HAL
open science

Parallel Computation of PDFs on Big Spatial Data Using Spark

Ji Liu, Noel Moreno Lemus, Esther Pacitti, Fábio Porto, Patrick Valduriez

To cite this version:

Ji Liu, Noel Moreno Lemus, Esther Pacitti, Fábio Porto, Patrick Valduriez. Parallel Computation of PDFs on Big Spatial Data Using Spark. Distributed and Parallel Databases, 2020, 38, pp.63-100. 10.1007/s10619-019-07260-3 . lirmm-02045144

HAL Id: lirmm-02045144

https://hal-lirmm.ccsd.cnrs.fr/lirmm-02045144v1

Submitted on 21 Feb 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Parallel Computation of PDFs on Big Spatial Data Using Spark

Ji Liu · Noel Moreno Lemus · Esther Pacitti · Fabio Porto · Patrick Valduriez

the date of receipt and acceptance should be inserted later

Abstract We consider big spatial data, which is typically produced in scientific areas such as geological or seismic interpretation. The spatial data can be produced by observation (*e.g.* using sensors or soil instruments) or numerical simulation programs and correspond to points that represent a 3D soil cube area. However, errors in signal processing and modeling create some uncertainty, and thus a lack of accuracy in identifying geological or seismic phenomenons. Such uncertainty must be carefully analyzed. To analyze uncertainty, the main solution is to compute a Probability Density Function (PDF) of each point in the spatial cube area. However, computing PDFs on big spatial data can be very time consuming (from several hours to even months on a computer cluster). In this paper, we propose a new solution to efficiently compute such PDFs in parallel using Spark, with three methods: data grouping, machine learning prediction and sampling. We evaluate our solution by extensive experiments on different computer clusters using big data ranging from hundreds of GB to several TB. The experimental results show that our solution scales up very well and can reduce the execution time by a factor of 33 (in the order of seconds or minutes) compared with a baseline method.

Keywords Spatial data · big data · parallel processing · Spark

1 Introduction

Big spatial data is now routinely produced and used in scientific areas such as geological or seismic interpretation [10]. The spatial data are produced by observation, using sensors [46], [12] or soil instruments [25], or numerical simulation, using mathematical models [15]. These spatial data allow identifying some phenomenon over a spatial reference [19]. For instance, the

J. Liu
Inria and LIRMM, Univ. of Montpelier, France E-mail: jiliuwork@gmail.com

N. Moreno Lemus
LNCC Petrópolis, Brazil E-mail: nmlemus@gmail.com

E. Pacitti
Inria and LIRMM, Univ. of Montpelier, France E-mail: esther.pacitti@lirmm.fr

F. Porto
LNCC Petrópolis, Brazil E-mail: fporto@lncc.br

P. Valduriez
Inria and LIRMM, Univ. of Montpelier, France E-mail: patrick.valduriez@inria.fr

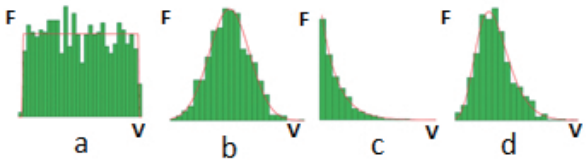


Fig. 1: **The distribution of a point.**

spatial reference may be a three dimensional soil cube area and the phenomenon a seismic fault, represented as quantities of interest (QOIs) of sampled points (or points for short) in the cube space. The cube area is composed of multiple horizontal slices, each slice having multiple lines and each line having multiple points. A single simulation produces a spatial dataset whose points represent a 3D soil cube area.

However, errors in signal processing and modeling create some uncertainty, and thus a lack of accuracy when identifying phenomenons. Such uncertainty must be carefully quantified. In order to understand uncertainty, several simulation runs with different input parameters are usually conducted, thus generating multiple spatial datasets that can be very big, *e.g.* hundreds of GB or TB. Within multiple spatial datasets, each point in the cube area is associated to a set of different observation values. The observation values are captured by sensors, or generated from simulation, at a specific point of the spatial area.

Uncertainty quantification of spatial data is of much importance for geological or seismic scientists [34], [45]. It is the process of quantifying the uncertainty error of each point in the spatial cube space, which requires computing a Probability Density Function (PDF) of each point [26]. The PDF is composed of the distribution type (*e.g.* normal, exponential) and necessary statistical parameters (*e.g.* the mean and standard deviation values for normal and rate for exponential).

Figure 1 shows that the set of observation values at a point may be modelled by four distribution types, *i.e.* uniform (a), normal (b), exponential (c), and log-normal (d). The horizontal axis represents the values (V) and the vertical axis represents the frequency (F). The green bars represent the frequency of the observation values in value intervals and the red outline represents the calculated PDF. Given a resulting PDF modeling the data distribution at a point, there may be some error between the distribution of the observation values and one produced by a PDF. During the process of fitting data to a PDF, we aim at reducing such PDF error (or error for short in the rest of the paper) to a minimum. For instance, the set of observation values corresponding to the QOI at a point obeys a normal distribution shown in Figure 1 (b). The mean value of the set of observation values (see Equation 1) may be used as a representative of QOI since it has the highest chance to be the QOI. However, the distribution can be different from normal. And analyzing uncertainty just using the mean or standard deviation values is difficult and imprecise. For instance, if the distribution type of simulated values is exponential (see Figure 1 (c)), we should take the value zero (different from the mean value of the simulated values) as the QOI value since it has the highest frequency. Thus, once the PDF of observations in a point has been computed, we can calculate the QOI value that has the highest probability. The QOI value can be assumed to represent the imprecision among the point observations, and computed across the whole spatial dataset as a measure of point observations uncertainty.

Calculating the PDF that best fits the observation values at each point can be time consuming. For instance, one simulation of an area of 10km (distance) * 10km (depth) * 5km (width) corresponds to 2.4 TB data with 10000 measurements at each point [2]. This area contains 6.25 * 10^8 points. The time to calculate the PDF with consideration of 4 distribution types (normal,

uniform, exponential and log-normal) can be up to several days or months using a computer cluster (or cluster for short).

The problem we address is how to efficiently compute PDFs under bounded error constraints. There are three main challenges, *i.e.* big data, long and complex computation and error bound. First, the input spatial datasets can be very big. Imagine that the size of each spatial dataset is several hundred MB, then 10,000 spatial datasets correspond to several TB, which makes it hard to process. Second, computing PDFs is complex and takes time. The PDF computation with the datasets of several TB in a single computer or a small cluster (6 nodes, each with 32 CPU cores) may take up to several months. Third, the PDF computation should not have unacceptable error, *i.e.* under error bound. For instance, if the error is bigger than 50%, then the result will not be representative.

Solutions for computing PDFs are now available in popular programming environments for numerical and statistical analysis, such as MATLAB and R. MATLAB provides libraries [31][36] that use a baseline method for PDF fitting. R provides the GLDEX library that supports the Generalized Lambda Distribution (GLD) family of PDFs. However, these implementations do not address the problem of efficient computation of thousands or millions of PDFs that may lead to redundant computation. Machine Learning (ML) techniques are widely used to process big data [4, 14] for the sake of prediction and classification [44]. They can be used for reducing the dimension of data [24], reducing the time for repeating work [8] and discovering knowledge based on big data [21]. However, unlike the traditional approach, we do not use ML to reduce the data dimension. Instead, we use ML to discover new knowledge in order to reduce redundant computation of PDFs. Furthermore, we use data aggregation to reduce the quantity of data to process.

In this paper, we propose a new solution to efficiently compute PDFs in parallel using Spark [49], a popular in-memory big data processing framework (see [29] for a survey on big data systems). The main reason to choose Spark is that, unlike MPI, it makes it easy to parallelize application code in a cluster. Futhermore, it makes efficient use of main memory to deal with intermediate data (RDD) that is used repeatedly, as with ML algorithms. In addition to deploying PDF computation over Spark, we propose three new methods to efficiently compute PDFs: data grouping, ML prediction and sampling. Data grouping consists in grouping similar points to compute the PDF. In the original input data, the data corresponding to some points may be the same or very similar to the data corresponding to a common point. ML prediction uses ML classification methods to predict the distribution type of each point. Sampling method enables to efficiently compute statistical parameters of a region by sampling a fraction of the total number of points to reduce the computation space.

To validate our solution, we use the spatial data generated from simulations based on the models from the seismic benchmark of the HPC4e project between Europe and Brazil for oil and gas exploration [2]. This benchmark includes MATLAB models for seismic wave propagation. In oil and gas exploration, seismic waves are sent deep into the Earth and allowed to bounce back. Geophysicists record the waves to learn about oil and gas reservoirs located beneath Earth's surface.

This paper makes the following contributions:

- A scalable approach and architecture to compute PDFs of QOIs in large spatial datasets using Spark;
- Three new methods to reduce the time of computing PDFs, *i.e.* data grouping, ML prediction and sampling;
- An extensive experimental evaluation based on the implementation of the methods in a Spark/HDFS cluster and big datasets ranging from 235 GB to 2.4 TB. The experimental

results show that our methods scale up very well and reduce the execution time by a factor of 33 (in the order of seconds or minutes) compared with a baseline method;

The three new methods, *i.e.* data grouping, ML prediction and sampling, are the main contributions of the paper. In addition, the layered architecture and the optimization methods (see details in Section 5.2) are also contributions for the basic processing of PDFs. The architecture is deployed in the execution environment and the optimization methods are directly implemented in a Scala program, thus facilitating the use of the three new methods.

The paper is organized as follows. Section 2 discusses related work. Section 3 introduces some background on Spark. Section 4 gives the problem definition. Section 5 presents our architecture for computing PDFs with Spark, with its main functions. Section 6 presents our solution to compute PDFs in parallel, with three methods, *i.e.* data grouping, ML prediction and sampling. Section 7 presents our experimental evaluation on different clusters with different data sizes, ranging from hundreds of GB to several TB. Section 8 concludes.

2 Related work

The importance of modeling data distribution through PDFs have been acknowledged in various domains such as seismology, finance, meteorology or civil engineering [11,47]. There is variability in almost any value that can be measured by observation and almost all measurements are made with some intrinsic error. Thus, simple numbers are often inadequate for describing a quantity, while PDFs are more appropriate. As the number of observations increases, efficient techniques to fit a set of observations to a PDF become paramount in practical applications [13,17,6]. For instance, the fitdistr function, which implements the Maximum Likelihood Estimation (MLE) (for normal, log-normal, geometric, exponential and Poisson distributions) and Nelder-Mead (NM) [35] (for the other distributions) methods in R [1], calculates the PDF with small error for a given PDF type. This function scans the input data once and the time complexity of both methods is $O(n)$, with n input observation values for the same point [42].

In this paper, we are interested in situations where large number of observations of the same quantity of interest are registered in thousands or even millions of spatial positions. This is the case, for instance, in numerical simulations that compute the values of QOIs through a set of points that discretizes a physical domain, *e.g.* representing the subsurface of the ocean in a seismic simulation. In this context, Campisano et al. [9] model series of observations at spatial positions in a grid as spatial-time series. The model helps supporting predictions on QOIs. This work is orthogonal to ours, as we strive to inform about the uncertainty exhibited by observations drawn from a continuous domain of values through a spatial region, which can be obtained by fitting the distribution to a PDF.

Another approach adopts the Generalized Lambda Distribution (GLD), a function for modeling a family of distributions [37], such as: normal,uniform, exponential, lognormal, gamma, etc., using λ parameters, represented as $GLD(\lambda_1, \lambda_2, \lambda_3, \lambda_4)$. The lambda parameters specify the location, scale and both sides of the tail of distributions, respectively, thus making it general enough to cover a variety of distributions. This generality of the GLD function is particularly interesting to avoid testing for a best fit among possible PDF function types. GLDs however cover a space of values in the λ_3 and λ_4 parameter space while being invalid in some other regions. Thus, when the distribution of a certain set of observation values is not covered by the space of λ_3 and λ_4, the GLD error can be very big. Therefore, for some distributions, the GLD approach may not offer a good fit, or even none at all. Moreover, there are a handfull of parameter estimation (*i.e.* data fitting) techniques, such as *Maximum log-likelihood* [5]. Theses techniques solve an optimization problem that starts from a random initial parametrization and numerically integrates the space

of probabilities minimizing, for example, the squared error, between the GLD and the set of observations. Therefore, for very large number of points (*i.e.* sets of observations) for which we may want to compute the corresponding GLD, specifying strategies that reduce the number of fitting computations is still needed and is the focus of this work.

In order to tackle the redundant PDF computation problem, we apply ML techniques. ML is now widely used to process big data [4,14] in activities such as regression analysis and classification [44,40]. It has been used to reduce the number of redundant computation by clustering data and reducing data dimensions to detect parameter dependent structures [8] or discover knowledge out of big data [21]. More recently, ML has been used as substitutes for traditional data structures in database systems, such as B+ tree indexing and caching [27].

Considering that fitting thousands or millions of sets of observations to PDF functions is a daunting task, we use ML to save total computation cost and reduce fitting elapsed time. In particular, we use the unsupervised k-means algorithm [33] to cluster similar sets of observations and the decision tree supervised algorithm to classify the distribution types using the mean and variance statistical moments of known observation sets with associated PDF types as training data [20].

