, the parallelism and available resources increase, but the network overhead can incur more. Figure 3.8b shows the relative speed based on four nodes. In this experiment, we compare systems implemented based on Spark. When we denote T_n as the processing time using n-node, the relative speed of 32 nodes is T_4 / T_{32}. All systems cannot achieve ideal speedup. Compared with SciSpark and RasterFrames, Spangle is more scalable and achieves a speedup by 1.25x (SciSpark) and 1.15x (RasterFrames) at 100 nodes.

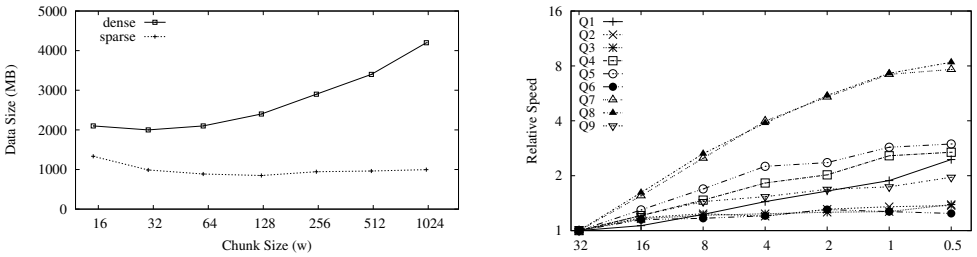
Next, we evaluate the data size and processing time with a sparse dataset between the dense and sparse modes described in Section 3.2.2. Since the chunk size has an impact on the performance, we varied it in this experiment. CHL stores the average value every eight days. We fixed one at the time dimension 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 3.9 shows that the processing time in dense and sparse 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 3.2.3.



(a) Filter (b) Aggregate

Figure 3.9: The processing time along with the chunk size



(a) Data size by modes (b) Varying the density

Figure 3.10: Data size and density along with density

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 3.10a 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

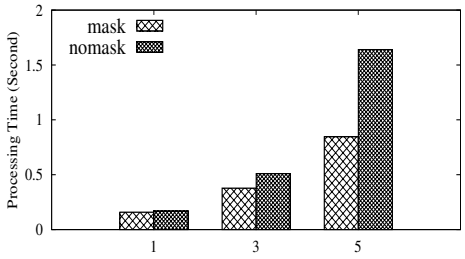


Figure 3.11: The performance using MaskRDD

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 3.10b shows the relative speed of each query in Spangle. Overall, the data density affects the query processing time, and when an array is getting sparse, the query processing time decrease. If an array gets sparse, the number of valid data decreases, and accordingly, the processing time is reduced.

Figure 3.11 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.

Dataset	Matrix Size	Density	Dataset	Matrix Size	Density
Covtype [90]	581K×54	0.218	VenturiLevel [91]	4M×4M	1.6e-19
Mouse [91]	45K×45K	0.014	Hugetric [91]	6M×6M	5.4e-7
Hardesty [91]	8M×8M	6.4e-7	Wb-edu [91]	10M×10M	5.8e-7
Mawi [91]	129M×129M	9.3e-9	Nlpkkt200 [91]	16M×16M	1.2e-6

(a) Matrix Datasets

Dataset	Edge	Vertex
Enron [92]	367K	36K
Epinions [92]	508K	75K
LiveJournal [92]	69M	4.9M
Twitter [93]	1,468M	61.6M

(b) Graph datasets[3]

Datasets	# of Rows		Feature
	Training	Test	
URL reputation [90]	1.9M	479K	3.2M
KDD Cup 2010 [94]	8.4M	510K	20M
KDD Cup 2012 [94]	120M	30M	55M

(c) Logistic regression datasets

Table 3.3: Machine learning datasets

3.5.3 Machine Learning

Datasets

Table 3.3 shows the datasets used in this experiment. In Table 3.3a, we used relatively sparse matrices for ML core operations and generated vectors that consist of random values for matrix-vector multiplications. In Table 3.3c, 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 ma-

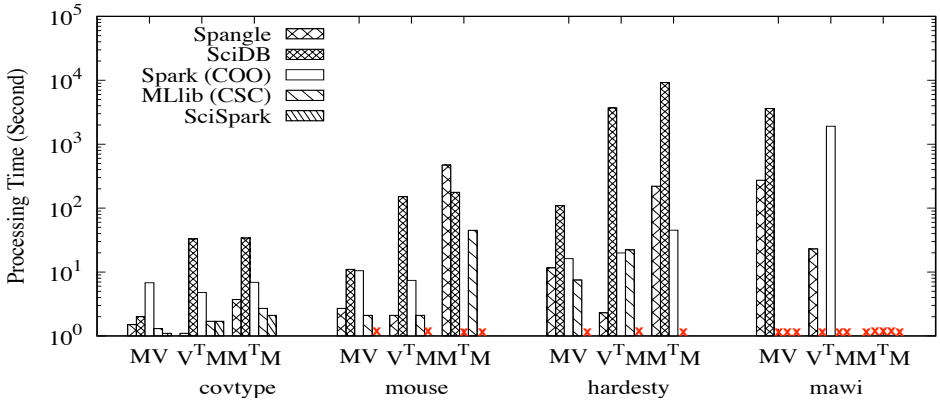


Figure 3.12: Machine learning core operations

chine learning core operations, and we summarize them as follows: matrix-vector multiplications (M$\times$V and $V^T\times$M) and transpose-self matrix multiplication ($M^T\times$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$\times$V or $V^T\times$M. The algorithms are often expressed as $M^T\times$M (*e.g.*, principal component analysis).

Figure 3.12 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 $V^T\times$M, compared with M$\times$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 3.4. 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^TM 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

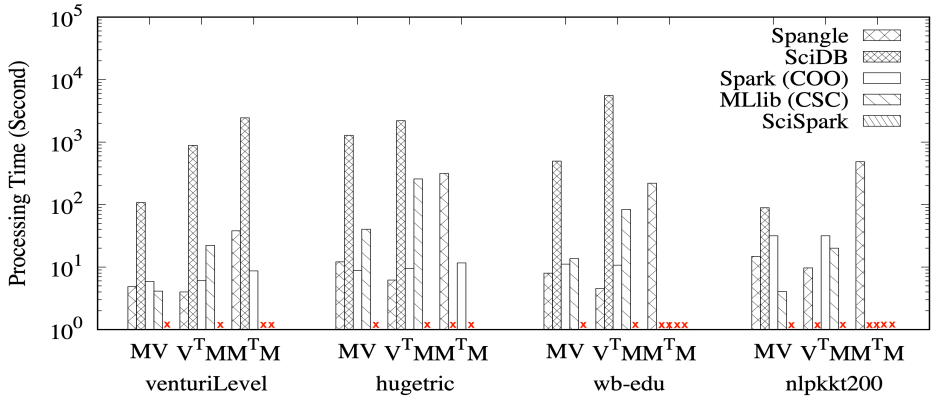


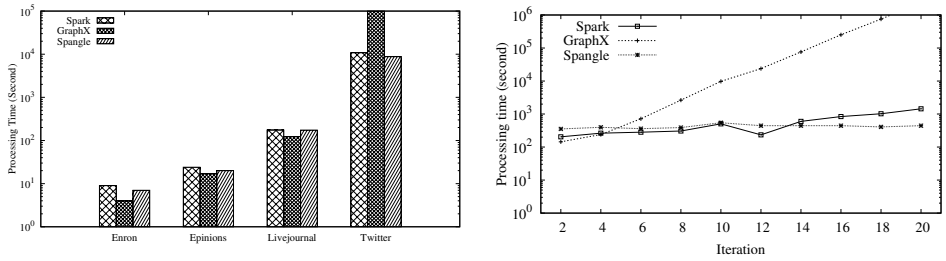
Figure 3.13: Matrix operations with increasing the matrix size

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 Figure 3.13, we evaluate the performance of matrix operation with increasing the matrix size. Compared with other systems and formats, Spangle only supports M^TM in wb-edu (10M $\times$ 10M) and nlpkkt200 (16M $\times$ 16M). Other systems and formats are suffered from the out of memory, and SciDB does not finish in bounded time, similar to Figure 3.12.

In summary, the major factor of the performance in matrix operations is not only computational algorithms but also the matrix size with density. 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 [95]. Figure 3.14 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



(a) Three Systems (b) Twitter

Figure 3.14: The processing time in PageRank

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.

GraphX has two RDDs for graphs, VertexRDD and EdgeRDD. While GraphX proceeds graph computations, temporal results should send to VertexRDD, which occurs data shuffling. To reduce data size for shuffling, a new RDD is created. EdgeRDD is not used, and Spark spills cached data (i.e., EdgeRDD) by LRU because memory is not enough to cache EdgeRDD. For the next step, EdgeRDD should be in memory, which leads to re-generating overhead by lineage. That is, VertexRDD and EdgeRDD are re-generated over time because one of them is spilled. Therefore, the processing

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	-	-	-	-

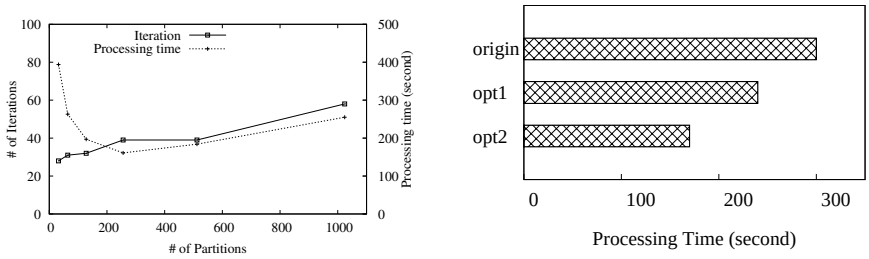
Table 3.4: The performance comparison using three datasets

time increases in each iteration by this re-generating overhead.

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, 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 3.4 shows the performance of Spangle and MLlib. We expected that two systems could 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 3.15a 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 low parallelism. Meanwhile, a large number of partitions incur network overhead for the reduce operation. Figure 3.15b shows the performance of



(a) Iterations and processing time

(b) Optimization for SGD

Figure 3.15: The processing time of logistic regression

Spangle using our optimization technique.

We investigate the two-step optimization introduced in Section 3.4.3. The `opt1` is a customized equation (3.3). The `opt2` is not to execute the transpose operation for a vector but to replace the description, instead. Specifically, the opt1 optimization reduces the computational cost. This decreases the processing time by 20%. Instead of transposing a training set, the transposing vector, relatively small and transformed for each iteration, has less overhead. The opt2 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 about 43%.

3.6 Summary

In this chapter, 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 processing can accelerate the creation of a more accurate machine learning model.

Chapter 4

Advanced Database Techniques for Large-Scale Arrays

Recently, there has been an increasing interest in analyzing scientific data generated by observations and scientific experiments, such as satellite images. Scientists often need a result from highly accurate and sophisticated analysis using diverse types and a high volume of data. These data are generated by high precision and resolution sensors, represented as multi-dimensional arrays (*i.e.*, raster data). It is required to manage scientific data by the appropriate model. At first, the relational data model was taken into account for such needs, but it was soon verified that the array data model is appropriate to manage scientific data [2].

Over the past few decades, much research has been done to develop query languages [30, 96, 97] and database management systems [11, 12] specialized in array manipulation and analysis. One of the popular array database management systems is SciDB [12], which stores data as multi-dimensional arrays and shows outstanding performance on scientific workloads and complex analytics that use algebra computations. In order to manage an increasing volume and various types of scientific data, we need advanced techniques. We describe the motivations for our work below.

Selective Scan for Filter Operator. When SciDB processes a query with *where* predicates, it uses *filter* operator internally to produce a result array that matches the predicates. Most queries for scientific data analysis utilize spatial information. However, the filter operator of SciDB reads all data without considering features of array-based DBMSs and spatial information. It unconditionally runs a full scan regardless of most range queries written in AQL with *where* predicates, which is clearly inefficient.

Scalable Loader. Many scientific data are commonly stored in dedicated structured file formats such as Hierarchical Data Format version 5 (HDF5) [7] and Network Common Data Form (NetCDF) [8]. Since those files contain metadata that includes dimensions and attributes, it helps scientists interpret logical structures of data in different platforms, standing for portability. In addition, most of the scientific data have dimensions or values represented by coordinates. The files consist of arrays where coordinates close to each other can be physically represented to be close, which ensures the locality of coordinate systems.

Scientific Database in Federation. If databases are not able to be interconnected with each other, users themselves have to establish the connections and integrate data in their application manually. In addition, to collect data in a single location, users should carry out heavy work, which includes downloading and loading data. The work is inefficient and inconvenient since it takes a significantly long time to load raw data with those many cumbersome steps. Furthermore, users need a federated schema before federating databases [98, 99]. A federated schema is an integration of multiple schemas. Unless schemas from multiple databases can be converted into a federated schema, transforming data format or re-defining a federated schema is required whenever data need to be integrated. Thus, it is not easy to find a simple solution offering a remedy for a complex query or analysis using distributed data in multiple databases. To manage an increasing volume of data in various types effectively, data are often organized in multiple databases, which can have different ownership and properties. Data can be partitioned and distributed geographically in multiple databases across

separate servers, considering their features and purposes of organizations.

In this chapter, we describe three techniques for query processing, data loading, and federated databases to enhance scientific analysis. The main contributions are as follows.

- We provide the new implementation of the *Filter* operator in order for SciDB to accelerate the processing of selective queries. The new implementation avoids full scans by leveraging range predicates and array indexes.
- We propose a new scalable approach, *Scalable Loader for SciDB* (*SLS*), which accelerates data loading on SciDB. It streamlines the file format conversion process and minimizes the loading cost by removing sort and data exchange steps.
- We implement *SDF* that facilitates the sharing and exchange of scientific data and generates federated schemas in the query execution time to avoid overhead due to different schemas. It is fully operational with fast connections between separate SciDB clusters by two connection models.

The rest of this chapter is organized as follows. Section 4.1 describe the selective scan for the filter operator in SciDB. In Section 4.2, we describe the design and implementation of our new loading method. In Section 4.3, we present the system architecture and data flow of *SDF*. Subsequently, we explain implementation details and system considerations. We evaluate our work using real datasets and compare the performance in Section 4.4. Then, we summarize our work in Section 4.5.

4.1 Selective Scan for Filter Operator

SciDB provides arithmetic and logical operations via a language called Array Functional Language (AFL) and an SQL-like language called Array Query Language (AQL). When executing an AQL query, it is translated into an AFL query in the preparing phase. While processing queries, chunks by which SciDB stores and manages each

Operator	Between	**Filter**	Lookup	Slice	Subarray	Thin	Scan
Full Scan Performed	No	Yes	Yes	No	No	Yes	Yes

Table 4.1: Selection and scan operators

array are read. Chunk metadata, such as coordinates that represent chunk boundaries, is saved in a chunk map table.

Table 4.1 compares six selection operators and a scan operator with respect to the types of data access performed by SciDB. The three operators, $Between$, $Slice$, and $Subarray$, take advantage of the coordinates of data chunks to read them selectively, but the others (*e.g.*, *Filter* operator) perform the full scan. Note that SciDB transforms most AQL range queries with where clause into the filter operator. Such features are commonly found in other array DBMSs as well. RasDaMan [11, 100], for example, has the unit of storage and access called $tile$. RasDaMan Query Language (RasQL) offers $trimming$ to reduce the spatial domain and $section$ to slice the dimensionality from an array.

As described above, AQL queries are translated into AFL queries. If an AQL query contains a *where* clause, it is internally converted to a **filter** operator which filters out data based on *where* predicates. Precisely, the filter operator examines the attributes and the coordinates of data. Then it produces an array with the same schema of the input array that consists of cells that satisfy *where* predicates. Even for range queries which can utilize indexes, filter operator naively executes full scans and checks whether each cell is in the range or not, yielding poor performance regardless of the range. In our approach, we utilize metadata which is array information such as starting and ending coordinates, chunk length of each dimension, etc. By exploiting the regularity of coordinates in arrays, each instance can easily identify which chunks are relevant to an input range query. Thus, it avoids heavy full scans.

Figure 4.1 shows an example of windowing and how selective scans are processed. There are four instances and nine chunks. Q represents a range query, and C_2, C_3, C_5

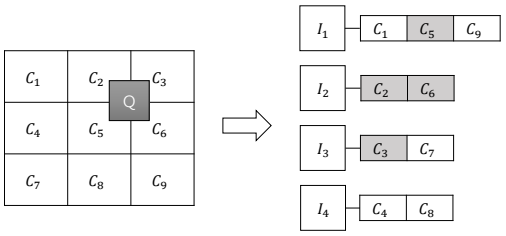


Figure 4.1: An example of windowing (I_n: n-th instance, C_n: n-th chunk)

Algorithm 2: The pseudo code of windowing

```
Data: chunks, boundary
Result: position
initialization;
if chunk is not null then
    current_position = chunk.getPosition();
    if the chunk is not in a boundary then
        chunk++;
    position = current_position;
```

and C_6 are relevant chunks of Q. In the case of the current filter operator, all instances fully scan the entire data (i.e., nine chunks) and examine the data read using query predicates. This step contains redundant processes because the chunks, including data requested by the query Q are relevant only to four chunks. However, using our approach, we can reduce the time to search data as well as irrelevant I/Os. For example, the 4th instance does not need to read all chunks. SciDB only reads chunks (C_2, C_3, C_5, and C_6) related to the query.

The pseudo code of windowing is in Algorithm 2. If boundaries of a query are given (i.e., predicates contain boundary conditions), we bypass unrelated chunks of a query and empty chunks.

4.2 Scalable Loader

Data loading is a beginning step in the analytics process. SciDB loads data in a one-dimensional array format, even though scientific data have a few dimensions, inadequate to compute linear operations and utilize the array model. In order to reshape one-dimensional arrays to multi-dimensional arrays, data conversion is required. The $redimension()$ transforms arrays to an appropriate dimensionality. It is analogous to both indexing and clustering, making data objects close to each other stored together. Hence, as it is a core operator to ensure data locality, we call data loading the entire process, containing the loading and redimensioning.

However, the data loading takes a significantly long time and causes extra overhead. For example, scientists may need to quickly forecast the weather by analyzing weather data over the past few days, but data loading takes many hours, even a few days. Compared with the query processing time, the time is humongous, only taking a few seconds or minutes. Specifically, one of our experiments shows that the entire process takes about 151,300 seconds (about 42 hours) on MODIS RRS data. In addition, when a raw file format is not loaded into SciDB, the situation is even worse since it incurs the additional cost to convert the file format.

In this section, we illustrate how the loading and redimensioning processes are improved. Our new approach to data loading is called *Scalable Loader for SciDB* (SLS), scalable by eliminating all-to-all exchange in the redimensioning stage. The whole process of *SLS* is in Figure 4.2. We assume that the relevant schemas of one-dimensional and n-dimensional arrays are already defined. That is, the metadata for a one-dimensional or n-dimensional array is available, and we use this information to optimize the loading process. We distribute data by Message Passing Interface (MPI) in the loading stage and use a hash-aware approach. Since SciDB loader does not regard how to partition data in redimensioning stage, it causes communication overhead. The hash function is identical to the redimensioning stage. This approach minimizes the overhead by avoiding all-to-all exchange, which leads to scalable data loading. In

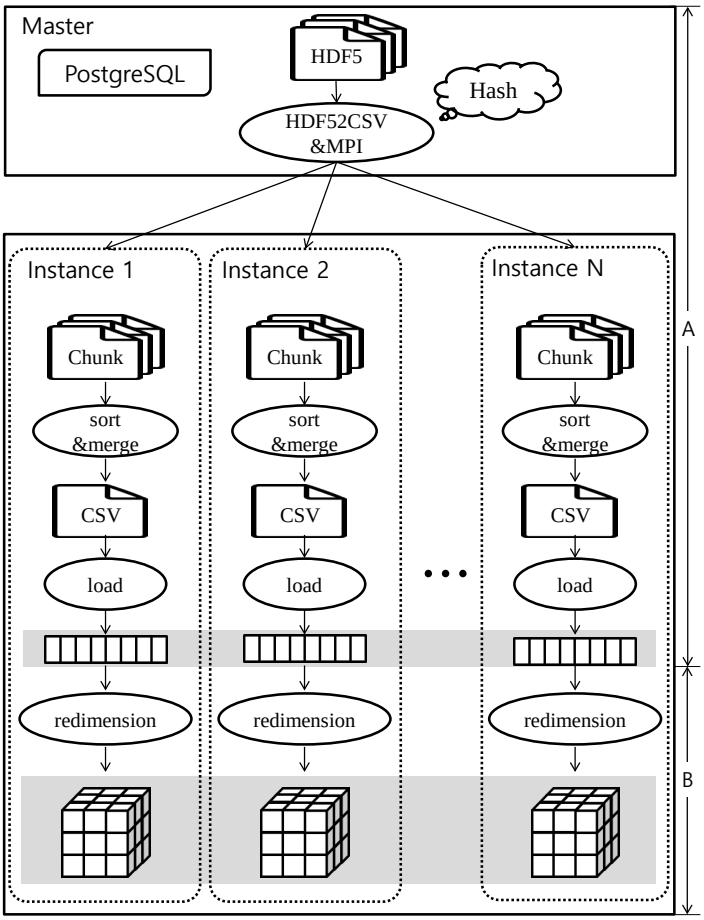


Figure 4.2: Data loading by *SLS*: (A) Loading stage, (B) Redimensioning stage

addition, we can reduce the loading time substantially by simplifying the conversion process and modifying the distribution method. Especially, redimensioning, which occupies a significant portion of time in data loading, is optimized by removing sort and redistribution. We implement a new redimension operator using a plug-in method, user-defined operators.

Figure 4.3 shows overall steps, and we describe them in detail.

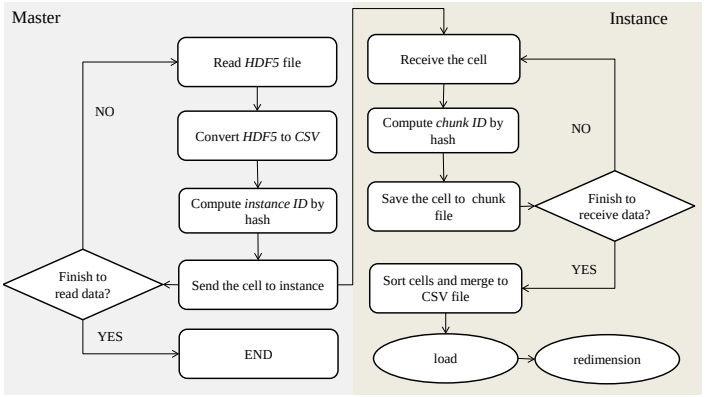


Figure 4.3: The diagram of data loading process

4.2.1 Loading

Initially, because the raw data (*i.e.*, HDF5) resides only in the master node, they should be split and sent to corresponding instances for parallel loading. Before sending the data, SLS reads HDF5 files with the metadata (i.e., properties of data) and converts files into CSV format. Unlike vanilla SciDB, the conversion is pipelined. That is, SLS transmits the CSV files to instances on the fly. It executes the process being cell-to-slave mapping and calculates a hash value corresponding to each cell. The hash value comes from a chunk ID and the number of instances determining the target instance the data should reside. This computation is the same as the data distribution method (*i.e.*, hash) of SciDB.