3 Background on Spark

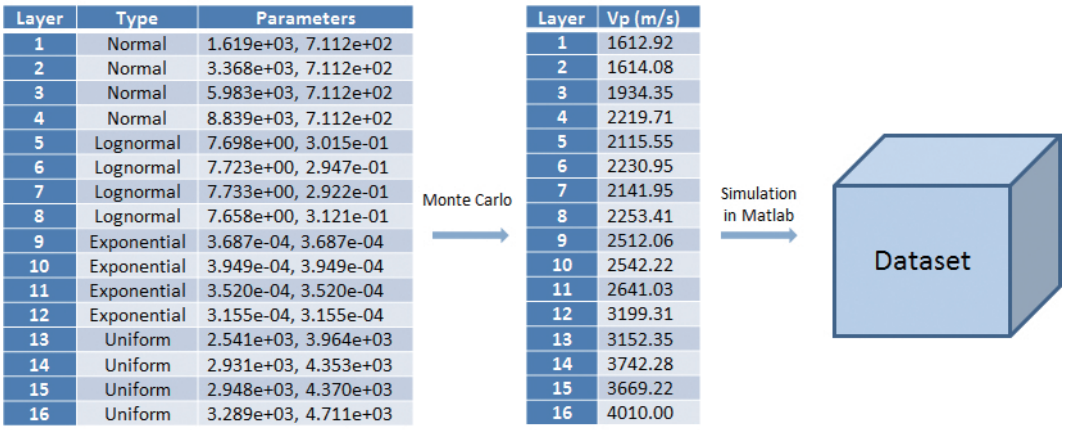
Spark [49] is an Apache open-source data processing framework. It extends the MapReduce model [16] for two important classes of analytics applications: iterative processing (machine learning, graph processing) and interactive data mining (with R, Excel or Python). Compared with MapReduce, it improves the ease of use with the Scala language (a functional extension of Java) and a rich set of operators (Map, Reduce, Filter, Join, Aggregate, Count, etc.). In Spark, there are two types of operations: transformations, which create a new dataset from an existing one (*e.g.* Map and Filter), and actions, which return a value to the user after running a computation on the dataset (*e.g.* Reduce and Count). Spark can be deployed on shared-nothing clusters, *i.e.* clusters of commodity computers with no sharing of either disk or memory among computers. In a Spark cluster, a master node is used to coordinate job execution while worker nodes are in charge of executing the parallel operations.

Spark provides an important abstraction, called resilient distributed dataset (RDD), which is a read-only and fault-tolerant collection of data elements (represented as key-value pairs) partitioned across the nodes of a shared-nothing cluster. RDDs can be created from disk-based resident data in files or intermediate data produced by transformations. They can also be made memory resident for efficient reuse across parallel operations.

Spark data can be stored in the Hadoop Distributed File System (HDFS) [41], a popular open source file system inspired by Google File System [22]. Like GFS, HDFS is a highly scalable, fault-tolerant file system for shared-nothing clusters. HDFS partitions files into large blocks, which are distributed and replicated on multiple nodes. An HDFS file can be represented as a Spark RDD and processed in parallel.

Spark provides a functional-style programming interface with various operations to execute a user-provided function in parallel on an RDD. We can distinguish between operations without shuffling, *e.g.* Map and Filter, and with shuffling, *e.g.* Reduce, Aggregate and Join. Shuffling is the process of redistributing the data produced by an operation, *e.g.* Map, so that it gets partitioned for the next operation to be done in parallel, *e.g.* Reduce. This process is complex and expensive and requires moving data across cluster nodes.

In this paper, we exploit Spark MLlib [3], a scalable machine learning (ML) library that can handle big data [28] in memory by exploiting Spark RDDs and Spark operations. In one method



Layer	Type	Parameters
1	Normal	1.619e+03, 7.112e+02
2	Normal	3.368e+03, 7.112e+02
3	Normal	5.983e+03, 7.112e+02
4	Normal	8.839e+03, 7.112e+02
5	Lognormal	7.698e+00, 3.015e-01
6	Lognormal	7.723e+00, 2.947e-01
7	Lognormal	7.733e+00, 2.922e-01
8	Lognormal	7.658e+00, 3.121e-01
9	Exponential	3.687e-04, 3.687e-04
10	Exponential	3.949e-04, 3.949e-04
11	Exponential	3.520e-04, 3.520e-04
12	Exponential	3.155e-04, 3.155e-04
13	Uniform	2.541e+03, 3.964e+03
14	Uniform	2.931e+03, 4.353e+03
15	Uniform	2.948e+03, 4.370e+03
16	Uniform	3.289e+03, 4.711e+03

Layer	Vp (m/s)
1	1612.92
2	1614.08
3	1934.35
4	2219.71
5	2115.55
6	2230.95
7	2141.95
8	2253.41
9	2512.06
10	2542.22
11	2641.03
12	3199.31
13	3152.35
14	3742.28
15	3669.22
16	4010.00

Fig. 2: **Data generation from simulation.** This process is repeated multiple times in order to generate multiple datasets. The values of Vp are different at different iterations. $Type$ represents the distribution type.

which we propose, we use ML techniques to classify the distribution types of the observation values at different points in order to reduce useless calculation.

4 Problem Definition

In this section, we first present how we generate the spatial datasets, which is the input for the problem. Then, we present the challenges to process the big input data. Afterwards, we present the problem in details.

4.1 Spatial Dataset Generation

A spatial dataset contains the information to generate an observation value matrix that corresponds to a three dimensional cube area. As a result of running multiple simulations, multiple spatial datasets are produced with a set of observation values at each point. The set of observation values at each point enables the computation of their mean value, standard deviation values and PDF.

Let us illustrate the problem with the spatial data generated from simulations based on the models from the seismic benchmark of the HPC4e project [2]. An important parameter of the models is wave phase velocity, noted Vp in electromagnetic theory, which is the rate at which the phase of the wave propagates in space. The models contain 16 layers and each layer is associated to a value of Vp. The top layer delineates the topography and contains the description information of the other 15 layers. Each of the 15 layers is used to generate the observation values of points in a horizontal space of the cube area. A set of 16 layers can generate a spatial dataset, where each point corresponds to an observation value. Then, N sets of 16 layers can generate N spatial datasets, where each point corresponds to a set of N observation values.

Since our purpose is to study the uncertainty in the output as a result of the wave propagation of the input uncertainty through the model, we assume that the input value of each layer is uncertain and obeys a PDF. The distribution types for every four layers are: Normal (for the

first four layers), Log-normal (for the fourth to eighth layers), Exponential (for the ninth to twelfth layers) and Uniform (for the rest). We use a Monte Carlo method to generate different sets of the 16 input parameters. For each set of input parameters, we generate a spatial dataset using the models as shown in Figure 2. In Figure 2, we use the Monte Carlo method to generate a set of 16 variables, each of which corresponds to the Vp of each layer. Then, we execute the MATLAB programs of the seismic benchmark to generate a spatial dataset. Finally, we repeat this process multiple times in order to generate multiple datasets.

4.2 Challenges

Computing PDFs on spatial datasets raises three main challenges, *i.e.* big data, long and complex computation and error bound.

First, as the input spatial datasets can be very big, a simple scan of the input data is not enough. After scanning the data, we need to perform a join operation to merge the observation values of each point with a set of observation values. This join operation may take much time when there are many points. Since the data corresponding to the same point are distributed at different nodes in a cluster, the join operation requires shuffling, which may take much time. In addition, when the input data is very big, we cannot cache all the data in memory.

Second, computing PDFs is complex and takes time for big spatial datasets. In order to calculate the PDF of a point, we use the fitdistr function, which implements MLE and NM method (a commonly used and robust numerical method) in an R program. The fitdistr function optimizes a cost function using multiple iterations, which may take much time. In addition, this function is only used for calculating the PDF of a single point, which takes a vector of observation values as input and calculates the PDF of a specific distribution type. Furthermore, this function is designed for execution in a single machine, and is not scalable for a cluster environment. In our problem, we need to first extract the observation values of each point from a spatial dataset. Then, for each point, we must calculate the PDF of an appropriate type, which corresponds to the minimum error. Finally, the results must be stored in a file system. This complex process can take much time with a basic approach. For instance, with a cluster of 6 nodes, each with 32 CPU cores, it takes about 63 days using the baseline method (see details in Section 6.1) and the fitdistr function in R language to compute the PDFs of all the points in all the slices of a cube with a dataset of 2.4TB. Thus, computing PDFs is complex and takes time for big spatial datasets.

Third, the PDF computation should not have unacceptable error, *i.e.* under error bound. For instance, if the error is bigger than 50%, then the result will not be representative. This requirement implies choosing a distribution type that corresponds to the minimum error. The fitdistr function in the R program can only fit the parameters corresponding to a specific distribution type to calculate the PDF based on a set of observation values. But it cannot choose a distribution type.

4.3 Problem Formulation

The main problem we address is to efficiently compute PDFs on big spatial datasets, such as the seismic simulation data discussed above. Since it takes much time to compute the PDFs of the points in the whole cube area, our approach is to divide it into multiple spatial regions and compute the PDFs of all the points in a chosen region within a reasonable time. The spatial region is chosen based on some statistical parameters of the region. Thus, the main problem is how to efficiently calculate the PDF of each point in a spatial region, *e.g.* a horizontal slice in the cube

area, with a small average error between the PDF and the distribution of the observation values. A slice is a set of points that correspond to the same position in one of the three dimensions. In order to choose a region, the mean and standard deviation values and the distribution type of a part of the points in the spatial region should be computed. In addition, the average mean value, the average standard deviation value and the percentage of points corresponding to each distribution type in all the points of the region can also be computed. We denote these statistical parameters of a spatial region to compute by the features of a spatial region. A related subproblem is how to efficiently calculate the features of a spatial region. In this paper, we take a horizontal slice as a spatial region.

The input data is the set of spatial datasets, which are generated from the simulation as explained in Section 4.1. Let DS be a set of spatial datasets, d be a spatial dataset in DS and N be the number of points in a slice (the same for Equations 3, 4 and 6). d is generated using a set of Vp of the 16th layers (each set of Vp is different). As shown in Figure 3, each d is composed of a matrix of observation values corresponding to the sampled points in the cube area. X, Y, I represent the first, second and third dimensions of the cube area and the index of observation values in the dataset. In the spatial data domain, the first dimension (x) is inline; the second dimension (y) is crossline and the third dimension (i) is time or depth. A dataset is composed of multiple slices, and each slice is composed of multiple lines. Each slice corresponds to the same position in the third dimension (I), and each line corresponds to the same position in the second (Y) and third (I) dimensions. In this paper, we take a slice (*i.e.* Slice 201 in Section 7) as the region to calculate PDFs. Each point ($p_{x,y,i}$) is represented as an observation value in the dataset and the indices are x, y and i in each dimension. Based on multiple datasets, each point $p_{x,y}$ in Slice i corresponds to a set of values $V = \{v_1, v_2, ..., v_n\}$ while v_k is the observation value corresponding to the point $p_{x,y}$ in Slice i in $d_k \in DS$ and n is the number of observation values in the set (the same for Equations 1, 2 and 5). d_k represents the kth dataset (d) in DS.