After hashing, the master node can send data to the corresponding instances, converting from HDF5 to CSV file format on the fly in an MPI packet. The MPI is a de facto standard interface to assist in communicating among processes or machines. Consequently, we can minimize I/O overhead and reduce the loading time. SciDB instances receive those data (cells) and merge them into local CSV chunk files. In this process, the cell-to-chunk mapping is executed following the array schema. This mapping determines which cell is stored in a corresponding chunk file. The cells are collected to each chunk file with a unique chunk ID, indicating the chunk sequence

order.

SciDB explicitly examines whether all cells are sorted by the specified order during redimensioning. To avoid sorting, which consumes the significant time, we execute the inter-bucket sorting ahead of redimensioning. The inter-bucket sorting algorithm is in Algorithm 3. To compute a unique chunk number, we first compute the bucket interval. Since the major order is from the last dimension, the chunk number increases from the last dimension to the first dimension. Afterward, SLS allocates each row to a file by computing the chunk file number. As this computation is performed with receiving data concurrently, SLS can minimize the overhead.

Algorithm 3: Inter-bucket sorting

Input : the number of dimension dim, coordinates $coord$, starting coordinates s_coord, ending coordinates e_coord, chunk interval ci, bucket interval bi, received data $record$

Output: chunk_files cf

$bi[dim] \leftarrow 1$;

for $i \leftarrow dim - 1$ **to** *0* **do**

    bi[i] $\leftarrow bi[i+1] \times ((e\_coord[i + 1]$ - $s\_coord[i + 1] + 1)$ / $ci[i+1]) + 1)$;

while $record\ != null$ **do**

    $k \leftarrow 0$;

    $bn \leftarrow 1$;

    for $i \leftarrow 1$ **to** dim **do**

        $bn \leftarrow bn$ + $(coord[i]$ - $s\_coord[i])$ / $ci[i] \times bi[i]$;

    $cf[bn] \leftarrow record$;

The chunk files are read by increasing order with chunk ID and merged simultaneously. While merging them, SLS sorts cells in the chunk file internally using intra-bucket sorting, described in Algorithm 4. This sorting re-organizes each chunk file.

By two sorting algorithms, we reduce the data loading time and bypass the sorting

Algorithm 4: Intra-bucket sorting

Input : the number of dimension dim, coordinates $coord$, starting coordinates of a chunk s_chunk, chunk_interval ci, index idx, data $record$

Output: a sorted file sf

$idx \leftarrow 0$;

$interval \leftarrow 1$;

for $i \leftarrow dims$ **to** 0 **do**

    $idx \leftarrow idx + interval \times coord[i] - s\_chunk[i]$;

    $interval \leftarrow interval \times ci[i]$;

$sf[idx] \leftarrow record$;

in the redimensioning stage, leading to improved data loading performance. After all, SLS loads the sorted files to a one-dimensional array.

4.2.2 Redimensioning

The steps in the redimensioning stage are the same as vanilla SciDB except for sort and redistribution (all-to-all exchange) steps, described in Section 2.3. Before SLS loads data into SciDB, they are already distributed by hash and have sorted order. There are no further sort and redistribution in the redimensioning stage. In addition, as the redistribution is removed, the remaining step is to store a target array locally and synchronize instances with each other. It makes the third step simpler because the preparation for aggregating data is unnecessary.

In summary, the first step remains for converting an n-dimensional array to a one-dimensional array with mapping attributes to corresponding coordinates. The third and fourth steps still exist without preparation and redistribution of the data. SLS loads data in parallel, which removes a barrier to scalability. It can minimize computation and network overhead in data exchanges between instances.

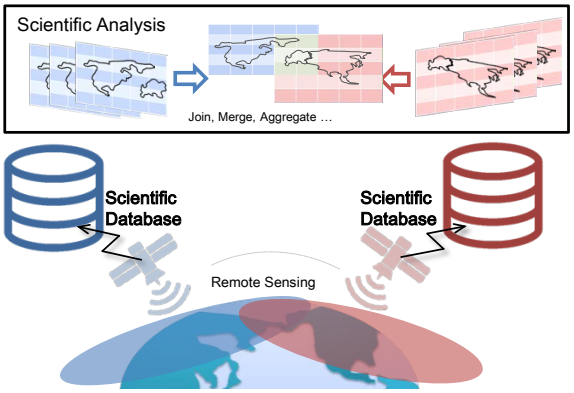


Figure 4.4: An example of database federation

4.3 Federated Database System for Scientific Data

Scientific data are often organized in multiple databases which can have different ownership and properties. Data can be partitioned and distributed geographically in multiple databases across separate servers, considering their features and purposes of organizations. For instance, multiple sensors send their events or changes to their servers, as Figure 4.4. These generated data may be separately stored and managed in different databases. When scientists need to analyze such distributed data together, they should be able to access multiple different databases simultaneously. Especially, if there is an overlapping area observed by satellites with different spectral bands, analyzing these data together can give more accurate and comprehensive understanding.

It is common practice that data are classified by their features or different purposes. In addition, they are stored in separate databases by different sources and organizations in order to manage data effectively (*e.g.*, remote sensing with various bands over the same range). However, to enable large-scale data analytics, queries often need to access both local and remote databases. Basically, as data sharing or exchange with disparate databases is infeasible, A DBMS supports integrating operations with minimizing the overhead to access a remote database.

We adopt the principle of a federated database system to provide a view as a single

database from multiple databases. We add a novel feature to SciDB, called *SDF*, *Scientific Database in Federation*. By adopting the principle of a federated database system, *SDF* abstracts away details about the process of combining and integrating databases, which considerably appeals to scientists who need to pose sophisticated analytics but do not have expertise in database systems.

SDF accepts the primary ownership of systems (*i.e.*, autonomy) and gives a federated view as a single database from multiple databases. It can federate databases in the same cluster or in physically separate clusters. Moreover, *SDF* preserves unique system features including numerous analytics libraries while accessing remote databases. The main advantage is the ability to combine data from multiple databases in a single query statement and join separate databases together for complex analysis.

SDF is currently implemented in *SciDB* [12] by user-defined operators (UDO). UDO is a plug-in provided by SciDB and helps *SDF* federate multiple databases with relatively simple installation. It is aimed at using SciDB query languages (both AQL and AFL) to allow *SDF* to access remote databases at the query level. Unlike BigDAWG [60], *SDF* is not a middleware solution to federate databases.

SDF has two benefits. First, it is convenient to access distributed data simultaneously. Users can federate databases by a single query. For example, suppose in the example given below, `cloud` is a local array and `seawifs` is in a `remote` database, the following query (the running example in this paper)

join(cloud, connector('remote', 'seawifs'))

is able to be processed. The sub-query, *connector*, enables a local SciDB to access remote databases, described in Section 4.3.1. Also, to establish a connection, *SDF* uses a synonym table. Once the name of a remote database is registered (*remote* in the running example), its information is provided for easy use.

Second, scientific data analysis may often need a specific range of an array or an aggregated result rather than an entire array. As a method that *SDF* employs to federate databases is to pose a remote query to a remote cluster, a remote cluster can send back

a *foreign array*. The foreign array is referred to as a result array of a remote query processed by a remote SciDB. It is useful when the target array size is huge, but the foreign array size is small. In addition to those two benefits, *SDF* can give a scalable database view since the abstraction of federated query processing makes it unnecessary to modify a system engine every time databases are newly federated.

4.3.1 Operator Syntax

In order to federate databases, *SDF* provides four operators. We define operators of *SDF* in AFL, and the syntaxes are as follows:

createlink_statement ::= CreateLink(DB name, IP address, port number);
showlink_statement ::= ShowLink();
deletelink_statement ::= DeleteLink(DB name);
connector_statement ::= Connector(DB name, array | query);
As is described in Section 2.3, AQL queries are compiled into AFL, and these operators can be used in an AQL `FROM` clause as well as in a nested query. **CreateLink** saves information of remote databases using *PostgreSQL*. An IP address is used to connect a remote SciDB, and a port number is used to specify a target database. **ShowLink** lists the information of remote databases, and **Deletelink** removes information of a remote database.

Connector is a key operator of *SDF*. It may be involved in a federated query to send a remote query and access to a remote SciDB. The connector operator accepts two parameters, a remote database name and either an array name or a query named *remote query*. If the second parameter is an array name, it retrieves an array in a remote database. Otherwise, a local SciDB poses a query in a remote SciDB.

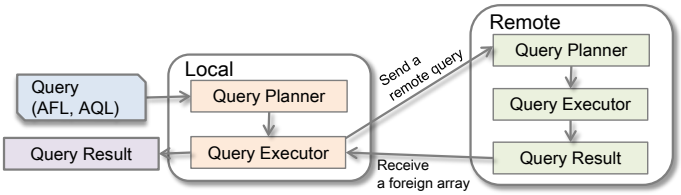


Figure 4.5: Processing sequence to process a federated query.

4.3.2 Federated Query Processing

SDF federates multiple databases by decomposing a *federated query* into sub-queries and posing a remote query to a remote SciDB. The federated query is referred to as a query which accesses separate databases. This federated query follows the regular query processing. When posing a query to a local SciDB, users utilize array information such as array and attribute names. Similarly, to pose a federated query, remote array information with a name which indicates a target remote SciDB is used. In addition, users need not create a federated schema in advance for federated query processing because *SDF* generates a federated schema by receiving schemas from remote databases in the query execution time.

There are a few steps for message exchange to process a federated query. To access a remote database, a local SciDB must create a link to a remote SciDB. As an IP address is required to establish a connection to a remote SciDB, and databases are distinguished by port numbers, any user who attempts to access other databases should obtain them in advance. *SDF* provides an alias (*i.e.*, a database name) to retrieve the information from PostgreSQL where SciDB stores its metadata.

When the connection is established, a local SciDB can access a remote database. Since the local SciDB does not have a federated schema, it requests a foreign array schema. Based on this, a federated schema is generated in the query execution time. Then, the local SciDB submits a remote query to a remote SciDB as Figure 4.5. After receiving a foreign array entirely, the local SciDB sends a message to inform that the whole process has been completely executed. In the end, the connection between a

local SciDB and remote SciDB is closed.

4.3.3 System Architecture

The query processor of SciDB decomposes a federated query into sub-queries. In our running example, the query is decomposed into *join* and *connector*. The connector operator, described in Section 4.3.1, sends a query to a remote SciDB. This connector operator uses *SDF* modules. The architecture of *SDF* is in Figure 4.6, and we describe its modules as follows.

Modules

SDF has seven modules to process a federated query, implemented by user-defined operators (UDO). As is described in Section 2.3, it is useful to add new algorithms or operations to SciDB using UDO. Compared with developing a new feature in a database engine, installation of UDO does not require re-building SciDB.

Address Manager manages information of remote databases such as IP addresses and port numbers. The information is stored in PostgreSQL because it is more suitable for frequent inserts and updates than SciDB. A SciDB cluster consists of several SciDB instances, but only the master instance has this module.

Connector establishes a connection between local and remote SciDB clusters. The connection should be maintained until the transmission of a foreign array is complete.

Executor sends a query to a remote SciDB via *Connector*. If an array does not exist or the syntax of a remote query is incorrect, the remote SciDB returns error messages. Otherwise, the query is processed, and the remote SciDB returns a schema and a foreign array. While *Executor* submits a remote query, additional parameters informing a query type (requesting either a schema or an array) piggyback on the query. If an array is requested, the query brings parameters such as an instance ID and the number of instances.