Based on these notations, we define Equations 1 - 4 and 6, which are based on the formulas in [18]. The mean ($\mu_{x,y}$) and standard deviation ($\sigma_{x,y}$) values of a point can be calculated according to Equations 1 and 2, respectively. And the average mean ($\overline{\mu_i}$) and standard deviation ($\overline{\sigma_i}$) values of Slice i can be calculated according to Equations 3 and 4, respectively. The error $e_{x,y,i}$ between the PDF F and the set of observation values V can be calculated according to Equation 5, which compares the probability of the values in different intervals in V and the probability computed according to the PDF. The intervals are obtained by evenly splitting the space between the maximum value and the minimum value in V. min is the minimum value in V, max is the maximum value in V and L represents the number of all considered intervals, which can be configured. $Freq_k$ represents the number of values in V that are in the kth interval. The integral of $PDF(x)$ computes the probability according to the PDF in the kth interval. Equation 5 is inspired by the Kolmogorov-Smirnov Test [30], which tests whether a PDF is adequate for a dataset. In addition, we assume that the probability of the values outside the space between the maximum value and the minimum value is negligible for this equation. Then, the average error E of Slice i can be calculated according to Equation 6.

$$\mu_{x,y} = \frac{\sum_{i=1}^{n} v_i}{n} \tag{1}$$

$$\sigma_{x,y} = \sqrt{\frac{\sum_{i=1}^{n} (v_i - \mu)^2}{n-1}} \tag{2}$$

$$\overline{\mu_i} = \frac{\sum_{p_{x,y} \in slice_i} \mu_{x,y}}{N} \tag{3}$$

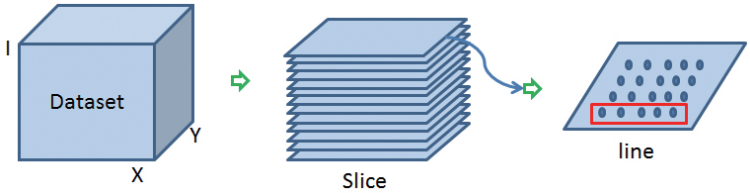


Fig. 3: **Dataset structure.** A dataset is composed of multiple slices. Each slice is composed of multiple lines. The red rectangle represents a line. Each circle represents a point, which contains an observation value of the point in the original cube area.

$$\overline{\sigma_i} = \frac{\sum_{p_{x,y} \in slice_i} \sigma_{x,y}}{N} \tag{4}$$

$$e_{x,y,i} = \sum_{k=1}^{L} \mid \frac{Freq_k}{n} - \int_{min+(max-min)*\frac{k-1}{L}}^{min+(max-min)*\frac{k}{L}} PDF(x)dx \mid \tag{5}$$

$$E = \frac{\sum_{p_{x,y} \in slice_i} e_{x,y,i}}{N} \tag{6}$$

We can now express the main problem as follows: given a set of spatial datasets $DS = \{d_1, d_2, ..., d_n\}$ corresponding to the same spatial cube area $C = \{slice_1, slice_2, ..., slice_j\}$, how to efficiently calculate the mean, standard deviation values and the PDF F at each point in $slice_i \in C$ with a small average error E not higher than a predefined average error ε. In addition, we also need to compute the statistical parameters of slices in order to choose Slice i mentioned in the first subproblem. Thus, the related subproblem can be expressed as: how to efficiently calculate the features of a slice when given the same datasets DS. The features are:

- μ, σ and distribution type of some points in the slice
- $\overline{\mu_i}$, $\overline{\sigma_i}$ and the percentage of points for each distribution type in all the points of the slice

5 Architecture for Computing PDFs

In this section, we describe the architecture for PDF computation (see Figure 4). This architecture has four layers, *i.e.* infrastructure, basic process to compute PDFs, memory management and methods to compute PDFs. The higher layers take advantage of the lower layers' services to implement their functionality. The infrastructure layer provides the basic execution environment, including Spark, HDFS and Network File System (NFS) in a cluster. The basic processing layer provides guiding principles to load the big spatial data and to compute PDFs. The memory management layer allows optimizing the execution of the basic process by caching data and managing tumbling windows on big data. The methods to compute PDFs are presented in Section 6.

The architecture has four layers and decomposes the problem in smaller problems, which can be solved independently and efficiently. In each layer, we use different optimization methods, *e.g.* data caching, window size ajustement, parallel data loading, to speed up execution. As a result, the baseline approach (see details in Section 6.1) also benefits from our architecture, which explains why it yields relatively good performance and good scalability.

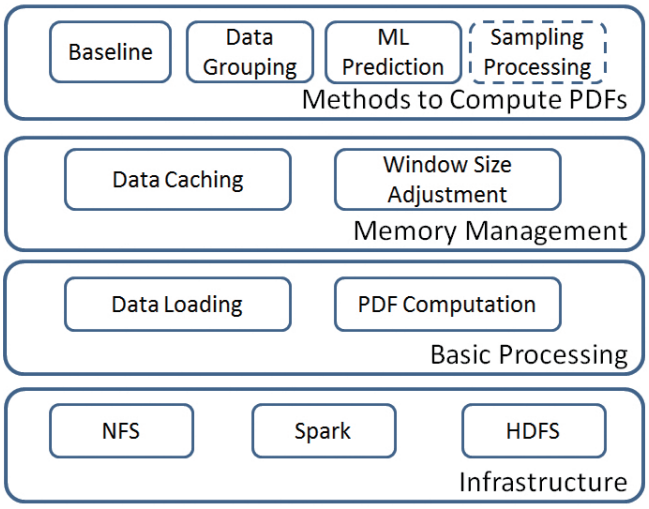


Fig. 4: **Architecture for Computing PDFs.**

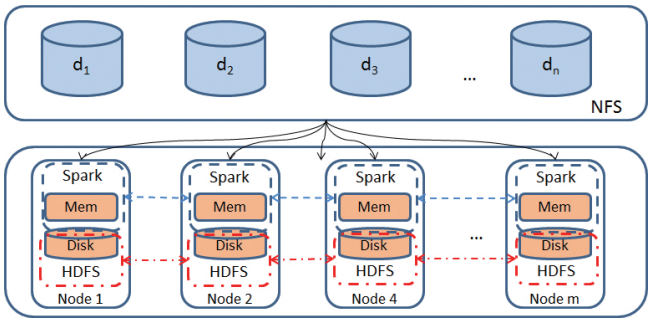


Fig. 5: **Infrastructure.** d_i represents the ith spatial dataset.

5.1 Infrastructure with Spark

Figure 5 illustrates the infrastructure we deploy to process big spatial data. The big spatial data is produced by simulation application programs and stored in NFS [39], a shared-disk file system that is popular in scientific applications. Spark and HDFS are deployed over the nodes of the cluster. The intermediate data produced during PDF computation and the output data are stored in HDFS, which provides persistence and fault-tolerance.

Keeping the input spatial data in NFS allows us to maximize the use of the cluster resources (disk, memory and CPU), which can be fully dedicated for PDF computation. With the input data stored in NFS, the NFS server is outside the Spark/HDFS cluster and takes care of file management services, including transferring the data that is read to the cluster nodes. An alternative solution would have been to store the input data in HDFS, which would lead to have HDFS tasks competing with Spark tasks for resource usage on the same cluster nodes. We did try this solution and it is much less efficient in terms of data transfer between cluste nodes. This is because HDFS is more complex and does more work due to fault-tolerance and data replication. Note that we use NFS only for data loading (see Algorithm 2), while the data generated from execution is persisted in HDFS.

We use Spark as our execution environment for computing PDFs. We developed a program written in Scala, which realizes the functionality of different methods to compute PDFs. Once the method to compute PDFs is chosen, the Scala program is executed as a job within Spark.

5.2 Principles for the Basic Processing of PDFs

The basic processing of PDFs consists of data loading, from NFS to Spark RDDs, followed by PDF computation using Spark. The data loading process treats the data corresponding to a slice and pre-processes it, *i.e.* calculates statistical parameters of observation values of each point and relates the observation values of each point to an identification of the point. The identification of each point is an integer value which represents the location of the point in the cube area. Then, the PDF computation process groups the data and calculates the PDFs and errors of all the points in a slice based on the pre-processed data. To make the basic processing of PDFs efficient, we use the following guiding principles.

1. **Parallel data loading.** To perform data loading in parallel, we store the identifications of points in an RDD, which is evenly distributed on multiple cluster nodes. For each point in a node, all the corresponding values in different spatial datasets are retrieved from NFS. At the same time, the mean and standard deviation values are calculated. Then, the identification of the point is stored as the key of the point. The mean and standard deviation values and the observations values are stored as the value of the point. The key and value of the point are stored as a key-value pair in the RDD. This loading process is realized by a Map operation in Spark, which is fully parallel.
2. **Data grouping.** In order to avoid repeating the PDF computation for the same or similar sets of observation values, we group the data of different points that shares similar statistical features, *e.g.* the mean and standard values. Then, a representative point of the group is chosen. The PDF of the representative point is taken to represent all the points in the group. Thus, the PDF computation of all the points in the group is reduced to the computation of the representative point. The grouping can be realized using an *Aggregation* operation in Spark. However, the shuffling process in the Aggregation operation may take much time. When this occurs, then we can simply avoid data grouping. To decide whether to use data grouping, the ideal solution would be to have a cost model, but it is difficult to estimate the time to transfer data within Spark. However, we can test if data grouping is effective on small workloads, *e.g.* a few lines of points in a slice as explained in Section 7.
   After data grouping, the set of the identifications of the points in the group is stored as a part of the value in the key-value pair. For each representative point, the key represents the identification while the value contains the mean and standard deviation values and the observation values.
3. **Parallel processing of PDFs.** The PDF of each point (or representative point) is computed based on its observation values. This process is also realized in a Map operation, which distributes the key-value pairs in RDD of different points to different nodes. There are two methods to calculate the PDF of a point, *i.e.* with ML and without ML. With ML, the distribution type corresponding of a point is predicted (see Section 6.3 for more details), and then the PDF and error, *i.e.* PDF error, are computed. Without ML, the PDFs of different distribution types are computed, and the PDF of the minimum error is chosen as the PDF of the point. The PDF computation of different points is executed in parallel in different nodes of the cluster. After the parallel processing of PDFs, the key remains the identification of each point (representative point) while the PDF is stored as a part of the value in the key-value pair of the RDD. In addition, since the mean and standard deviation values and

the observation values are no longer useful, the corresponding data is removed from the value in the key-value pair. Finally, the PDF of each point is persisted in a file or database system for future use. In addition, an average error of the PDF of the points in the slice is calculated and shown as the result of executing the Scala program.

4. **Tumbling window.** Since a slice can have many points that won't fit in memory, we use a tumbling window over the slice during data loading, data grouping and parallel processing of PDFs. A window represents a set of points to process, which corresponds to a swath area of several continuous lines in the slice to process. Any two windows have no intersection. After the processing of one window, the process of the next window begins until the end of the slice. Once the size of the window is configured, it stays the same during execution. The size of the window has strong impact on execution and must be chosen carefully (see details in Section 5.3.2). Note that the tumbling window is different from the data chunk in Spark. Data chunk is generated by splitting the data in RDD, which is stored in each node of the cluster node. The window here is a unit of points to process by all the nodes in the cluster.
5. **Use of external programs.** In the parallel data loading and parallel processing of PDFs, we use external programs, which do the specific loading of the points. Since some Java functions, *e.g. skipBytes* (Skips and discards a specified number of bytes in the current file input stream), may not work correctly during the parallel execution of a Map operation in Spark [23], we call an external Java program in the Map operation to retrieve observation values of a point from different spatial datasets and pre-process values. Since the PDF computation is implemented by an external program (in R), we call it within the Map operation for the parallel processing of PDFs. Finally, the output data of the external program is transformed to key-value pairs and stored in RDDs by the same Map operation, which executes the external program.

Figure 6 shows the data flow of the process. First, the data is loaded from NFS in parallel. The data is stored in an RDD; the key is the identification (*id*) of each point and the value is the set of the mean and standard deviation value and the observation values of each point. The *id* of each point is calculated based on its indices in the three dimensions. If we apply data grouping, the key value pairs are processed in three steps. First, we take the mean and standard deviation value as the key and put the *id* and the set of observation values of each point as value. Then, the RDD is grouped based on the key, *i.e.* the mean and standard deviation values. The set of *id* of the points of the same group is stored in the value. In addition, we store the first set of the observation values in the value. Then, we take the set of *id* as the key, and the mean value, the standard deviation value and the set of observation values as the value. If we do not apply data grouping, we have only one *id* in the key. We calculate the PDF based on the set of observation values. Finally, we have the set of *id* as key and the PDF type and the PDF parameters as value, which is stored in HDFS.

5.3 Memory Management

In order to efficiently compute PDFs, we use two memory management techniques to optimize the calculation of PDFs over big data: data caching and window size adjustment.

5.3.1 Data Caching

We use data caching, *i.e.* keeping data in main memory, to reduce disk accesses. To identify which data to cache, we distinguish between four kinds of data: input data, instruction data, intermediate data and output data. The input data is the original data to be processed, *i.e.* the

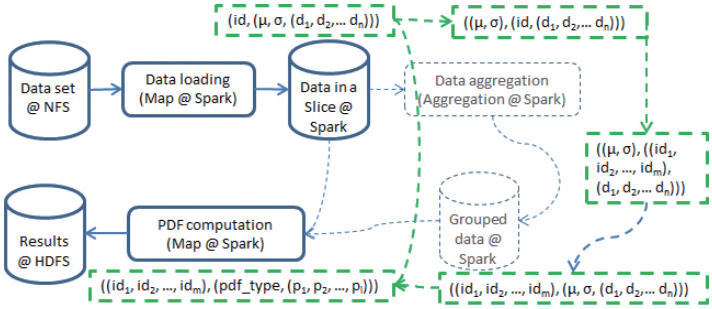


Fig. 6: **Data flow and key value pairs.** The blue boxes and arrows represent the data flow. @Spark represents that the data is stored in a Spark RDD. The green boxes and arrows represent key value pairs corresponding to the data flow. *id* represents the id of each point. A set of d_1 to d_n represents the observation values. μ and σ represent the mean and standard derivation value of the set of observation values. The set of id_1 to id_m represents the ids of all the points in the group after data grouping. If data grouping is not used, there is only one *id* as the key of the key value pair in the result of the PDF computation. *pdf_type* represents the type of PDF. The set of p_1 to p_l represents the parameters of the calculated PDF.

big spatial datasets. The instruction data is the data corresponding to the external programs, *e.g.* Java or R programs used in data loading and PDF computation. The intermediate data is the data generated by the data loading process or the execution of external programs, and used by subsequent execution. The output data is the final data generated by the PDF computation.

We use a simple caching strategy. We do not cache input data because it can be very big and read only once. We only cache instruction data and intermediate data, which are accessed much during execution. However, intermediate data that is not used in subsequent operations is removed from main memory. The output data is written to memory first and then persisted.

During execution, some intermediate data that is stored in Spark RDDs can be cached using the Spark *Cache* operation, which stores RDD data in main memory. However, the instruction data of external programs and the intermediate data directly generated by executing these external programs are outside of RDDs. We cache this data in temporary files in memory, using a memory-based file system [43]. Then, the information in the cached files is retrieved and stored as intermediate data in RDDs, which can be again cached using the *Cache* operation of Spark.

5.3.2 Window Size Adjustment

The size of the tumbling window is critical for efficient PDF computation. When it is too small, the degree of parallel execution among multiple cluster nodes is low. Increasing the window size increases the degree of parallelism, but may introduce some overhead in terms of data transfers among nodes or management of concurrent tasks. Thus, it is important to find an optimal window size in order to make a good trade-off between the degree of parallelism and overhead.

To find the optimal window size, we distinguish between the data loading process and the PDF computation process, which have different data access patterns. In data loading, the processing of each point can be done independently by a Map operation in parallel. Thus, the overhead of data transfers and concurrent task management with a big window size is small. Thus, we can choose a window size that ensures that there is enough work to do for each node and that multiple nodes can be used. For instance, consider a cluster with n nodes, each node with c CPU cores. Assuming that the data loading for each point can occupy a CPU core, we can choose a

maximum window size corresponding to $n*c$ points. If the number of points is less than $n*c$, we choose the maximum number of points as the window size.

During PDF computation, the overhead of having a big window can be high since the processing of different points are not independent, in particular when we use data grouping. For instance, when the window size is bigger, the data of more points are present at each node. In order to group data, the data of each point need to be compared with the data of more points, which takes more time. In addition, there is much more data transferred among different nodes in the data shuffling process of data grouping. Furthermore, since the PDF computation takes much time, the management of concurrent tasks within each node also increases execution time for a big window. As a result, the overhead of having a big window becomes high for PDF computation.

To find an optimal window size, we test the Scala program on a small workload (with a small number of points) with different window sizes, and then use the optimal size for the PDF computation of all the points in the slice.

6 Methods to Compute PDFs

In this section, we present our methods to compute PDFs efficiently. First, we introduce a baseline method, which will be useful for comparison. Then, we propose two methods, *i.e.* data grouping and ML prediction, to compute PDFs, which addresses the main problem defined in Section 4. Finally, we propose a sampling method to calculate the features of a slice, which addresses the related problem.

6.1 **Baseline Method**

The baseline method computes the PDF of each point in a slice as follows (see Algorithm 1). Line 2 loads the spatial datasets and calculates the mean and standard deviation values of each point by using Algorithm 2. Lines 3 - 13 compute the PDF for all the points in the ith slice. Line 5 gets a window in the slice. Line 6 selects all the points in the window to process. The implementation of the select function is different for data grouping (see details in Section 6.2). For each point in the window, Line 8 gets the set of corresponding observation values, which can be automatically realized by Spark as the $RawData$ is stored in RDD with the id of a point as key and the set of observation values as value. Line 9 computes the PDF based on the observation values and the error between the PDF and the observation values. This can be achieved by executing an R program. The loop of Lines 8-10 can be executed in parallel using the Map operation in Spark. The $ComputePDF\&Error$ function is realized by Algorithm 3. The data is persisted in the storage resources (Line 12) and the average error E is calculated (Line 15). Lines 3-15 correspond to the PDF computation process.

Algorithm 2 loads and preprocesses the spatial data. Line 4 chooses a window to load the data. Then, for each point in the window, the data in each dataset of DS is loaded (Lines 6-10). This process is realized in a Map function in Spark. A Java program is called in the Map function to read the data at a specific position instead of loading all the data. Then, the mean and standard deviation values are calculated (Lines 11 and 12). Finally, the loaded data is cached in memory in a Spark RDD (Line 16).

Algorithm 3 computes the PDF of a point with the smallest error in a set of candidate distribution types. Line 3 calculates the statistical parameters of PDFs based on different distribution types in a set of distribution candidates $Types$. For instance, the parameters for Normal are mean and standard deviation values while the parameter for exponential is rate. The more types are

Algorithm 1 PDF computation

Input: DS: a set of spatial datasets corresponding to a spatial cube area; i: the ith slice to analyze; $Types$: a set of distribution types
Output: PDF: the PDF of all the points in the ith slice of the cube area; E: the average error between the PDF and the observation values of all the points in the ith slice

```
1: PDF ← ∅
2: RawData ← loadData(DS, i)
3: while not all points in slice_i are processed do
4:     pdfs ← ∅
5:     window ← GetNextWindow(slice_i)
6:     Points ← Select(window, RawData)
7:     for each p ∈ Points do
8:         d ← getObservationValues(p, RawData)
9:         (pdf, error) ← ComputePDF&Error(d, Types)
10:        pdfs ← pdf ∪ pdfs
11:    end for
12:    persist(pdfs)
13:    PDF ← PDF ∪ pdfs
14: end while
15: E ← Average(error)
end
```

Algorithm 2 Data loading (loadData)

Input: DS: a set of datasets corresponding to a spatial cube area; i: the ith slice to analyze
Output: $RawData$: mean, standard deviation and the original dataset of each point in the cube area

```
1: RawData ← ∅
2: while RawData does not contain all the points in slice_i do
3:     windowData ← ∅
4:     window ← GetNextWindow(slice_i)
5:     for each p ∈ window do
6:         rd ← ∅
7:         for each ds ∈ DS do
8:             data ← GetData(ds, p, i)
9:             rd ← data ∪ rd
10:        end for
11:        μ ← ComputeMean(rd)
12:        σ ← ComputeStd(rd)
13:        rd ← μ ∪ σ ∪ rd
14:        windowData ← rd ∪ windowData
15:    end for
16:    Cache(windowData)
17:    RawData ← windowData ∪ RawData
18: end while
end
```

considered in $Types$, the longer the execution time of Algorithm 3 is. Then, the corresponding error of the PDF based on *type* and *parameters* are calculated using Equation 5. Finally, the PDF that incurs the smallest error is chosen (Line 7).

We exploit Algorithms 1 and 2 in both the data grouping and ML prediction methods. However, the *Select* and $ComputePDF\&Error$ functions are different in different methods.

6.2 Data Grouping

As some positions in the cube area may share the same physical properties, the corresponding points may have the same distribution of observation values. Thus, using the data grouping

Algorithm 3 PDF computing (ComputePDF&Error)

Input: d: a set of observation values for a point; $Types$: a set of distribution types
Output: PDF: the PDF of the point; $error$: the error between the PDF defined by the distribution type and the statistical parameters and the observation values of the point

```
results ← ∅
for each type ∈ Types do
    parameters ← fitDistribution(d, type)        ▷ fitDistribution implemented by MLE or NM
    pError ← CalculateError(type, parameters, d)        ▷ According to Equation 5
    results ← {(type, parameter, pError)} ∪ results
end for
(PDF, error) ← GetSmallestError(results)
end
```

method, we can execute the PDF computation process only once and use the result to represent all the corresponding points. In this method, the *Select* function in Algorithm 1 has two steps. First, the points with exactly the same mean and standard deviation values are aggregated into a group. The grouping can be realized by the *aggregation* operation in Spark. Then, one point in each group is selected to represent the group of points. Then, the data corresponding to selected points are processed to compute the PDFs.

We randomly select one point in an aggregated group. We assume that the points that share the same mean and standard deviation have the same PDF although their observation values can be different. With this assumption, the selection of the point in an aggregated group does not have an effect on the computed PDF of the group. There are scenarios where the mean and standard deviations share the same values, respectively, but the PDFs are different. In this situation, we can take into consideration other normalized moments [1], *e.g.* 3rd, 4th etc. However, it may take additional time to calculate other normalized moments. Thus, we only consider the mean and standard deviations. We will experiment with datasets where the points of the same set of mean and standard deviation values have the same PDFs.

In some cases, although some points may share the same PDF, the observation values of the points are slightly different. And the mean and standard values may be different by a very small fluctuation. In this case, we can cluster the points that have similar mean and standard deviation values with an acceptable error. For instance, we can use k-means [33], which is already implemented in parallel within Spark MLlib. However, the implementation of point clustering is out of the scope of this paper and we leave it as future work.

When the number of nodes in the cluster is small and the window size is small, there is few data to be transferred for the grouping. And the grouping method can avoid repeated execution on the same set of observation values. However, when the number of nodes is high or the window size is big, there is much more data to be transferred among different nodes, which may take much time. In some situations, the time to transfer the data may be longer than the time to run the repeated execution. In addition, if a point corresponds to big amounts of data (many observation values), even though the window size is small, the shuffling process of data grouping may also take much time. In this case, the data grouping method is not efficient.

[1] The nth moment is defined as:

$$\mu_n = \sum_{i=1}^{m} (x_i - \mu)^n \tag{7}$$

where m is the number of values in the dataset, x_i is the ith value in the dataset and μ is the mean value of the dataset.

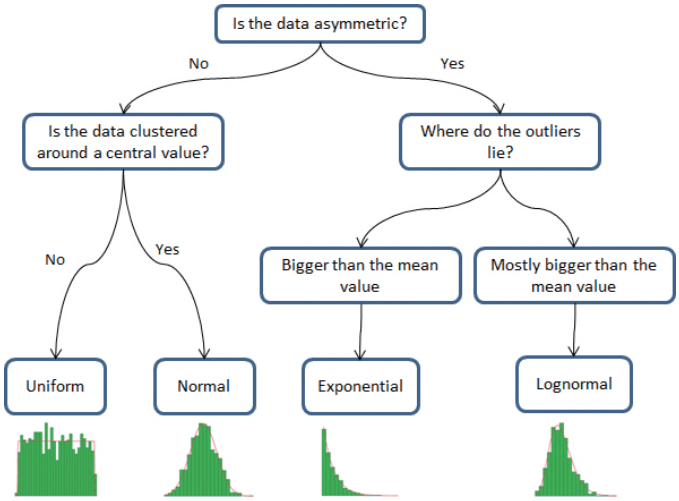


Fig. 7: A decision tree to choose the distribution type for a set of data.

6.2.1 **Reuse Optimization**

When there are points of the same mean and standard deviation values in different windows, we can reuse the existing calculated results in order to avoid new executions. Thus, we propose the reuse optimization method, which not only aggregates the data to groups but also checks if there are already existing results, *i.e.* the PDFs for the point of the same mean and standard deviation values, generated from the previous execution of processed windows. We expect this method to be better than data grouping only. However, it may take time to store all the calculated results and to search existing PDFs from a large list of previously generated results. This extra time may be longer than the reduced time, *i.e.* the time to compute the PDF on the dataset (the execution time of Algorithm 3).

6.3 **ML Prediction**

In this section, we propose an ML prediction method based on a decision tree to compute PDFs. Decision tree classifier [38] is a typical ML technique to classify an object into different categories. The basic idea is to break up a complex decision into a union of several simpler decisions, hoping the final solution obtained this way would resemble the intended desired solution. The decision tree can be described as a tree selected from a lattice [7]. A tree $G = (V, E)$ consists of a finite, nonempty set of nodes V and a set of edges E. A path in a tree is a sequence of edges. The depth of the graph in a tree is the length of the largest path from the root to a leaf. A leaf is the node that has no proper descendant.

The data to be used to generate a decision tree is training data. In order to generate a decision tree, the training data can be split into different datasets (bins) according to different features and each dataset is a unit to be processed. The maximum number of bins is defined in order to reduce the time to generate a decision tree. Figure 7 shows a decision tree of depth 3 to choose a distribution type for a dataset. However, in this paper, we take some statistical features as parameters to choose a distribution as explained below.

In Algorithm 3, the execution of Lines 3-5 is repeated several times (the number of distribution type candidates), which is very inefficient. We assume that we can learn the correlation among

statistical features, *e.g.* mean and standard deviation values, and the distribution type. Then, we can directly predict the distribution type based on the relationship and use the predicted distribution type to execute Lines 3-5 in Algorithm 3 once for each point.

We assume that we have some previously generated output data, which contains the results of several points, *i.e.* the mean and standard deviation values and the type of distribution. Before execution of our method, we can generate a ML model (that correlates between statistical features and distribution types), *i.e.* decision tree, based on the previous existing data. Then, we use Algorithm 4 to replace Algorithm 3.

When the points in the current slice have the same correlation between the statistical features and the distribution types as that in the previously generated output data, we can use ML prediction. Generally, the points in different slices but corresponding to the same spatial dataset have the same correlation. Thus, we can use the output data generated based on some points in one slice, *e.g.* Slice 0, to generate decision tree model and use the model to calculate PDFs of the points in other slices, *e.g.* Slice 201. This correlation information is not based on the fact that various points present the same statistical features, *e.g.* PDF, although this fact is general since the positions in the cube area related to the points may share similar physical properties. The correlation is related to each set of datasets and corresponds to a cube area. In addition, we can find that the correlation information in Slice 0 can always be used in our targeted slice with a few acceptable extra errors as shown in Section 7.