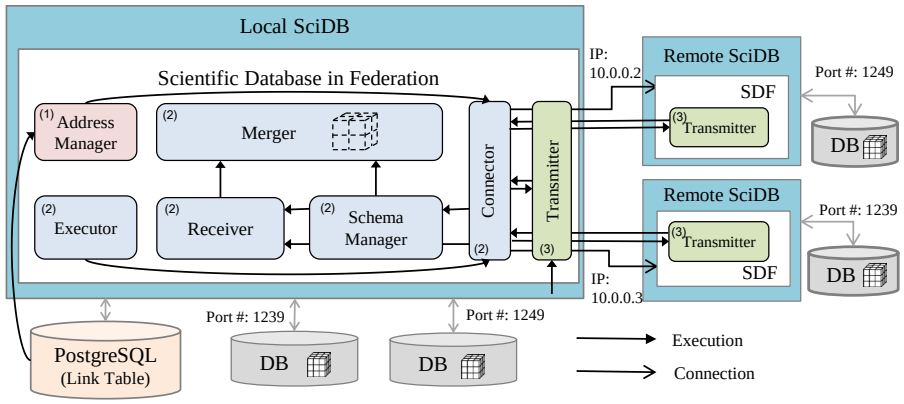


Figure 4.6: *SDF* architecture: local and remote SciDB clusters are connected for federated query processing. A SciDB can be connected using an IP address, and a database is distinguished by its unique port number. Only a master node has a (1) module. The (2) modules are in a master or all SciDB instances depending on the connection model. The (3) module should be in a remote SciDB. A local SciDB also should have it, if local databases need to be federated.

Schema Manager receives the schema of a foreign array from a remote SciDB, which makes it unnecessary to define a federated schema in advance. Especially, the schema of an original array can be transformed if a query (*e.g.*, aggregate) creates a new array different from an original array. The schema is used to create a virtual array in a local SciDB.

Receiver receives a foreign array from a remote database. The transmission unit can be either a cell or a chunk, described in Section 4.3.5.

Merger produces a virtual array using a schema from *Schema Manager* and an array from *Receiver*. While receiving a foreign array, *Receiver* directly passes received chunks to *Merger* to create a virtual array in a local database.

Transmitter sends a schema and a foreign array to a local SciDB. A local SciDB can also use *Transmitter* when sharing arrays between local databases. In a cluster-to-master model described in Section 4.3.4, *Transmitter* filters out cells and sends sub-

arrays of a foreign array to instances, using instance IDs. It should obtain the schema of a foreign array ahead of the filtering.

4.3.4 Connection Model

Since SciDB runs in a cluster environment, establishing a connection between clusters should be provided. *SDF* provides two connection models, *master-to-master* and *cluster-to-master*, for shared-nothing architecture. A cluster-to-master model has a more complicated process but better performance than a master-to-master model.

Master-to-Master

Separate clusters can have inequivalent slaves. The number of slaves leads to different data partitioning. To federate databases, a local cluster should be aware of where remote data have resided. By providing two connection models, *master-to-master* and *cluster-to-master*, *SDF* can federate multiple databases regardless of the number of slaves. Especially, in the cluster-to-master model, a query is converted to multiple localized sub-queries. This conversion leverages the query performance of aggregation queries. In addition, we designed *SDF* to federate databases with several factors such as query processing, the transmission unit, and query performance.

A master-to-master model is a simple model. A master of a local SciDB connects with a master of a remote SciDB. As all modules are in a local master, a local master sends a remote query to a remote master and receives a foreign array alone. The master also maps a foreign array to a virtual array and distributes the corresponding instance ID. A federated query can be processed without setting the equivalent number of instances of federated SciDB clusters because the connection is established between two masters.

Algorithm 5 shows how *Receiver* works. It uses iterators to receive chunks and maps a foreign array into a virtual array by iteration. Unless an array fits in memory, it is temporarily written on disk to avoid OS swapping. The writes may cause the degrad-

Algorithm 5: Receiver

```
Data: An foreign array, A virtual array
Result: A virtual array
initialize(virtual_array, schema);
foreign_iterator = foreign_array.getiterator();
virtual_iterator = virtual_array.getiterator();
while foreign_Iterator != EOF do
    Merger(virtual_iterator, foreign_iterator);
    virtual_iterator++;
    foreign_iterator++;
return virtual_array;
```

ing performance of query processing. However, although this I/O can be overhead, it is almost overlapped with the receiving process. During the process, *SDF* calls *Merger* to merge received chunks in a pipeline.

The local master distributes the virtual array to all local SciDB instances using hash. The equations (A) and (B) in Section 2.3 are used to distribute the array. According to the equations, the hash function requires an array schema and the number of instances. Since a local master has metadata and an array schema, it can assign a chunk to a SciDB instance using a hash value. However, this model causes extra overhead compared with a cluster-to-master model since foreign arrays are indirectly sent to all instances.

Cluster-to-Master

In contrast to the master-to-master model, a cluster-to-master model can directly send sub-arrays of a foreign array to local instances as Figure 4.7. This model also does not require the identical number of instances between local and remote SciDB clusters. In order to process a federated query, *Address Manager* retrieves the information to

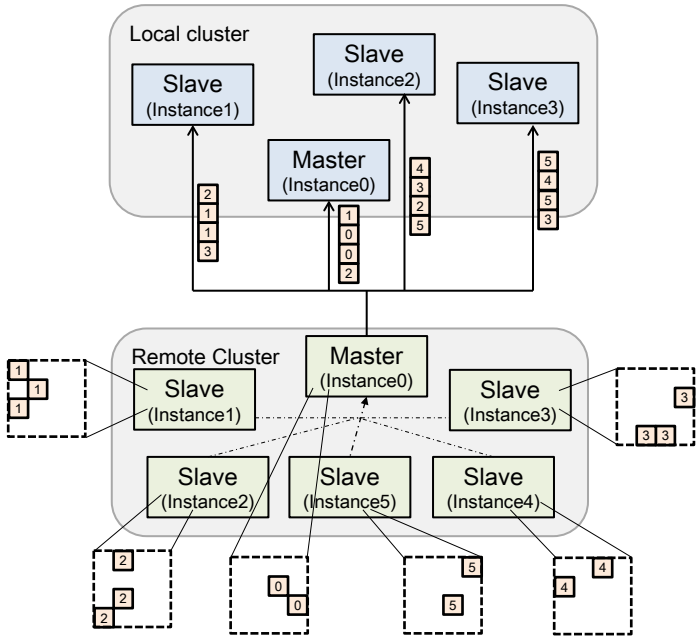


Figure 4.7: Data flow of the cluster-to-master model. This model does not require the identical number of instances between SciDB clusters. A remote master directly distributes a foreign array to local instances.

connect a remote SciDB and its database, the same as the process of master-to-master model. Afterward, it broadcasts the information to local SciDB instances. When they receive it, *Connector* connects all instances to a remote master. The instances transform the query into localized sub-queries and pose them to a remote master. Internally, the number of instances and each instance ID piggyback on a localized query.

In a remote SciDB, while queries are processed, *Transmitter* prepares sub-arrays to send them to corresponding instances. It utilizes a foreign array schema to calculate the hash value which determines the target instance of each chunk. The role of this module is more critical when an aggregate query is processed. With the query, the schema may be re-created, which leads to changing not only hash values but also a subset of cells. This subset is used to create a sub-array for a corresponding instance. If the value is computed from an initial schema, the result must be incorrect.

Algorithm 6: Transmitter

```
Data: A foreign array, the_number_of_instances, instance_ID
Result: A chunk
while foreign_array.iterator != EOF do
    chunkID= hash(foreign_array.chunk);
    I_ID = chunkID % the_number_of_instances;
    if I_ID == instance_ID then
        return foreign_array.chunk;
    foreign_array.iterator++;
```

For example, a cell with the coordinates {1, 1, 10} is converted to {1, 1} by a group-by clause of an aggregate query. Assuming that the first ID derived from hash value using {1, 1, 10} is `four`, it is used to create a sub-array for the instance whose ID is `four`. However, if {1, 1} is used, the ID is `one`, and the cell must place on the sub-array for the local SciDB instance, whose ID is `one`. Thus, *Transmitter* sifts cells to send data to target instances correctly.

Algorithm 6 shows the filtering process of chunks for local SciDB instances. *Transmitter* computes a chunk ID and I_ID using hash and modular calculation, the same as the equations (A) and (B) described in Section 2.3. If I_ID is equal to the instance ID, *Transmitter* sends the chunk to a target instance, bypassing irrelevant chunks. The receiving process is the same process in the master-to-master model, but it is operated concurrently in every instance.

4.3.5 System Considerations

While designing *SDF*, we consider several system factors for federated query processing. In this section, we describe the query processing, the transmission unit, and query performance including the query plan and the relationship between disk I/O and network bandwidth.

Query Processing

There are two considerations for query processing. First, it may be considered overhead if a local SciDB receives a foreign array to create a virtual array in order to treat it as a local array. However, this work caches the array which is reused soon. It minimizes the overhead as well as enables an initial query processing mechanism in a local SciDB after a remote array is sent to it.

Second, instead of accessing remote databases directly, a local SciDB receives the foreign array from a remote SciDB. This design has two reasons. Suppose a local SciDB accesses a remote database bypassing remote SciDB instances, especially the master. In that case, it may have the authority to control a remote database, which can disturb query processing and even scheduling in a remote SciDB. That is, the autonomy between the two systems is not compromised. Moreover, the local instances should maintain the metadata of remote databases (*e.g.*, the location where a corresponding array exists). It is not transparent to integrate databases, and complicated operations are needed such as searching a SciDB instance that has a specific part of an array and ordering data sequence, which may incur the additional cost.

Transmission Unit

The unit for data transmission affects the query performance. There exist two units to transmit arrays: *cell* or *chunk*. A chunk is a group of cells, described in Section 2.3. If a local SciDB receives a foreign array by a chunk instead of a cell, it yields better performance. While receiving a foreign array, a local SciDB executes copy operations to create a virtual array in memory. If the unit is a cell, the operation should be performed with cell by cell, which causes extraneous overhead compared with chunk by chunk.

Query Performance

It is possible to cause overhead depending on query plans and the relationship between network bandwidth and I/O throughput. First, to optimize the query performance, fed-

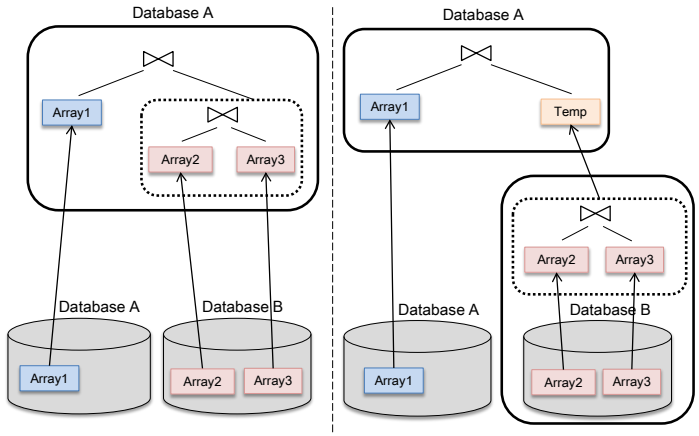


Figure 4.8: Query plans for query performance.

erated query plans need to be considered. Figure 4.8 describes an example of query plans. The join query for *array2* and *array3*, stored in a database B, is possible to be performed in either a database A or database B. Even though the result of the query is the same, the performance can be different. For instance, because of network overhead, a query (*e.g.*, aggregate query) which reduces the size of a foreign array can improve the performance rather than receiving the whole arrays and then processing the query.

Second, given that an array is sent to a local SciDB, network bandwidth is fully used to support transmission throughput. Meanwhile, most disk-based database systems may have suffered from disk I/Os, considered a bottleneck in processing a query. Because of these reasons, we consider the relationship between disk I/O and network bandwidth.