Algorithm 4 PDF computing based on ML prediction

Input: d: a vector of observation values for a point; $model$: the decision tree; μ: the mean value of d; σ: the standard deviation value of d

Output: T: the distribution type of the point; P: the parameters of the distribution; $error$: the error between the PDF defined by the distribution type and the parameters of the distribution and the real distribution of data in d

1: $T \leftarrow predict(model, \mu, \sigma)$
2: $P \leftarrow fitDistribution(d, T)$
3: $error \leftarrow CalculateError(T, P, d)$ ▷ According to Equation 5
end

6.3.1 ML Model Generation

In order to generate the decision tree model, there are some hyperparameters to determine, *e.g.* the depth of the tree ($depth$) and the maximum number of bins ($maxBins$). In order to tune the hyperparameters, we randomly split the previously generated output data into two datasets, *i.e.* training set and validation set. We use the training set to train the model with different combinations of hyperparameters. Then, we test the generated models on the validation set in order to choose an optimal combination of hyperparameters. We take the wrong prediction rate in the validation set as the model error while tuning the hyperparameters. The model error represents the preciseness of the decision tree model.

The model error decreases at the beginning and then increases, because of overfitting [48], as the values of $depth$ and $maxBins$ increase. While the time to train the model becomes longer when the values of $depth$ and $maxBins$ increase. We choose the minimum values of $depth$ and $maxBins$, from which the error does not decrease when they ($depth$ or $maxBin$) increase. The process to tune the hyperparameters can take much time. We assume that the points in different slices have the same correlation between the statistical features and distribution types. We also assume that the hyperparameters can be shared to train the models based on different previously

generated output data in order to avoid tuning the hyperparameters and to reduce the time to train the decision tree model. Thus, the chosen values of *depth* and *maxBins* are used as fixed hyperparameters to generate the decision tree model for different previously generated output data.

Within the previously generated output data, each point has mean and standard deviation values and the distribution type. Since we have fixed hyperparameters, we do not need to use a validation dataset to tune the hyperparameters. Thus, in order to generate the model, we randomly partition the previously generated output dataset into two parts, *i.e.* training set and test set. In the training processing, we train the model using the training set. The input of the trained model is the mean and standard values and the output is a predicted distribution type. After generating the model, we take the wrong prediction rate in the test set as the model error while training models. During the PDF computation process, the generated model is broadcast to all the nodes, which reduces communication cost.

There are scenarios in which the mean and standard deviation share the same values, respectively, but the distribution type is different. In this situation, we can take into consideration other normalized moments as explained in Section 6.2. We will experiment with datasets where the points of the same set of mean and standard deviation values have the same distribution types.

6.4 **Complexity Analysis**

Since the ML approach optimizes the $ComputePDF\&Error$ function, it can be combined with other methods as shown in Table 1. Thus, we call the pure adoption of ML: ML or baseline + ML; the combination of data grouping and ML: data grouping + ML or grouping + ML; the combination of reuse and ML: reuse + ML. Grouping + ML first uses the data grouping methods to group points and then use ML prediction to calculate PDF and error for each representative point. Reuse + ML first groups the points using the data grouping method and then searches the PDF corresponding to the set of the mean and standard deviation values of each representative point in the results generated by previous execution. If there is no results for the set of mean and standard deviation values, it uses the ML method to calculate the PDF and the error.

Name	Configuration	Select Function in Algorithm 1	Algorithms	Reuse
Baseline	No data grouping or ML	Select all the points	1, 3	No
Grouping	Data grouping without ML	Select one point per group	1, 3	No
Reuse	Reuse without ML	Select one point per group	1, 3	Yes
ML	Baseline with ML	Select all the points	1, 4	No
Grouping + ML	Data grouping with ML	Select one point per group	1, 4	No
Reuse + ML	Reuse with ML	Select one point per group	1, 4	Yes

Table 1: **Configuration of different methods.** Reuse represents that the process tries to reuse the calculated PDFs corresponding to the same mean and standard deviation values in other windows (see details in Section 6.2.1).

Table 2 shows the computation complexity and the communication complexity of different combinations of methods. Let α (the same as that in Table 2) represent the average ratio between the number of points that have different sets of mean and standard values and the total number of points in a window. β (the same as that in Table 2) represents the average ratio between the number of points that have different sets of mean and standard values and the number of all points in a slice. The "average" in α and β is calculated in terms of the whole data set. Thus, β is smaller

than α when there are points that have the same set of mean and standard deviation values [1]. WS represents the window size. When $WS * \alpha$ is bigger than 1, the computation complexity of Grouping is bigger than Baseline. In this case, we ignore data grouping. When $N * WS * \beta$ is bigger than 1, the computation complexity of Reuse is bigger than that of Baseline. When N increases, the computation complexity of Reuse increases much faster than Baseline because of N^2. The communication complexity of Grouping and Reuse is bigger than that of Baseline. When the number of nodes in a cluster g or the number of observation values for each point get big, the communication complexity of Grouping or Reuse increases linearly, which corresponds to longer execution time. When combining with ML, Grouping and Reuse have the same features, *i.e.* bigger computation complexity when $WS * \alpha$ ($N * WS * \beta$ for Reuse) is bigger than 1 and bigger communication complexity. The computation complexity of ML is smaller than that of Baseline, which corresponds to smaller execution time as shown in Section 7. In addition, ML has the same communication complexity as that of Baseline.

Name	Computation complexity	Communication complexity
Baseline	$O(N * M * L)$	$O(N * M * L)$
Grouping	$O(N * WS * M * L * \alpha)$	$O(N * M * L + N * K * D * L)$
Reuse	$O(N^2 * WS * M * L * \beta)$	$O(N * M * L + N * K * M * L)$
ML	$O(N * M)$	$O(N * M * L)$
Grouping + ML	$O(N * M * WS * \alpha)$	$O(N * M * L + N * K * M * L)$
Reuse + ML	$O(N^2 * M * WS * \beta)$	$O(N * M * L + N * K * M * L)$

Table 2: **Computation and communication complexity of different methods.** N represents the number of points. M represents the number of observation values for each point. L represents the number of distribution types in Algorithm 3. WS represents the window size. K represents the number of nodes in a cluster. α represents the ratio between the number of points that have different sets of mean and standard values and the total number of points in a window. β represents the ratio between the number of points that have different sets of mean and standard values and the number of all the points in a slice.

6.5 **Sampling**

In order to choose a slice to compute PDFs, we need to compute the features of a slice very quickly. We propose a sampling method (see Algorithm 5) that samples the points and uses ML prediction to generate the distribution type of each point. This method does not run the statistical calculation of the each point in the $ComputePDF\&Error$ function in Algorithm 1 in order to reduce execution time.

In Algorithm 5, Line 1 samples the points in slice i based on a predefined sampling rate. A sampling rate represents the ratio between the selected points from the sampling process and all the original points. Lines 3-13 implement the process that loads the data from the datasets and calculates the mean and standard deviation values for each double sampled point. Line 14 groups the data as explained in Section 6.2. When the number of nodes in the cluster is high, we can

[1] Let n_{window} be the number of points that have different sets of mean and standard values in a window and N_{window} be the total number of points in a window. Then, $\alpha = \frac{n_{window}}{N_{window}}$. We assume that there are l windows in a slice. When the points in different windows have different sets of mean and standard deviation values, $\beta = \frac{l * n_{window}}{l * N_{window}} = \alpha$. When k points have same sets of mean and standard deviation values in two different slices, $\beta = \frac{l * n_{window} - k}{l * N_{window}} < \alpha$.

remove Line 14 in order to reduce the time of shuffling. Lines 16 - 19 calculate the distribution type of each double sampled point based on a decision tree (see Section 6.3.1 for details). Lines 21 - 25 calculate the final results, *i.e.* the average mean value, the average standard deviation and the percentage of different distribution types. Lines 16 - 25 correspond to the PDF computation process.

Algorithm 5 Sampling process

Input: DS: a set of datasets produced by simulations; i: the ith slice to analyze; $model$: the decision tree corresponding to the relationship between the mean and standard value and the distribution type; $Types$: a set of distribution types; $rate$: the sampling rate

Output: $\bar{\mu}$: the average mean value of the slice; $\bar{\sigma}$: the average standard deviation of the slice; $TypesPercentage$: the percentage of distribution types of the points in the slice

1: $Points \leftarrow Sample(slice_i, rate)$
2: $RawData \leftarrow \emptyset$
3: **for each** $p \in points$ **do**
4: $\quad rd \leftarrow \emptyset$
5: $\quad$ **for each** $ds \in DS$ **do**
6: $\quad\quad data \leftarrow GetData(ds, p, i)$
7: $\quad\quad rd \leftarrow data \cup rd$
8: $\quad$ **end for**
9: $\quad \mu \leftarrow ComputeMean(rd)$
10: $\quad \sigma \leftarrow ComputeStd(rd)$
11: $\quad rd \leftarrow \mu \cup \sigma \cup rd$
12: $\quad RawData \leftarrow rd \cup RawData$
13: **end for**
14: $Points \leftarrow selectByGrouping(RawData)$
15: $allTypes \leftarrow \emptyset$
16: **for each** $p \in Points$ **do**
17: $\quad type \leftarrow predict(model, RawData)$
18: $\quad allTypes \leftarrow type \cup allTypes$
19: **end for**
20: $TypesPercentage \leftarrow \emptyset$
21: **for each** $type \in Types$ **do**
22: $\quad pct \leftarrow calculatePercentage(type, allTypes)$
23: $\quad TypesPercentage \leftarrow pct \cup TypesPercentage$
24: **end for**
25: $(\bar{\mu}, \bar{\sigma}) \leftarrow averageCalculation(RawData)$
end

We can randomly sample the points. The random sampling method takes much time while the selected points may contain little repeated information. We could also use a k-means clustering algorithm [33] and choose the point that is the closest to the center of each cluster as double sampled points. The double sampled data generated by k-means contains diverse information, which does not help much to choose a slice (see details in Section 7.2.3). Furthermore, k-means may take much time to converge. Therefore, instead of k-means, we use random sampling in Algorithm 5.

7 Experimental Evaluation

In this section, we evaluate and compare the different methods presented in Section 6 for different datasets and cluster sizes. To ease reading, we call each method by a name as: Baseline, Grouping, Reuse, ML and Sampling. First, we introduce the experimental setup, with two different clusters

(a small one and a big one). Then, we perform experiments with a 235 GB dataset. Finally, we perform experiments with big datasets (1.9 TB and 2.4 TB).

7.1 **Experimental Setup**

In this section, we present the different spatial datasets, the candidate distribution types (see Algorithms 3 and 5) and the cluster testbeds, which we use in our various experiments.

To generate the spatial data, we use the UQlab framework [32] to produce 16 values as the input parameters, *i.e.* Vp, of the models from the seismic benchmark of the HPC4e project [2]. The input parameters obey four distribution types, *i.e.* normal, exponential, uniform and log-normal. In each simulation, we generate a set of the input parameters according to the PDF of each layer and generate a spatial dataset by using the set of the input parameters and the models. We run the simulation multiple times and generate three sets of spatial datasets, denoted by $Set1$, $Set2$ and $Set3$. The simulation is repeated 1000 times to generate $Set1$. $Set1$ contains 1000 files, each of which is a spatial dataset and has 235 GB. In $Set1$, the dimension of the cube area is 251 * 501 * 501, *i.e.* 501 slices, each slice has 501 lines and each line is composed of 251 points. To generate $Set2$, we run the simulation 1000 times while the dimension is 501 * 1001 * 1001. $Set2$ has 1.9 TB. A point is associated to 1000 observation values in both $Set1$ and $Set2$. Finally, we run the simulation 10000 times with the dimension of 251 * 501 * 501 in order to generate $Set3$, which has 2.4 TB. In this dataset, a point is associated to 10000 observation values. We use the same slice (Slice 201 because it has interesting information) in all the experiments. We consider two sets of candidate distribution types $Types$, introduced in Algorithms 3 and 5. The first set is the set of distribution types of the input parameters, *i.e.* normal, exponential, uniform and log-normal, since we assume that the distribution types of different points are within those of the input parameters of the simulation. In addition, we assume that the distribution types may belong to other types beyond the scope of the distribution types of the input parameters because of non-linear relationship between the input parameters and the values of each point of the spatial cube area. With this assumption, we have a second type of candidate distribution types, *i.e.* normal, exponential, uniform, log-normal, Cauchy, gamma, geometric, logistic, Student's t and Weibull. We call the first set $4 - types$ and the second $10 - types$.

We use two cluster testbeds, each with NFS, HDFS and Spark deployed. The first one, which we call LNCC cluster, is a cluster located at LNCC with 6 nodes, each having 32 CPU cores and 94 GB memory. The second one is a cluster of Grid5000, which we call G5k cluster, with 64 nodes, each having 16 CPU cores and 131 GB of RAM storage.

7.2 **Experiments on a 235 GB Dataset**

In this section, we compare the different methods based on the spatial dataset $Set1$ of 235 GB. First, we compare the performance of different methods using the LNCC cluster: Baseline, Grouping, Reuse and the combination of these methods with ML. Then, we study the scalability of different methods using the G5k cluster. Finally, we study the performance of Sampling.

We take the relationship between the set of mean and standard deviation values and the distribution types of 25000 points in Slice 0 in the previously generated output data of $Set1$ to tune the hyperparameters of the decision tree. The hyperparameters are tuned at the very beginning, which takes 18 minutes for $4 - types$ and 22 minutes for $10 - types$ using Spark on a workstation with 8 CPU cores and 32 GB of RAM.

We take advantage of the previously generated output data of $Set1$ to generate the decision tree model for the experiments of this section. The model error is 0.03 for $4 - types$, and 0.08 for

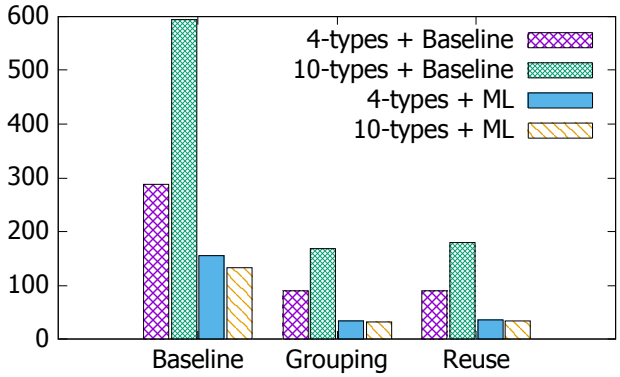


Fig. 8: **Execution time for PDF computation on a small workload (6 lines and 3006 points) with 235 GB input data.** The time unit is second.

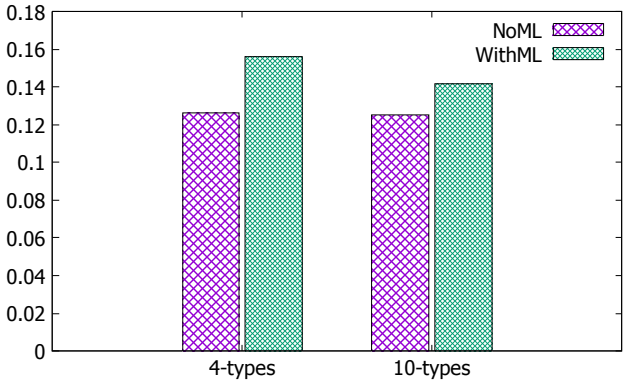


Fig. 9: **Error in PDF computation.** NoML represents the error of different methods without adoption of ML, *i.e.* Baseline, Grouping and Reuse with $4-types$ or $10-types$ (see Figure 8). WithML represents the combinations of methods with ML, *i.e.* Baseline + ML, Grouping + ML, Reuse + ML with $4-types$ or $10-types$ (see Figure 8).

$10-types$. Although the model error of $10-types$ is bigger than that of $4-types$, the average error E in Algorithm 1 is smaller. The time to train the model ranges from 1 to 20 seconds, which is negligible compared with the execution time of the whole PDF computation process.

7.2.1 Performance Comparisons

In this section, we compare the performance of different methods, *i.e.* Baseline, Grouping, Reuse, and the combination with ML, using the LNCC cluster. First, we execute the Scala program (for computing PDFs with different methods) on a small workload (6 lines and 3006 points). Second, we compare the performance of different methods for different window sizes. Finally, we compare the performance of different methods with the tuned window size for the whole slice.

Experiments with Small Workload In this section, we use a small workload of 6 lines, *i.e.* 3006 points, to compare the performance of different methods. The execution time for PDF computation is shown in Figure 8 and the error is shown in Figure 9.

We execute the program for the points of the first 6 lines in Slice 201 using Baseline, Grouping, Reuse, ML and the combination of these methods with ML. We take 3 lines as a window for PDF computation and we process the data of the points in two windows. The execution time for data loading (see Algorithm 2) is 67s, which is the same for all the methods since we use the same algorithm.

Figure 8 shows the good performance of our methods: Grouping (without Reuse), Reuse and ML. The execution time of Baseline with $10-types$ is much longer than that with $4-types$. This is expected since the complexity of Algorithm 3 is $O(n)$, with n distribution types, and the execution time increases with n. However, with ML, the execution time is significantly reduced (46% for $4-types$ and 78% for $10-types$). If we use Grouping without ML or Reuse, the execution time is also reduced a lot (69% for $4-types$ and 72% for $10-types$). This shows that there are many points that obey the same distribution and have the same mean and standard deviation values. In addition, since the data size corresponding to a point is small, *i.e.* 1000 observation values, the shuffling time for computing the aggregation function is short. Thus, Grouping outperforms Baseline much. When we couple Grouping and ML, the advantage becomes more obvious. The combined method is 88% and 95% better compared with the Baseline for $4-types$ and $10-types$, *i.e.* the combined method is up to more than 17 times better than Baseline. Reuse is slightly worse than Grouping, but it is still much better than Baseline. This is expected since it takes more time to search for the existing results than to compute PDFs. The difference between Grouping and Reuse is small when the workload is small but it can be significant for bigger workloads.

Figure 9 shows the average error (E in Equation 6) corresponding to different methods. In Figure 9, we distinguish between the methods that do not use ML, *i.e.* Baseline, Grouping and Reuse, which we call NoML, and those that do exploit ML, *i.e.* Baseline with ML, Grouping with ML and Reuse with ML, which we call WithML. The methods NoML and WithML have the same error for the same distribution type set.

The figure shows that $10-types$ does not reduce much the average error for Baseline (up to 0.0013). However, PDF computation may take more time when we consider $10-types$ as shown in Figure 8. The average error is higher for WithML than NoML. The difference is small (up to 0.017 for $10-types$ calculated based on the measured error shown in Figure 9) but WithML can reduce much the execution time of the PDF computation. In addition, Figure 9 shows that $10-types$ leads to a smaller average error, even though the model error is higher for WithML. This is reasonable since there are some types that are very difficult to distinguish in $10-types$ but the wrong classification of the distribution types does not increase much the average error. In addition, with more distribution candidate types, the decision tree produces a better classification.

Window Size Adjustment The window size is critical for the PDF computation with Grouping. To adjust the window size, we conduct the following experiment. We execute the Scala program with Grouping (with $4-types$ and without ML prediction) for two windows while each window is composed of different numbers of lines. Then, we choose a window that corresponds to the shortest average execution time of each line. Figure 10 shows the average execution time of PDF computation for different window sizes using Grouping. As shown in Figure 10, the average execution time of each line decreases for larger window sizes. This is reasonable because when the number of lines increases, more points are aggregated to each group so that more redundant calculations are avoided. When the window size is 25 lines, the average execution time of each line is minimal. From that point on, when the window size increases, the execution time also increases. This is expected since when the number of lines in a window increases, the time to shuffle data among different nodes increases more than the reduced time by avoiding redundant calculations. As a result, the average execution time of each line becomes more significant. However, the

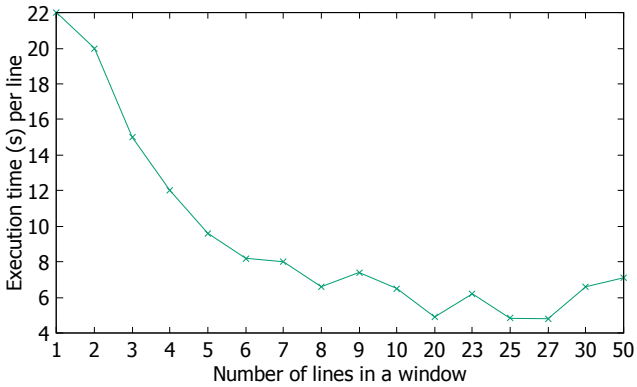


Fig. 10: **Average execution time per line for PDF computation with two windows.**

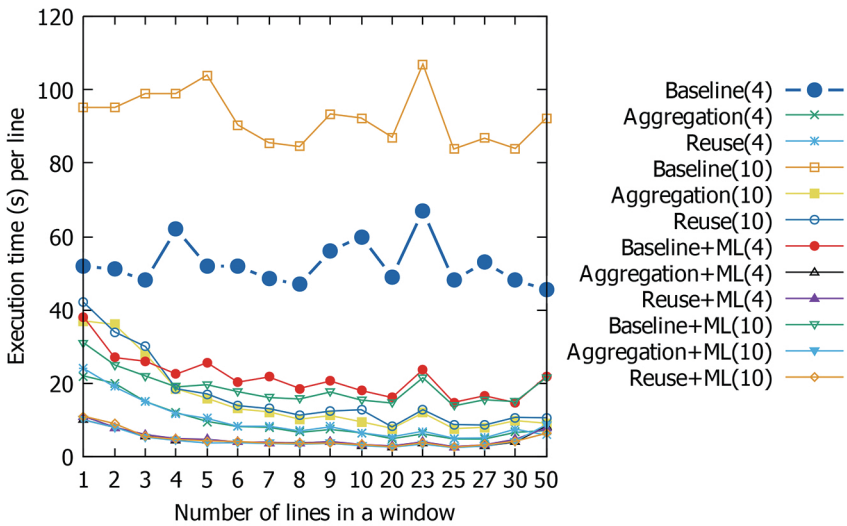


Fig. 11: **Average execution time per line for PDF computation with two windows.** With $4-types$ (4) and $10-types$ (10).

average execution time of data loading stays the same for different window sizes, *i.e.* about 12s per line.

We measure the execution time of PDF computation for different window sizes using other methods. In Figure 11, we can see that the window size of 25 is the optimal size for the other methods. In addition, the execution time of PDF computation is almost the same for different combinations of methods: Grouping plus ML and Reuse plus ML both with $4-types$ or $10-types$. Baseline always corresponds to longer execution times of PDF computation. Compared with Baseline, Grouping, Reuse and ML can reduce the execution time up to 91% (more than 10 times faster) and 84% (more than 6 times faster). In addition, the combination of Grouping and ML can be up to 97% (more than 33 times) faster. This shows the obvious performance advantage of our methods in computing PDFs.

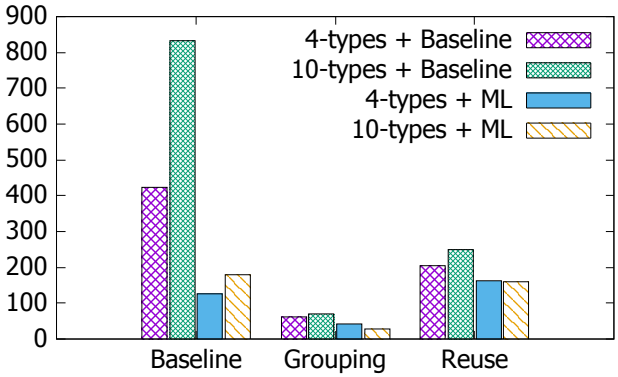


Fig. 12: **Execution time of PDF computation of Slice 201 with different methods (235 GB input data).** The time unit is minute. The window size is 25 lines.

Execution of One Slice In this section, we compare the performance of different methods to execute the program for the points of the whole slice, *i.e.* Slice 201. We take 25 lines as the window size for PDF computation and execute the program with different methods for the whole slice, *i.e.* 11 windows of points in Slice 201. The execution time of data loading is the same for the different methods, *i.e.* 4098s. The average execution time of each line is longer than that of the small workload since we cache all the data in memory during the execution of different methods and the time to store data increases as the amount of cached data grows. The execution time of PDF computation is shown in Figure 12 and the error is shown in 13.

Figure 12 shows that our proposed methods, *i.e.* Grouping, Reuse and ML, always outperform Baseline for both $4-types$ and $10-types$. Grouping can reduce the execution time up to 92% (more than 10 times) and ML up to 78% (more than 3 times faster). The performance of Reuse is better than that of Baseline when we do not combine ML and the advantage can be up to 70% (more than 2 times faster). However, the performance of the combination of Reuse and ML can be worse than the combination of Baseline and ML. This is possible when it takes too much time to search for the existing results. The combination of Grouping and ML can reduce the execution time up to 97% (more than 27 times faster) compared with Baseline.

Figure 13 shows the error of PDF computation. The error is smaller than that of the small workload. The error of NoML is still smaller than that of WithML. The error for $4-types$ is almost the same as that for $10-types$. But the error for $10-types$ using ML is smaller that for $4-types$. This is similar to what was observed when executing the small workload. In addition, the error for $10-types$ with ML is even smaller than the error for $4-types$ without ML. Although there is very small difference (up to 0.0016) of error between Baseline and ML, the execution time of the PDF computation process is largely reduce by ML.

7.2.2 Scalability Comparisons

In this section, we study the scalability of Baseline and the methods that have good performance, *i.e.* Grouping, ML and Grouping + ML. We use the G5k cluster with different numbers of nodes from 10 to 60.

The execution time of data loading with different methods remains the same for the same number of nodes. Figure 14 shows the execution time of data loading, which decreases rapidly as the number of nodes increases. This indicates the good scalability of our data loading process.