Assuming that the query is read-only, and each instance is allocated on a disk (to maximize the disk throughput) whose throughput is the same, disk I/O becomes a bottleneck if

$$N \times T < B$$

where N is the number of instances, T is the throughput of disk, and B is network bandwidth. We evaluate and describe the relationship in Section 4.4.3 in more detail.

4.4 Experiments

In this section, we evaluate the three database techniques: selective scan for filter operator, *SLS*, and *SDF*. We investigate and analyze the overall query processing performance, comparing vanilla SciDB with proposed techniques.

Experimental Environment

For the experiment, we used ten nodes on an Intel i7-4770 3.40GHz CPU, 8GB DDR3 1600MHz RAM, 128GB SSD and 1TB 7200RPM HDD connecting to 1Gbps network switch. They run Ubuntu 14.04.2 with Linux kernel 3.13 and the installed version of SciDB is 14.12.0.8739, where a vanilla SciDB and our modified SLS concurrently reside. We used MPICH2 3.0.4 as an implementation of MPI. Each node has a SciDB instance. To evaluate the federated database system, the experiment was conducted on ten nodes, five nodes for a local SciDB cluster and the other five nodes for a remote SciDB cluster.

Data used for experiments were two different types: MODIS SeaWiFS and MODIS remote-sensing reflectance (in short, RRS). We used SeaWiFS L3 Chlorophyll (CHL) data provided by NASA (NASA Goddard Space Flight Center). As a collection of the earth images of the satellite SeaStar, the data have three dimensions (*longitude, latitude, time*) and values measured at each unit (*chlorophyll*). Each cell is an 8-day average of chlorophyll of the earth with a resolution of 9km x 9km. A single HDF5 file consists of these values per one 8-day period, modeled as a dataset having two dimensions (longitude and latitude, 4,320 x 2,160).

Each chunk has 1 million cells, and the chunk size of a one-dimensional array is a million rows. In the three dimensional array, the chunk size is 1000 (longitude) x 1000 (latitude) x 1 (time). The total number of chunks in a database is 8,115. The number of cells is 5,048,179,200. MODIS RRS has a higher resolution (4km x 4km and 8,640 x 4,320) than SeaWiFS but measures different experimental values known as remote-

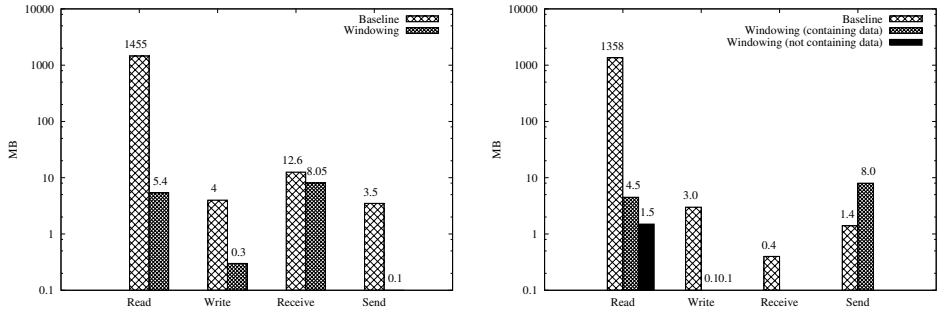
Query 1	SELECT $*$ FROM *CHL* WHERE *longitude* $\leq$ 1000 AND *latitude* $\leq$ 1000 AND *time* = 1997241 (0.18% of whole cells)
Query 2	SELECT $*$ FROM *CHL* WHERE *longitude* $\geq$ 500 AND *longitude* $\leq$ 1500 AND *latitude* $\geq$ 500 AND *latitude* $\leq$ 1500 AND *time* $\leq$ 2000365 AND *chl* $>$ 0 (0.57% of whole cells)
Query 3	SELECT $*$ FROM *CHL* WHERE *chl* $>$ 0.001
Query 4	SELECT $*$ FROM *CHL* WHERE *longitude* $\geq$ 4000 AND *latitude* $\leq$ 500 (1.71% of whole cells)
Query 5	SELECT $*$ FROM *CHL* WHERE *longitude* $\geq$ 500 AND *longitude* $\leq$ 1500 AND *time* $<$ 1998365 (2.65% of whole cells)
Query 6	SELECT $*$ FROM *CHL* WHERE *time* $\geq$ 2009001 AND *chl* $>$ 0.1 (8.5% of whole cells)

Table 4.2: Representative Queries

sensing reflectance. Similar to SeaWiFS, the experimental data are stored in HDF5 format. The number of files in total is 545, and the number of cells is 20,342,016,000.

4.4.1 Selective Scan for Filter Operator

We ran six queries, as shown in Table 4.2, and measured the elapsed time, amounts of I/O, and network traffic. These queries represent range searches based on a different combination of attributes and coordinates. Query 1, 4, and 5 include a predicate based only on coordinates. Query 3 includes a predicate based only on attribute chl. Query 2 and 6 include a predicate combined with dimensions and an attribute. Figure 4.9 indicates amounts of I/O and network traffic in all nodes when processing Query 1. Slave



(a) Master node (b) Slave nodes

Figure 4.9: Amounts of I/O and network communication (Query 1)

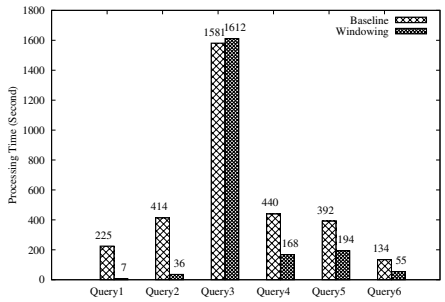
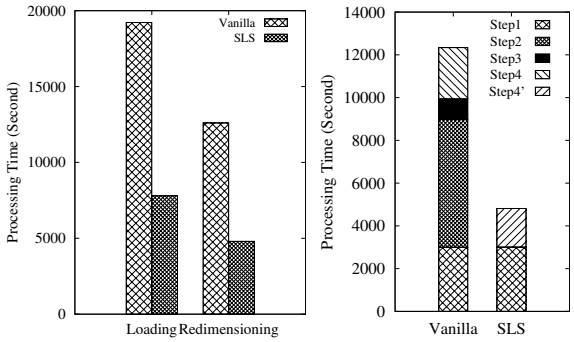


Figure 4.10: The elapsed time for processing queries

nodes send data to the master node, and this result is in Figure 4.9a. Figure 4.9b shows amounts of I/O and network traffic comparing the current filter operator to *windowing*. Results are distinguished whether data are satisfied with the condition of queries. In windowing, an instance with relevant data only occurs the data read and communications.

Figure 4.10 shows the elapsed time for processing six queries. The overall performance improved for all queries, except for Query 3. Especially, the elapsed time of Query 1 is reduced by 96%. The number of corresponding chunks for the condition varies by data selectivity. However, suppose a query includes predicates that do not use spatial information (coordinates), such as Query 3. The performance is hardly im-



(a) Loading and redimensioning time

(b) Breaking down the time on redimensioning

Figure 4.11: Comparison between vanilla and *SLS*, with CHL

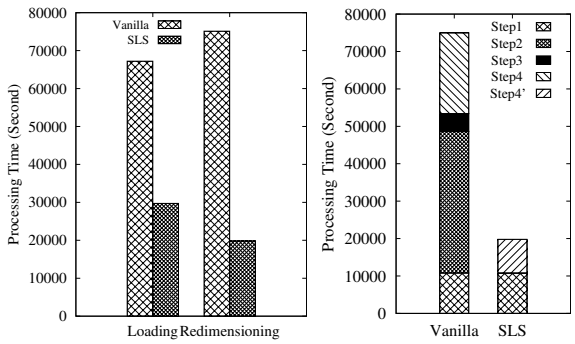
proved, and even worse because *windowing* consumes much time for computing the query boundaries.

4.4.2 Scalable Loader

We evaluate SLS compared to vanilla SciDB concerning loading and redimensioning time. Especially in redimensioning stage, we break down steps to analyze more detail. In addition, as increasing data size, we measured the processing time of data loading and observed the performance trend. After that, we evaluated scalability while increasing the number of nodes.

Loading and Redimensioning

Figure 4.11 shows the processing time of data loading. SLS reduces the entire loading time by about 60%. There are two reasons. First, SLS reduces the processing time because we remove redundant I/O overhead to make temporal files. Specifically, in vanilla SciDB, since the master node makes temporal files (e.g., TXT file format) by itself, all slaves should wait until files are created, which delays the loading step. However, in SLS, the temporal files are skipped to be stored, and files for loading to



(a) Loading and redimensioning time

(b) Breaking down the time on redimensioning

Figure 4.12: Comparison between vanilla and SLS, with RRS

SciDB are sent on the fly from the master node to slave nodes (distributing workload).

Second, we use MPI instead of SSH. MPI has a higher speed than SSH and can make the master node send data directly to slave nodes, related to minimizing I/O overhead. As files are distributed with data conversion, the I/O is not incurred to make intermediate files.

In redimensioning stage, the data are sorted before loading, which contributes to bypassing the sorting. Likewise, the redistribution step is removed since the master node already distributes data to the instances where data eventually should arrive. The processing time of the preparation step (step 3) also is removed because preparation work for the redistribution process is not required. Data synchronization only remained (step 4). As a result, the loading and redimensioning times are reduced compared to vanilla SciDB by optimizing and removing redundant steps.

Moreover, we used RRS data, larger than CHL, to measure its performance. Figure 4.12 shows the loading and redimensioning time for the RRS data. Reducing the redimensioning time as well as the loading time, we achieve the performance of data loading remarkably. SLS achieves the performance improvement by reducing 65% of the entire loading time. Compared to the CHL, there is an opposite result that the redi-

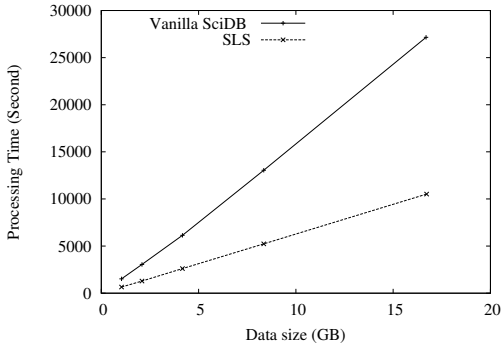


Figure 4.13: Trend of time with respect to the data size.

mensioning time takes longer than the loading time. It implies that the redimensioning cost is more expensive because of sorting and re-distributing data when the data volume is enormous. Our approach streamlines the redimensioning stages by removing or reducing those expensive parts.

Data Size

Since the scientific data is huge and SciDB is for handling massive data, we evaluate the performance with the data size. We vary the data size to observe the processing time trend along with the size to show the benefit of our approach. In this experiment, we use CHL data, and the number of files increases to vary the data size.

Figure 4.13 shows the processing time with respect to the whole process with different data sizes. Both increase linearly, but they have various aspects of the performance. Compared to vanilla SciDB, SLS achieves about 60 - 65% of the processing time because we subtract and simplify the processes, and thus essential steps remain. Based on this, we conclude that our approach is appropriate for larger data.