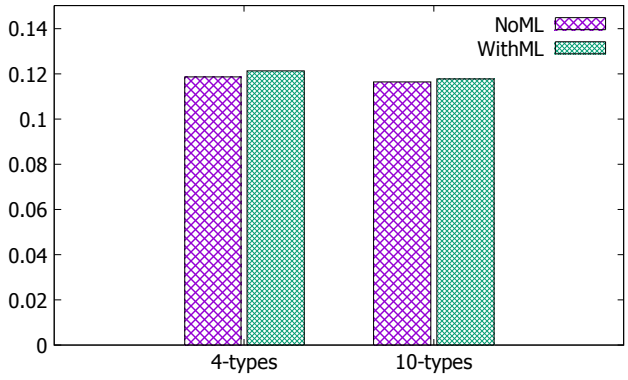


Fig. 13: **Error in PDF computation.** NoML stands for Baseline, Grouping and Reuse plus $4-type$ and $10-types$. WithML stands for Baseline + ML, Grouping + ML, Reuse + ML $4-type$ and $10-types$.

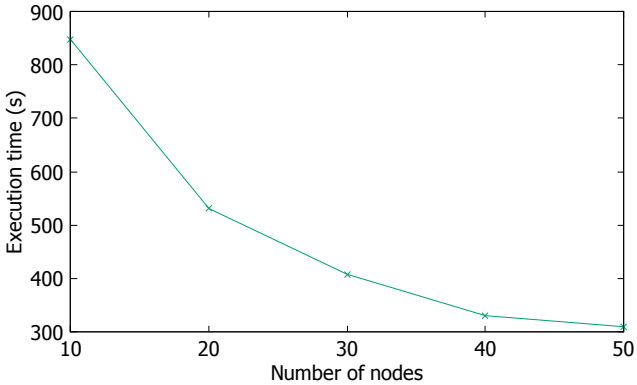


Fig. 14: **Execution time for data loading with different numbers of nodes.**

The execution time of PDF computation for different methods and different numbers of nodes is shown in Figure 15. We do not consider Reuse since it is less efficient than Grouping. We focus on ML for $10-types$ because it has small error and the execution time is similar to that for $4-types$. The figure shows that the execution time of Grouping and ML is always shorter than that of Baseline. The advantage of Grouping can be up to 87% (more than 6 times faster). ML outperforms Baseline up to 89% (more than 8 times faster). Grouping + ML can be better than Baseline by up to 90% (more than 9 times faster). In addition, the execution time of each method decreases as the number of nodes increases, which indicates that all methods have good scalability. However, the advantage becomes less obvious from 50 nodes on.

Figure 16 gives a focus on our methods. We do not compare the error of PDF computation since it is similar to that given in Section 7.2.1. The figure shows that Grouping + ML is better than either Grouping or ML when the number of nodes is 10. However, ML starts outperforming Grouping + ML when the number of nodes exceeds 10. This is because the shuffling of data among different nodes takes much time. As the number of nodes increases, the time to transfer data among the nodes increases. Thus, the performance of the aggregation function becomes a bottleneck for Grouping + ML when the number of nodes is high.

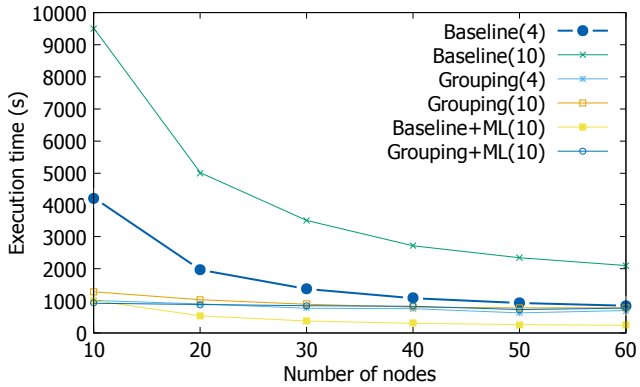


Fig. 15: **Execution time of PDF computation with different numbers of nodes.**

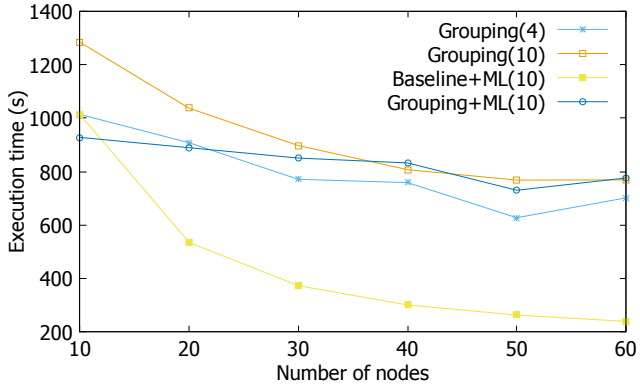


Fig. 16: **Execution time of PDF computation with different numbers of nodes and a focus on Grouping, ML, Baseline + ML and Grouping + ML.**

7.2.3 Performance of Sampling

We now study the efficiency of Sampling. We carried out the experiments in the LNCC cluster. We use two sampling methods, *i.e.* random sampling and k-means clustering (see Section 6.5). We compare the performance of the two sampling method with different sampling rates.

Figure 17 shows the execution time of random sampling with different sampling rates. The execution time for data loading decreases almost linearly as the sampling rate decreases (both the X and Y axis use a base-10 log scale). This is reasonable since when the sampling rate gets small, the data loading needs to process less points and thus loads less data. The execution time for PDF computation is very short (about 2 seconds). This is expected since we avoid calling the R program to compute PDFs and use a decision tree model to predict the distribution type of each point. The execution time is almost the same for different sampling rates since it is already very short and data shuffling also takes some time. In addition, we only need to transfer the mean and standard deviation values instead of the whole dataset for the prediction, which also reduces time. This execution time is about 2 seconds, which is very short. However, this method cannot calculate the error of the PDF computation process. It can help to have a general view of the whole slice. Then a slice is chosen to compute the PDFs of the corresponding points.

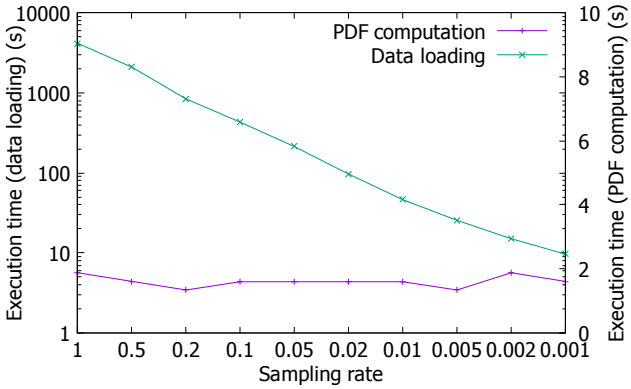


Fig. 17: **Execution time with different sampling rates using random sampling.**

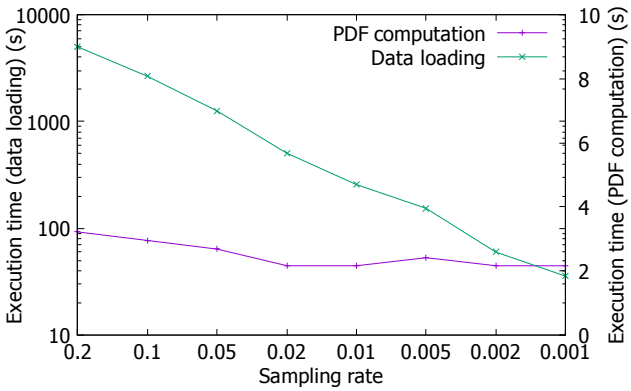


Fig. 18: **Execution time with different sampling rates using k-means sampling.**

Figure 18 shows the performance of k-means clustering for sampling the points. The results are almost the same as that of random sampling, *i.e.* the execution time for data loading decreases linearly and the execution time for PDF computation is very short and almost the same for different sampling rates. The execution time of k-means is longer than that of random for the same sampling rate. We measure the sampling rate from 0.2 since the corresponding execution time of k-means for data loading is already longer than that of random sampling with the sampling rate of 1.

Figure 19 shows the Euclidean distance of the distribution type percentage between the double sampled points and all points in one slice using k-means and random sampling. When the sampling rate is small, the result of k-means is close to the percentage of overall points. This is because the double sampled points of k-means method contain diverse information. However, when the sampling rate is high, the results of random sampling are similar or better since enough points are selected with a high sampling rate. Since random sampling is faster than k-means, we choose it in the following experiments.

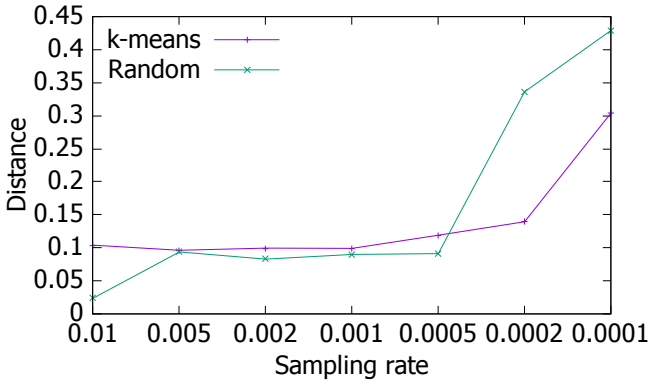


Fig. 19: **Distance of the distribution type percentage between the double sampled points and all points.**

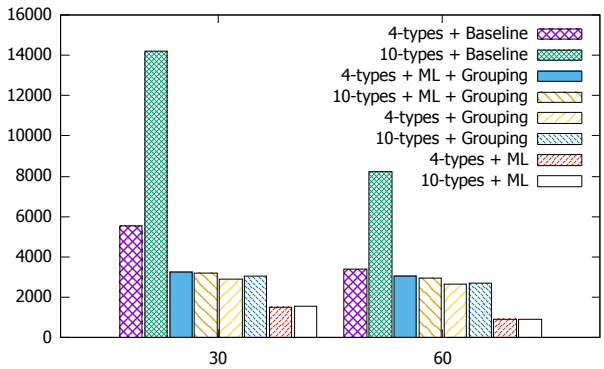


Fig. 20: **Execution time for all the points in Slice 201 (1.9 TB input data).** The time unit is second.

7.3 Experiments on Big Datasets

In this section, we compare the performance of different methods based on big spatial datasets of several TB, *i.e.* $Set2$ and $Set3$ (see details in Section 7.1). We use the G5k cluster with 30 and 60 nodes.

7.3.1 Experiments with 1000 Simulations

In this section, we perform experiments using $Set2$ of 1.9 TB generated by 1000 simulations. We take the relationship between the combination of mean and standard deviation values and the distribution types of 25000 points in Slice 0 as previously generated data to build the decision tree model. The model error of $4 - types$ is 0.02 and the model error of $10 - types$ is 0.09. The time to load the model ranges between 1 and 20 seconds, which is negligible compared with the execution time of PDF computation. The time for data loading is 2671 seconds with 30 nodes and 1619 seconds with 60 nodes, which shows the good scalability of the data loading process.

Figure 20 shows the good performance of our methods, *i.e.* Grouping and ML. Grouping is better than Baseline (up to 79%) but not as good as ML, because of data shuffling with many

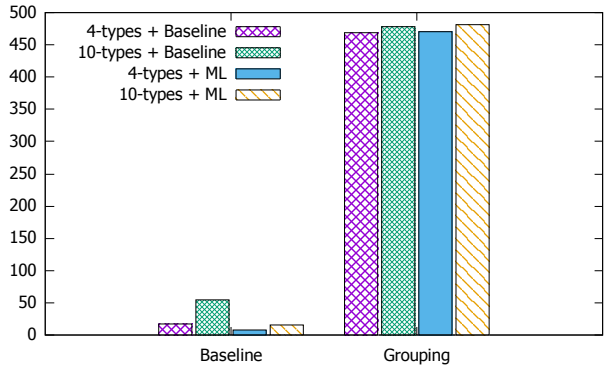


Fig. 21: **Execution time for PDF computation on small workload (2 lines and 1002 points) with 2.4 TB input data.** The time unit is second.

nodes. And the scalability of Grouping is not good. ML largely outperforms Baseline (up to 89%) and has good scalability. Because of the shuffling bottleneck, the combination of Grouping and ML is worse than ML but still better (up to 77%) than Baseline. The error of ML with $10-types$ is always smaller than with $4-types$.

Finally, we experimented with random sampling to process the data in two clusters of 30 and 60 nodes. The execution time of PDF computation ranges between 260s and 280s while the sampling rate ranges between 0.001 and 1. The average execution time (272s for 30 nodes and 266 for 60 nodes) is 82% and 71% shorter than the minimum execution time (ML with $4-types$ for 30 and 60 nodes) as shown in Figure 20. Note that doubling the number of nodes does not yield much improvement. This is because using more nodes increases data transfers to send the mean and standard deviation values and the distribution type from each node to the Spark master node in Spark cluster to compute the distribution percentage.

7.3.2 Experiments with 10000 Simulations

In this section, we perform experiments using $Set3$ of 2.4 TB generated by 10000 simulations. Again, we take the relationship between the combination of mean and standard deviation values and the distribution types of 25000 points in Slice 0 as the previously generated data for the decision tree model. The model error of $4-types$ is 0.006 and the model error of $10-types$ is 0.012. The time to load the model ranges between 1 and 20 seconds, which is negligible compared with the execution time of PDF computation. First, we execute the program for the points of the first 2 lines in Slice 201 in a cluster of 30 nodes. We take 1 line as a window, and we process the data of two windows. The execution time for data loading (Algorithm 2) is 28s, which is the same for all the methods since we use the same algorithm (Algorithm 2). The execution time for PDF computation is shown in Figure 21.