Scalability

As SciDB has shared-nothing architecture, scalability is crucial for the system. By minimizing the number of shared data or interferences, systems can handle an amount

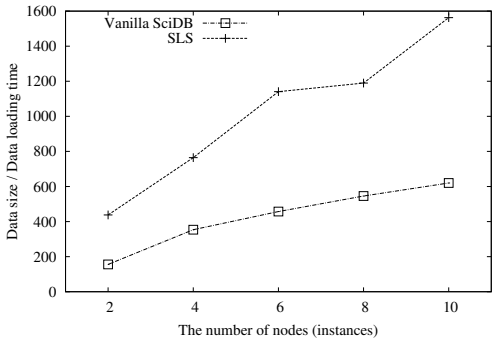


Figure 4.14: Comparing with vanilla-SciDB and SLS for scalability

of data simultaneously. If each node can run without synchronization, the system can process loading in parallel, minimizing idle time. Unless all instances are dependent on each other, adding an extra node also enhances the scalability. To evaluate scalability, we increased the number of nodes and measured the elapsed time of data loading. We use CHL, and the number of nodes increases by 2, 4, 6, 8, and 10.

Figure 4.14 shows the scalability of SLS and vanilla SciDB. The x-axis is the number of nodes (instances), and the y-axis is the amount of data size divided by elapsed time (i.e., data processing rate). SLS is more scalable than vanilla SciDB. SLS removes the all-to-all exchange in redimensioning, related to the communication cost.

However, when the number of nodes is eight, the graph does not grow linearly. The system should synchronize all nodes in the last step. The synchronization consumes significant time and hinders the performance improvement even though distributing workload reduces the whole processing time. The gradient of the graph means the amount of processed data per second dealt with by each node. As the SLS gradient average is higher than that of vanilla SciDB, the data processing of each node is fast.

4.4.3 Federated Database System for Scientific Data

In order to evaluate *SDF* and its connection models, we used five AFL queries: *scan*, *aggregate*, *store*, *join*, and *merge*. *Scan* and *aggregate* queries are used to analyze

Query	Scan	Aggregate
Vanilla SciDB	467	6,192
SDF	413	3,380

Table 4.3: The processing time (second) of scan and aggregate queries.

the query performance. We used these two queries as representative queries, using the streaming and materializing operators, respectively. The streaming operator does not need to load the entire array in memory at once, but the materializing operator processes the array after materializing it in memory. The *store* query stores a foreign array in storage. The *join* and *merge* queries access and concatenate two arrays that we mainly used for federated queries. A join query concatenates attributes by each dimension, but a merge query concatenates dimensions in two arrays. Without *SDF*, two queries are infeasible to be processed if two arrays are stored in separate databases. In addition to those queries, we used a query, *between*, to read an array selectively. We did not mainly use this query to evaluate *SDF*, but we show that *SDF* can process combination queries.

Vanilla SciDB vs. SDF

Since the vanilla SciDB does not support federated queries, we used two queries, *scan* and *aggregate*, that access only a database. A scan query reads CHL array, and an aggregate query computes an average of *chlorophyll* over the period using CHL and groups the result by longitude and latitude. In this evaluation, we used the cluster-to-master model of *SDF*.

Table 4.3 shows the query processing time of *SDF* and the vanilla SciDB in the 1G switch. *SDF* is slightly faster than the vanilla SciDB using the scan query. In the vanilla SciDB, once a query is posed to a remote SciDB, its master sends a foreign array to a local SciDB master. On the other hand, in *SDF*, the remote SciDB master distributes

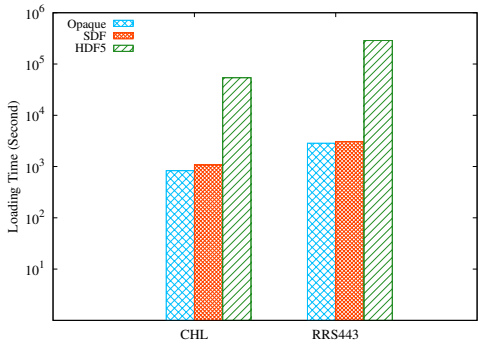


Figure 4.15: Comparing *SDF* with a SciDB loader using raw data (opaque and HDF5) in terms of the loading time.

sub-arrays of a foreign array to every local SciDB instance. The local SciDB instances pose a local federated query, which can process concurrently in the remote SciDB. As a result, *SDF* reduces the query processing time compared with the vanilla SciDB.

Meanwhile, *SDF* achieves the 1.8x speedup on the aggregate query. Network overhead for sending a foreign array rarely affects the processing time because aggregation reduces the data size (by this query, the size is 0.2% of the original data size). Rather, loading the entire array into memory and aggregating computation affect its performance, accounting for the most query processing time. These are processed concurrently in a remote SciDB, which improves the query performance.

Compared with the vanilla SciDB, *SDF* needs one more step, converting a foreign array to a virtual array. As the query processing is done in a remote SciDB, a foreign array is sent to a local SciDB and mapped into a virtual array in a local SciDB. Despite the additional step, its overhead is minimized by caching effects because the virtual array may be reused shortly after by a combination query. In our experiment, all SciDB instances read and send them to master to display the result.

Next, we compare two systems in terms of data loading. Since loading cost is expensive, it is the main bottleneck prior to data analytics. In this experiment, we loaded raw data into a multi-dimensional array using opaque and HDF5 format, the

raw data of CHL and RRS443. The opaque format is the fastest format to save an array into disk or load from disk. However, it is not well portable because other databases must have their schema in advance. This format is usually employed for data backup.

To load raw data (*i.e.*, HDF5), it should be converted to loadable formats (*e.g.*, CSV) that are general purpose. The loader distributes data to SciDB instances and loads them in parallel. Since the loaded form is a one-dimensional array, it is required to be transformed to an n-dimensional array. We used the *redimension* query to reshape data. By this conversion, the data close to each other can be closed physically. The data loading time is the elapsed time of all processes (*e.g.*, from HDF5 files to a three-dimension array). We do not include the extra time of both data formats into the loading time, such as transferring or downloading raw data, since it may be different under data sources. In the case of *SDF*, a source array is in a remote SciDB.

Figure 4.15 shows the loading time of *SDF* and SciDB loader using opaque and HDF5 formats. *SDF* significantly reduces the data loading time compared with the HDF5 format, even though the time to transfer or download data for SciDB loader is not counted. With CHL and RRS443, the SciDB loader is 50x and 100x slower than *SDF*, respectively. While data are loaded, the I/O and network overhead are inevitable. However, using *SDF*, the overhead can be avoidable by removing a few steps including data conversion (*e.g.*, from raw data to loadable format).

Moreover, all-to-all communication is needed for redimensioning to exchange data, which is extreme overhead but is not required in *SDF*. On the other hand, using the opaque format, *SDF* is slightly slower than SciDB loader. Since SciDB strictly understands the format as a loaded array format, and *SDF* includes data transmission, this format is the fastest to load. However, to use this format, a database should obtain an array schema ahead of data loading because it is not loadable without the corresponding schema or information. It implies that *SDF* may be regarded as an alternative for data loading once raw data have been loaded into other databases.

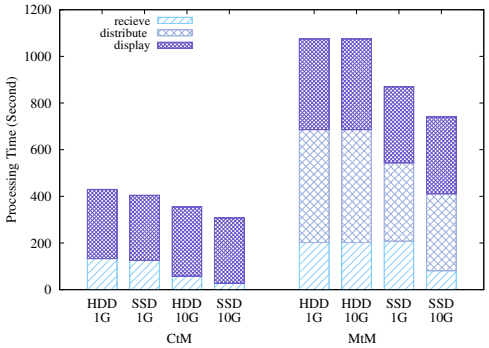


Figure 4.16: Break-down of the processing time.

- Receive: receiving a foreign array from remote SciDB.

- Distribute: a local master distributes sub-arrays to local slave instances.

- Display: a local master displays a foreign array.

Connection Model and Query Plan

We analyze the query processing using various queries while evaluating the performance of connection models, namely master-to-master (in short, MtM) and cluster-to-master (in short, CtM). Except for a scan query, other queries are evaluated by using different query plans, but the results are the same.

Scan Query. To study in-depth, we investigate the relationship between the throughput of disk I/O and network bandwidth, using HDD and SSD with both 1G and 10G switches, respectively. Figure 4.16 shows the performance of *SDF* with two models in the break-down of the query processing time. Since CtM avoids distributing data, its processing time is about 2x faster than that of MtM. The display time is the same in all results because the master reads all cells in a chunk. After the iteration finishes, the next chunk is received.

Next, when we compare the performance of two models by HDD and SSD using the 1G switch, they do not have a substantial difference in the receiving time. Consequently, the network bandwidth of the 1G switch is not sufficient to support the throughput of HDD and SSD. On the other hand, in terms of the throughput of HDD,

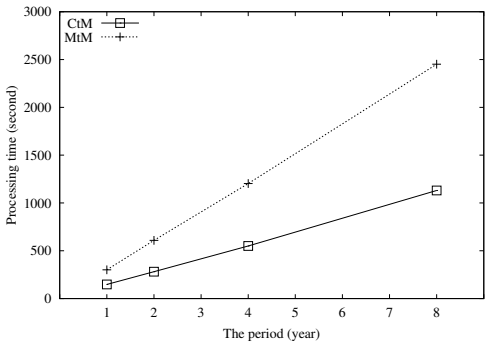


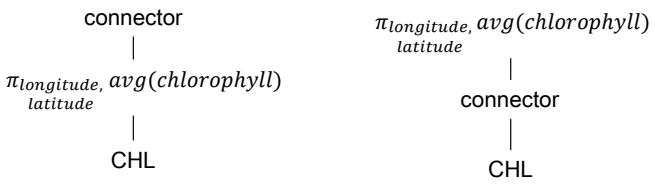
Figure 4.17: Comparing the CtM with MtM in varying the period.

the 10G switch affects the receiving time, 2x faster than the 1G switch. The receiving time of using SSD is also reduced, achieving 5x compared with the 1G switch. Still, the 10G switch cannot support the SSD throughput. For this reason, we do not deliver the performance considerably by SSD using the 10G switch compared with using the 1G switch. The MtM also shows the similar result with CtM. As the performance in the 10G switch compared with that in the 1G switch, the receiving time and distributing time are reduced. By SSD, the 10G switch can improve the receiving time compared with the 1G switch.

We also measured the query processing time in Figure 4.17 using RRS443 with varying the range in the time dimension to increase the data size. The processing time is proportional to the data size, and CtM is 2x faster than MtM. Thus, the data size affects the processing time on the scan query in two models.

Aggregate Query. We used two different plans using aggregate queries to observe the effects of materializing operators. The queries are depicted in Figure 4.18.

CA query (Figure 4.18a) aggregates CHL, computing an average of *chlorophyll* for longitude and latitude over the years, and a remote SciDB sends the foreign array to a local SciDB. In contrast, AC query (Figure 4.18b) is that a local SciDB receives CHL from a remote SciDB and aggregates the array for computing an average of *chlorophyll* over the years.



(a) CA query (b) AC query

Figure 4.18: Relational algebra of aggregate queries.

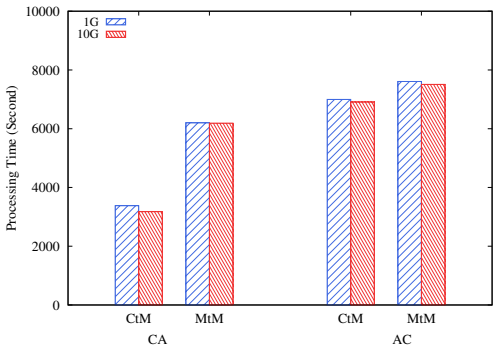
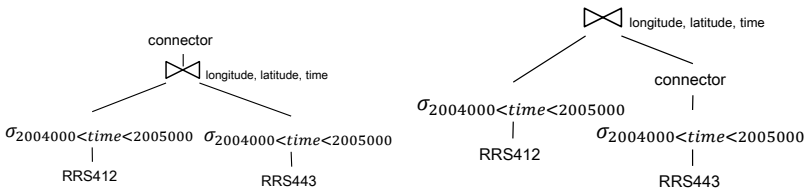


Figure 4.19: The processing time of aggregate queries.

Figure 4.19 shows the processing time of the queries. Since the computation accounts for most of the processing time, the network overhead can be negligible. Except for the CA query result using CtM model, three results show similar performance. The AC query relatively takes a long time because it needs the local query processing after a local SciDB receives CHL array. Especially, CA query using MtM is similar to the vanilla SciDB as the query processing mechanism is similar. However, as CA query using CtM is executed concurrently, the query processing time is reduced by half compared with MtM. The network switch also affects the performance in *SDF*, but it is insignificant. Each SciDB instance should collect corresponding cells to aggregate an array, which needs to shuffle data. By MtM, AC query requires the synchronization only once because a single query is posed. On the other hand, CtM needs multiple times, the number of localized aggregate queries (*i.e.*, the number of instances), to



(a) CJ query (b) JC query

Figure 4.20: Relational algebra of join queries.

synchronize remote SciDB instances. This communication overhead has an effect on the performance of CtM.

Join Query. To analyze the performance of a join query, we used two plans in Figure 4.20: (1) a join is performed for two arrays, and the result is sent to a local SciDB (CJ query); (2) an array is sent to a local SciDB, and then join is performed (JC query).

Figure 4.21 shows the performance of two join queries. Since CtM bypasses data distribution, it is faster than MtM. The reason is similar to the scan query. On the other hand, in MtM, JC query is faster than CJ query because the transmitting data size of JC query to a local SciDB is smaller than that of CJ query.

In the 10G switch, the performance is not achieved substantially compared with the 1G switch. The reason is similar to the scan query that the throughput of HDD hardly supports the higher network bandwidth. To improve the join query performance in CtM, executing the operation in a single database is better than in separate databases. In MtM, in contrast, since receiving a large volume of data causes overhead, reducing the size of data is more helpful in improving the performance. MtM is sensitive to the change in the volume of receiving data. Hence, there is no best common plan of join query in both models.

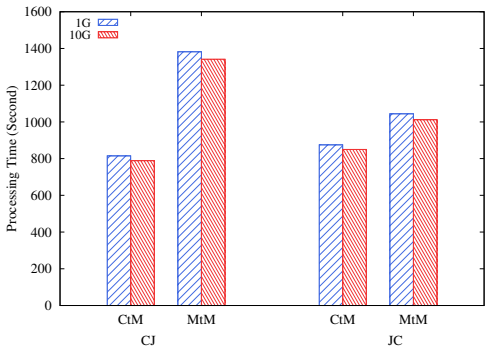
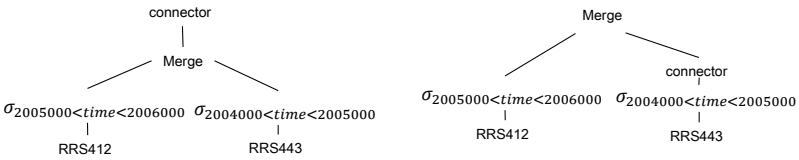


Figure 4.21: The processing time of join queries.



(a) CM query

(b) MC query

Figure 4.22: Relational algebra of merge queries.

Merge Query

The query is described in Figure 4.22. CM query (Figure 4.22a) merges two arrays in a remote SciDB and then receives the foreign array. MC query (Figure 4.22b) receives the remote array and merges it with a local array. The results of the queries are in Figure 4.23.

The join and merge queries have a similar feature that integrates two arrays, but has different read patterns. To combine two arrays, a merge query reads two arrays. The array with smaller start coordinates is read first and then the other array is read. This pattern is the same as reading two arrays in a row. With this feature, the data size is a more important factor in the performance. Thus, MC query is faster than CM query.

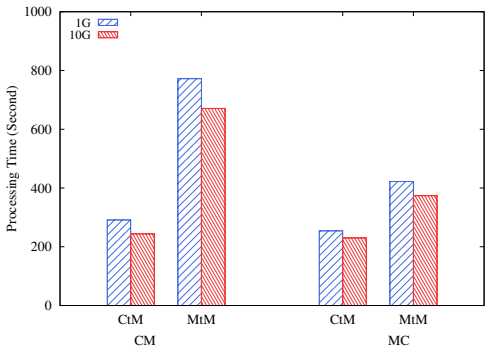


Figure 4.23: The processing time of merge queries.

Discussion

- Compared with the vanilla SciDB, *SDF* can reduce the query processing time up to 45% for an aggregation query in our experiment.
- With the experiments of scan and aggregate queries, the bottleneck of the query processing depends on query types. Most query processing suffers from network traffic, whereas the computation for aggregate query accounts for the whole processing time.
- The overhead to process a federated query is minimized by concurrent execution and caching effects.
- The cluster-to-master model provides better performance than the master-to-master model because bypassing the foreign array distribution.
- Federated query plans affect the query performance. In general, as the result size is small, the query performance can be improved because of reducing network overhead.

4.5 Summary

In SciDB, most queries for scientific data analysis utilize spatial information. However, the filter operator for processing queries reads all data without considering features of array-based DBMSs and spatial information. In this chapter, we propose a selective scan method, *windowing*, which only retrieves data selectively corresponding to a range. It significantly reduces the processing time of a selection query when queries contain a range search. Our implementation of the filter operator will accelerate scientific analysis and relieve the hesitation in using AQL statements. It will also enhance SciDB to handle a massive amount of scientific data in a more scalable manner.

Next, we propose a new scalable approach to improving the data loading process for SciDB. We use MPI to send data from the master node to slaves in the loading stage and reduce the I/O overhead significantly compared with SSH used by the vanilla loader of SciDB. Moreover, SLS sorts data separately in the redimensioning stage and avoids the all-to-all communication for data redistribution altogether. The results from our experiments also show that the reduction in loading time by SLS is indeed significant.

Last, in order to process the complicated analysis using data in separate databases, the database federation is necessary. We present *SDF*, implemented in SciDB using UDO, adopting the principle of a federated database system for a shared-nothing architecture. *SDF* abstracts away integrating processes, preserving system features and the primary authority of each SciDB; it facilitates the data sharing and exchange within multiple databases, providing an interconnect interface. As *SDF* enables the federated query processing, distributed and parallel executions could be allowed beyond clusters over significantly large data sets. It will be one of the big data solutions for numerous consolidated resources and help enhance scientific analytics.

Chapter 5

Conclusion

In this chapter, we conclude this dissertation with summary of our contributions and future directions.

5.1 Contributions

We first present the in-memory array processing system, named Spangle, in the map-reduce environment. We introduce Spangle, an array processing system that provides declarative interfaces. It can process complex analytics in parallel with large-scale arrays. In order to process raster data, which is skewed and sparse, Spangle identifies empty cells (*i.e.*, null value) as a bitmask. Furthermore, since managing data in a concise form is essential for in-memory systems, Spangle provides data compression in three different modes. It facilitates high-level programming for both iterative algorithms and interactive analysis. This high-level programming 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.

Next, we describe three techniques we propose. We first describe several approaches

to improve the query processing and data loading on SciDB. Most queries for scientific data analysis utilize spatial information. However, filter operator for processing queries reads all data without considering features of array-based DBMSs and spatial information. We propose a selective scan method, *windowing*, which only retrieves data selectively corresponding to a range. It reduces the processing time of a selection query significantly when queries contain a range search. Moreover, we present a scalable loader, which improves the data loading process for SciDB. We use MPI to send data from the master node to slaves in the loading stage such that the I/O overhead can be reduced significantly compared with SSH being used by the vanilla loader of SciDB. Moreover, sorting is done separately from the redimensioning stage so that the all-to-all communication for data redistribution can be removed altogether. That is, two serious bottlenecks, namely sort and redistribution, in the redimensioning stage are eliminated. The results from our experiments also show that the reduction in loading time by SLS is indeed significant.

Then, we suggest a federated system, which facilitates access to multiple databases at the query level. To process the complicated analysis using data in separate databases, the database federation is necessary. We present *SDF* adopting the principle of a federated database system for a shared-nothing architecture, built in SciDB using UDO. It abstracts underlying databases, and users can process queries such as joining data in different databases, even in geospatially separated databases.

Our work can enhance scalable array processing and management with a massive amount of scientific data, such as raster data and matrices, on Apache Spark and SciDB. Furthermore, our optimization technique processing can accelerate the creation of a more accurate machine learning model. It will be one of the big data solutions for numerous consolidated resources and handling a massive amount of scientific data in a more scalable manner.

5.2 Future Direction

In this section, we describe several concerns and discuss promising research directions for future work.

Adapting Systems for Modern Hardware. Recently, we can achieve high performance with modern hardware where many applications have heavy computation tasks. In order to fully use hardware acceleration, we often consider system design customized for the hardware. Graphics processing units (GPUs) have become an emerging and powerful co-processor for many applications in scientific computing and database systems. A large number of CPU cores can process elements of an array in parallel, which implies that systems based on the array data model are suitable for GPU because they do not need the data transformation. Moreover, remote direct memory access (RDMA) based networks improve data transmission performance in distributed systems. With the RDMA network, conventional network-optimized approaches are no longer the best fit.

Data Compression. In distributed systems, the data size is an important factor which affects the entire processing performance. Likewise, in-memory systems must regard the data size. Its performance drops substantially when the working data set exceeds available memory. There are many methods to compress data, but it is difficult to suggest the algorithm that supports minimum data size with elapsed time to compress and decompress data.

As we described in Section 3.2.3, the algorithms for bit-vectors access have been researched. While Spangle processes arrays, operations transform them, including building time to elapsed time. Thus, we need to consider the encoding cost to build bitmasks as well as payloads.

Nested Array. The nested arrays can be used when arrays have hierarchical relationships such as adaptive mesh refinement (AMR), but it is a challenge to represent them directly. As the size of the object, which contains all arrays, is naturally large, data shuffling incurs much network overhead with the large data size in distributed

systems. Thus, the implementation is different to avoid it. For example, SciDB implements nested arrays by adding extra dimensions. If a system provides the nested array and its operations (*e.g.*, getNextLevel()) efficiently, it can leverage productivity and usability.

Irregular Chunks. The scientific data are skewed and sparsely distributed, and there are studies to support the irregular chunks [17]. It is beneficial for data management because it reduces skew in parallel array processing. However, it has challenges for operations with two arrays. If two arrays have different chunks, it is required to support efficient access to subsets of arrays. If a system follows the in-memory architecture, it should maintain metadata or data structures to index them in limited memory space.

Optimization for Machine Learning Algorithms. As described, machine learning algorithms rely on linear algebra, which the array data model can support efficiently. In Chapter 3, Spangle shows effective matrix operations. Likewise, we can propose optimized design and overcome implementation challenges for machine learning algorithms based on Spangle.

Tensor Operation Support. Tensor is a mathematical representation for multi-dimensional arrays, including vector (*i.e.*, one-dimensional array) and matrix (*i.e.*, two-dimensional array). If Spangle supports tensor operations such as tensor decomposition, it can be widely used in many interesting applications such as clustering, trend detection, and anomaly detection.

Bibliography

[1] "Large Synoptic Survey Telescope." https://www.lsst.org/.

[2] 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.

[3] K. T. Butler, D. W. Davies, H. Cartwright, O. Isayev, and A. Walsh, "Machine Learning for Molecular and Materials Science," *Nature*, vol. 559, no. 7715, pp. 547–555, 2018.

[4] N. Laanait, Z. Zhang, and C. M. Schlepütz, "Imaging Nanoscale Lattice Variations by Machine Learning of X-Ray Diffraction Microscopy Data," *Nanotechnology*, vol. 27, no. 37, p. 374002, 2016.