Figure 21 shows the superior performance of ML. The execution time of Baseline with $10-types$ is much longer than that of $4-types$ as expected. Compared with Baseline, ML reduces execution time much (57% with $4-types$ and 72% with $10-types$). However, the execution time of Grouping is much longer. This is because the data size of each point is 9 times bigger since a point corresponds to 10000 observation values instead of 1000. During Grouping, much more data is transferred among different nodes, which takes much time. Thus, in the next experiment, we do not use Grouping.

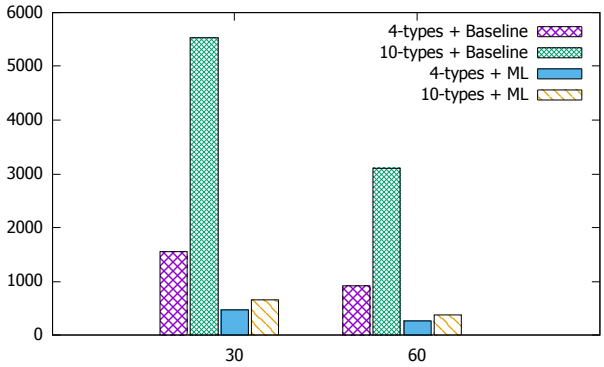


Fig. 22: **Execution time for all the points in Slice 201 (2.4 TB input data).** The time unit is second.

We also measure the error (E defined in Equation 6) during execution. $10 - types$ does not reduce much the average error for Baseline (up to 0.008). However, it may take much more time for PDF computation when we consider $10 - types$ as shown in Figure 21. Using WithML, the average error is slightly bigger than that of NoML. The difference is very small (up to 0.007) but ML can reduce much the execution time of PDF computation. Furthermore, $10 - types$ leads to smaller average error even though the model error is bigger when using ML.

Now, we execute the program for all points in Slice 201. As we only compare the execution with Baseline and ML, we take 126 lines as a window in order to parallelize the execution of different points in different nodes. The time for data loading is 4592 seconds with 30 nodes. Figure 22 shows the superior performance of ML (up to 88%, *i.e.* more than 7 times faster than Baseline). Furthermore, the PDF computation process scales very well. We measured the average error and found that ML incurs small error while reducing execution time much. Finally, we use the random sampling to process the data in two clusters of 30 and 60 nodes. The execution time of PDF computation ranges between 128s and 200s while the sampling rate ranges between 0.001 and 1. The average execution time (172s for 30 nodes and 155s for 60 nodes) is 64% and 41% smaller than the minimum execution time (ML with $4 - types$ for 30 and 60 nodes) as shown in Figure 22. Again (as for the experiment with 1000 simulations), doubling the number of nodes does not yield much improvement.

8 Conclusion

Uncertainty quantification of spatial data requires computing a Probability Density Function (PDF) of each point in a spatial cube area. However, computing PDFs on big spatial data, as produced by applications in scientific areas such as geological or seismic interpretation, can be very time consuming.

In this paper, we addressed the problem of efficiently computing PDFs under bounded error constraints. We proposed a parallel solution using a Spark cluster with three new methods: Grouping, ML and Sampling. Grouping aggregates the points of the same statistical features together in order to reduce redundant calculation. This method is very efficient when the data to be transferred is not too big and the number of cluster nodes is small. ML generates a decision tree model based on previously generated data and predicts the distribution type of a point in order to avoid useless calculation based on wrong distribution types. Sampling enables to

efficiently compute statistical parameters of a region by sampling a fraction of the total number of points to reduce the computation space.

To validate our solution, we implemented these methods in a Spark cluster and performed extensive experiments on two different clusters (with 6 and 64 nodes) using big spatial data ranging from hundreds of GB to several TB. This data was generated from simulations based on the models from a seismic benchmark for oil and gas exploration, which includes models for seismic wave propagation.

The experimental results show that our solution is efficient and scales up very well compared with Baseline. Grouping outperforms Baseline by up to 92% (more than 10 times) without introducing extra error. ML can be up to 91% (more than 9 times) better than Baseline with very slight acceptable error (up to 0.017). The combination of Grouping and ML can be up to 97% (more than 33 times) better than Baseline. As the number of nodes exceeds 10 nodes, ML outperforms the combination. Thus, in order to compute PDFs, the combination of Grouping and ML is the optimal method when each point corresponds to a small number of observation values, *e.g.* 1000, and when there is small number of nodes (less than 20). Otherwise, ML is the best option. We also showed that Sampling is very efficient to calculate general statistics information in order to choose a slice for calculating PDFs. Finally, Sampling should be used with the aforementioned best option in order to efficiently compute PDFs.

Acknowledgment

This work was partially funded by EU H2020 Project HPC4e with MCTI/RNP-Brazil, CNPq, FAPERJ, and Inria Associated Team SciDISC. The experiments were carried out using a cluster at LNCC in Brazil and the Grid5000 testbed in France (`https://www.grid5000.fr`).

References

1. Fitdistr Function in R language. `https://www.rdocumentation.org/packages/MASS/versions/7.3-51.1/topics/fitdistr`.
2. Hpc geophysical simulation test suite. `https://hpc4e.eu/downloads/hpc-geophysical-simulation-test-suite`.
3. Spark MLib. `https://spark.apache.org/mllib/`.
4. O. Y. Al-Jarrah, P. D. Yoo, S. Muhaidat, G. K. Karagiannidis, and K. Taha. Efficient machine learning for big data: A review. *Big Data Research*, 2(3):87–93, 2015.
5. J. Aldrich. R.a. fisher and the making of maximum likelihood 1912-1922. *Statistical Science*, 12(3):162–176, 1997.
6. L. V. Ballestra, G. Pacellib, and D. Radi. A very efficient approach to compute the first-passage probability density function in a time-changed brownian model: Applications in finance. *Physica A: Statistical Mechanics and its Applications*, 463(1):330–344, 2016.
7. R. Belohlávek, B. D. Baets, J. Outrata, and V. Vychodil. Inducing decision trees via concept lattices. *Int. Journal of General Systems*, 38(4):455–467, 2009.
8. B. Bohn, J. Garcke, R. Iza-Teran, A. Paprotny, B. Peherstorfer, U. Schepsmeier, and C. Thole. Analysis of car crash simulation data with nonlinear machine learning methods. In *Int. Conf. on Computational Science ICCS*, pages 621–630, 2013.
9. R. Campisano, H. Borges, F. Porto, F. Perosi, E. Pacitti, F. Masseglia, and E. S. Ogasawara. Discovering tight space-time sequences. In *Int. Conf. on Big Data Analytics and Knowledge Discovery*, pages 247–257, 2018.
10. R. Campisano, F. Porto, E. Pacitti, F. Masseglia, and E. S. Ogasawara. Spatial sequential pattern mining for seismic data. In *Simpósio Brasileiro de Banco de Dados (SBBD)*, pages 241–246, 2016.
11. Y. Chalabi and D. Würtz. Flexible distribution modeling with the generalized lambda distribution. 2012. 2012-05.
12. M. Chen, S. Mao, and Y. Liu. Big data: A survey. *Mobile Networks and Applications*, 19(2):171–209, 2014.

13. R. V. Coile, G. Balomenos, M. Pandey, R. Caspeele, P. Criel, L. Wang, and S. Alfred. Computationally efficient estimation of the probability density function for the load bearing capacity of concrete columns exposed to fire. In *Int. Symposium of the Int. Association for Life-Cycle Civil Engineering (IALCCE)*, page 8, 2016.
14. T. Condie, P. Mineiro, N. Polyzotis, and M. Weimer. Machine learning on big data. In *29th IEEE Int. Conf. on Data Engineering, ICDE*, pages 1242–1244, 2013.
15. N. Cressie. *Statistics for spatial data*. John Wiley & Sons, 2015.
16. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Symp. on Operating System Design and Implementation (OSDI)*, pages 137–150, 2004.
17. J. R. del Val, F. Simmross-Wattenberg, and C. Alberola-López. libstable: Fast, parallel, and high-precision computation of α-stable distributions in r, c/c++, and matlab. *Journal of Statistical Software*, 78(1):1–25, 2017.
18. W. J. Dixon and F. J. Massey. *Introduction to statistical analysis*. 1968.
19. S. Fotheringham, C. Brunsdon, and M. Charlton. *Quantitative Geography: Perspectives on Spatial Data Analysis*. 2000.
20. M. Friedl and C. Brodley. Decision tree classification of land cover from remotely sensed data. *Remote Sensing of Environment*, 61(3):399 – 409, 1997.
21. M. Gheisari, G. Wang, and M. Z. A. Bhuiyan. A survey on deep learning in big data. In *IEEE Int. Conf. on Computational Science and Engineering, CSE, and IEEE Int. Conf. on Embedded and Ubiquitous Computing, EUC*, pages 173–180, 2017.
22. S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *ACM Symp. on Operating Systems Principles (SOSP)*, pages 29–43, 2003.
23. E. R. Harold. *Java I/O: Tips and Techniques for Putting I/O to Work*, pages 131–132. 2006.
24. G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.
25. T. J. Jackson, D. M. L. Vine, A. Y. Hsu, A. Oldak, P. J. Starks, C. T. Swift, J. D. Isham, and M. Haken. Soil moisture mapping at regional scales using microwave radiometry: the southern great plains hydrology experiment. *IEEE Transactions Geoscience and Remote Sensing*, 37(5):2136–2151, 1999.
26. F. Kathryn, J. T. Oden, and D. Faghihi. A bayesian framework for adaptive selection, calibration, and validation of coarse-grained models of atomistic systems. *Journal of Computational Physics*, 295:189 – 208, 2015.
27. T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Int. Conf. on Management of Data (SIGMOD)*, pages 489–504, 2018.
28. S. Landset, T. M. Khoshgoftaar, A. N. Richter, and T. Hasanin. A survey of open source tools for machine learning with big data in the hadoop ecosystem. *Journal of Big Data*, 2(1):24, 2015.
29. J. Liu, E. Pacitti, and P. Valduriez. A survey of scheduling frameworks in big data systems. *International Journal of Cloud Computing*, page 27, 2018.
30. R. H. C. Lopes. Kolmogorov-smirnov test. In *Int. Encyclopedia of Statistical Science*, pages 718–720, 2011.
31. S. Marelli and B. Sudret. *UQLab: A Framework for Uncertainty Quantification in MATLAB*. ETH-Zürich, 2014.
32. S. Marelli and B. Sudret. Uqlab: A framework for uncertainty quantification in MATLAB. In *Int. Conf. on Vulnerability, Risk Analysis and Management (ICVRAM)*, pages 2554–2563, 2014.
33. X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.
34. C. Michele, T. Stefano, and S. Andrea. Sensitivity and uncertainty analysis in spatial modelling based on gis. *Agriculture, Ecosystems & Environment*, 81(1):71 – 79, 2000.
35. J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.
36. E. E. Prudencio and K. W. Schulz. The parallel C++ statistical library 'queso': Quantification of uncertainty for estimation, simulation and optimization. In *Euro-Par: Parallel Processing Workshops*, pages 398–407, 2011.
37. J. Ramberg and B. W. Schmeiser. An approximate method for generating asymmetric random variables. *Commun. ACM*, 17(2):78–82, 1974.
38. S. R. Safavian and D. Landgrebe. A survey of decision tree classifier methodology. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3):660–674, 1991.
39. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network file system. In *the Summer USENIX conf.*, pages 119–130, 1985.
40. S. Shalev-Shwatrz and S. Ben-David. *Understanding Machine Learning - From Theory to Algorithms*. Cambridge University Press, 2017.
41. K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *IEEE Symp. on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.
42. S. Singer and S. Singer. Complexity analysis of nelder–mead search iterations. In *Conf. on Applied Mathematics and Computation*, pages 185–196, 1999.

43. P. Snyder. tmpfs: A virtual memory file system. In *European UNIX Users' Group Conf.*, pages 241–248, 1990.
44. S. Suthaharan. Big data classification: Problems and challenges in network intrusion prediction with machine learning. *ACM SIGMETRICS Performance Evaluation Review*, 41(4):70–73, 2014.
45. G. Trajcevski. Uncertainty in spatial trajectories. In *Computing with Spatial Trajectories*, pages 63–107, 2011.
46. F. Wang and J. Liu. Networked wireless sensor data collection: Issues, challenges, and approaches. *IEEE Communications Surveys and Tutorials*, 13(4):673–687, 2011.
47. E. D. Z. Karian. *Fitting Statistical Distributions: The Generalized Lambda Distribution and Generalized Bootstrap Methods*. Chapman and Hall/CRC, 2000.
48. B. Zadrozny and C. Elkan. Obtaining calibrated probability estimates from decision trees and naive bayesian classifiers. In *Int. Conf. on Machine Learning (ICML)*, pages 609–616, 2001.
49. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.