[5] F. Hu, C. Yang, J. L. Schnase, D. Q. Duffy, M. Xu, M. K. Bowen, T. Lee, and W. Song, "ClimateSpark: An In-memory Distributed Computing Framework for Big Climate Data Analytics," *Computers & geosciences*, vol. 115, pp. 154–166, 2018.

[6] 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)*, pp. 2020–2026, IEEE, 2015.

[7] T. N. C. for Supercomputing Applications, "Hierarchical Data Format Version5." https://www.hdfgroup.org/HDF5/, Feb 2017.

[8] 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.

[9] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. Saltz, "Titan: a High-Performance Remote-Sensing Database," in *Proceedings 13th international conference on data engineering*, pp. 375–384, IEEE, 1997.

[10] C. Chang, A. Acharya, A. Sussman, and J. Saltz, "T2: A customizable parallel database for multi-dimensional data," *ACM SIGMOD Record*, vol. 27, no. 1, pp. 58–66, 1998.

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

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

[13] A. P. Marathe and K. Salem, "Query Processing Techniques for Arrays," in *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pp. 323–334, 1999.

[14] A. R. vanBallegooij, A. P. deVries, and M. L. Kersten, "RAM: Array Processing over a Relational DBMS," *Information Systems [INS]*, no. R 0301, 2003.

[15] 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.

[16] A. v. Ballegooij, R. Cornacchia, A. P. d. Vries, and M. Kersten, "Distribution Rules for Array Database Queries," in *International Conference on Database and Expert Systems Applications*, pp. 55–64, Springer, 2005.

[17] 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*, pp. 253–264, 2011.

[18] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson, "The TileDB Array Data Storage Manager," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 349–360, 2016.

[19] R. A. R. Zalipynis, "Chronosdb: Distributed, file based, geospatial array dbms," *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1247–1261, 2018.

[20] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt, "SciHadoop: Array-Based Query Processing in Hadoop," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 66, ACM, 2011.

[21] J. Buck, N. Watkins, G. Levin, A. Crume, K. Ioannidou, S. Brandt, C. Maltzahn, and N. Polyzotis, "SIDR: Efficient Structure-Aware Intelligent Data Routing in SciHadoop," *Technical Report UCSC-SOE-12–08*, 2012.

[22] Y. Wang, W. Jiang, and G. Agrawal, "Scimate: A Novel Mapreduce-like Framework for Multiple Scientific Data Formats," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pp. 443–450, IEEE Computer Society, 2012.

[23] B. Dong, K. Wu, S. Byna, J. Liu, W. Zhao, and F. Rusu, "ArrayUDF: User-Defined Scientific Data Analysis on Arrays," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 53–64, ACM, 2017.

[24] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani, "Parallel Data Analysis Directly on Scientific File Formats," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 385–396, ACM, 2014.

[25] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "HAMA: An Efficient Matrix Computation with the MapReduce Framework," in *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pp. 721–726, IEEE, 2010.

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

[27] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann, "Spatio-Temporal Retrieval with RasDaMan," in *VLDB*, pp. 746–749, 1999.

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

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

[30] M. Kersten, Y. Zhang, M. Ivanova, and N. Nes, "SciQL, a Query Language for Science Applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, (Uppsala, Sweden), pp. 1–12, ACM, 2011.

[31] 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.

[32] 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*, pp. 423–438, ACM, 2013.

[33] 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*, pp. 1383–1394, ACM, 2015.

[34] 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.

[35] 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)*, pp. 599–613, 2014.

[36] 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)*, pp. 2–2, USENIX Association, 2012.

[37] PostgreSQL, "PostgreSQL 9.6.1 Documentation." https://www.postgresql.org/-docs/9.6/static/postgres-fdw.html, 2017.

[38] J. Liu, E. Racah, Q. Koziol, R. S. Canon, and A. Gittens, "H5spark: Bridging the I/O Gap between Spark and Scientific Data Formats on HPC Systems," *Cray user group*, 2016.

[39] 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*, pp. 1410–1413, IEEE, 2016.

[40] 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*, pp. 231–242, IEEE, 2011.

[41] 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.), pp. 1187–1192, IEEE Computer Society, 2013.

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

[43] 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.

[44] "H2O." https://www.h2o.ai/.

[45] 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.

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

[47] 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)*, pp. 265–283, 2016.

[48] "PyTorch." https://pytorch.org//.

[49] H. Xing and G. Agrawal, "COMPASS: Compact Array Storage with Value Index," in *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*, pp. 1–12, 2018.

[50] H. Xing and G. Agrawal, "Accelerating Array Joining with Integrated Value-Index," in *Proceedings of the 31st International Conference on Scientific and Statistical Database Management*, pp. 145–156, 2019.

[51] R. A. R. Zalipynis, "BitFun: Fast Answers to Queries with Tunable Functions in Geospatial Array DBMS," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2909–2912, 2020.

[52] S. Ladra, J. R. Paramá, and F. Silva-Coira, "Compact and Queryable Representation of Raster Datasets," in *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*, pp. 1–12, 2016.

[53] N. R. Brisaboa, S. Ladra, and G. Navarro, "Compact Representation of Web Graphs with Extended Functionality," *Information Systems*, vol. 39, pp. 152–174, 2014.

[54] Z. Lyu, H. H. Zhang, G. Xiong, G. Guo, H. Wang, J. Chen, A. Praveen, Y. Yang, X. Gao, A. Wang, *et al.*, "Greenplum: A Hybrid Database for Transactional and Analytical Workloads," in *Proceedings of the 2021 International Conference on Management of Data*, pp. 2530–2542, 2021.

[55] Oracle, "Oracle Data Integrator." http://www.oracle.com/technetwork/middleware/data-integrator, 2016.

[56] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear, "The Vertica Analytic Database: C-Store 7 Years Later," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, 2012.

[57] V. Josifovski, P. Schwarz, L. Haas, and E. Lin, "Garlic: A New Flavor of Federated Query Processing for DB2," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 524–532, ACM, ACM, 2002.

[58] D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling, "Split query processing in polybase," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 1255–1266, ACM, 2013.

[59] Y. Tian, F. Özcan, T. Zou, R. Goncalves, and H. Pirahesh, "Building a Hybrid Warehouse: Efficient Joins between Data Stored in HDFS and Enterprise Warehouse," *ACM Transactions on Database Systems (TODS)*, vol. 41, no. 4, p. 21, 2016.

[60] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik, "The BigDAWG Polystore System," *ACM Sigmod Record*, vol. 44, no. 2, pp. 11–16, 2015.

[61] MySQL, "Federated Storage Engine of MySQL." https://dev.mysql.com/doc/refman/5.7/en/federated-storage-engine.html, 2017.

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

[63] 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.

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

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

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

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

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

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

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

[71] "sdsl-lite." https://github.com/simongog/sdsl-lite.

[72] G. J. Jacobson, *Succinct Static Data Structures*. Carnegie Mellon University, 1988.

[73] D. Clark, "Compact Pat Trees," 1997.

[74] R. González, S. Grabowski, V. Mäkinen, and G. Navarro, "Practical Implementation of Rank and Select Queries," in *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pp. 27–38, CTI Press and Ellinika Grammata Greece, 2005.

[75] S. Vigna, "Broadword Implementation of Rank/Select Queries," in *International Workshop on Experimental and Efficient Algorithms*, pp. 154–168, Springer, 2008.

[76] F. Claude and G. Navarro, "Practical Rank/Select Queries over Arbitrary Sequences," in *International Symposium on String Processing and Information Retrieval*, pp. 176–187, Springer, 2008.

[77] G. Navarro and E. Providel, "Fast, Small, Simple Rank/Select on Bitmaps," in *International Symposium on Experimental Algorithms*, pp. 295–306, Springer, 2012.

[78] R. Raman, V. Raman, and S. R. Satti, "Succinct Indexable Dictionaries with Applications to Encoding K-ary Trees, Prefix Sums and Multisets," *ACM Transactions on Algorithms (TALG)*, vol. 3, no. 4, pp. 43–es, 2007.

[79] D. Okanohara and K. Sadakane, "Practical Entropy-Compressed Rank/Select Dictionary," in *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 60–70, SIAM, 2007.

[80] P. Elias, "Efficient Storage and Retrieval by Content and Address of Static Files," *Journal of the ACM (JACM)*, vol. 21, no. 2, pp. 246–260, 1974.

[81] R. M. Fano, *On the Number of Bits Required to Implement an Associative Memory*. Massachusetts Institute of Technology, Project MAC, 1971.

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

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

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

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

[86] 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*, pp. 69–77, ACM, 2011.

[87] 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*, pp. 281–288, 2007.

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

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

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

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

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

[93] 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), pp. 591–600, ACM, 2010.

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

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

[96] A. P. Marathe and K. Salem, "A Language for Manipulating Arrays," in *Proceedings of the 23rd VLDB Conference*, (Athens, Greece), pp. 46–55, Sept. 1997.

[97] N. Widmann and P. Baumann, "Efficient Execution of Operations in a DBMS for Multidimensional Arrays," in *the 10th International Conference on Scientific and Statistical Database Management*, (Capri, Italy), pp. 155–165, July 1998.

[98] A. P. Sheth and J. A. Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases," *ACM Computing Surveys (CSUR)*, vol. 22, no. 3, pp. 183–236, 1990.

[99] D. Heimbigner and D. McLeod, "A Federated Architecture for Information Management," *ACM Transactions on Information Systems (TOIS)*, vol. 3, no. 3, pp. 253–278, 1985.

[100] N. Widmann and P. Baumann, "Efficient Execution of Operations in a DBMS for Multidimensional Arrays," in *Scientific and Statistical Database Management, 1998. Proceedings. Tenth International Conference on*, pp. 155–165, IEEE, 1998.

초 록

과학분야의 관측과 시뮬레이션 및 실험은 대규모 과학 데이터를 생성하며, 다차원 배열로 생성되는 과학 데이터의 복잡한 분석에서 성능 향상이 지속적으로 요구되어져 왔다. 과학 데이터를 관리하기 위해 배열 데이터 모델을 주로 사용하는데, 이 모델은 물리적으로 서로 가깝게 데이터를 저장 관리하여 좌표 시스템의 지역성을 보장할 수 있다. 본 논문에서는 배열 데이터 모델을 기반으로 데이터 처리 및 관리에 중점을 두고 있다. 우선, 맵-리듀스 프레임워크인 Apache Spark에서 복잡한 배열 처리 계산 위한 시스템인 *Spangle*을 소개한다. Spangle은 배열 데이터 모델을 채택하여 래스터 데이터의 분석이나 선형 대수에 크게 의존하는 기계 학습 알고리즘을 용이하게 처리할 수 있도록 고안하였다. 더불어, 배열 기반 DBMS중 하나인 SciDB를 사용해 배열 처리 및 사용성을 개선을 하였다. 첫번째로 filter 연산자에서 주어진 조건에 관계없이 데이터 전체를 읽는 연산을 개선해, 조건에 따라 공간 정보를 활용한 부분적으로 읽는 방법을 제시하였다. 다음으로, 효율적인 데이터 적재를 제공하는 *SLS* 를 제안한다. 변환 과정을 간소화하고, 특히 대부분을 차지하는 데이터 적재에서 정렬과 재배포를 제거하는 방법을 사용하였다. 마지막으로는 복잡한 분석을 위해 최소한의 부하로 데이터를 연동할수 있는 *SDF* 시스템을 제안한다. *SDF* 는 연동 과정을 추상화하여 서로 다른 데이터베이스의 기본 권한을 보존하면서 쿼리 연산을 가능하게 하는 시스템이다.

주요어: 배열 데이터 모델, 과학 데이터, 배열 처리, 배열 관리

학번: 2015-30267