Information Systems 98 (2021) 101710

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Fast, scalable and geo-distributed PCA for big data analytics

T. M. Tariq Adnan [a], Md. Mehrab Tanjim [b], Muhammad Abdullah Adnan [a,*]

[a] Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh
[b] University of California San Diego (UCSD), CA, USA

A R T I C L E I N F O

Article history:
Received 26 May 2019
Received in revised form 19 November 2020
Accepted 28 December 2020
Available online 6 January 2021
Recommended by Dennis Shasha

Keywords:
Big data
PCA
Dimensionality reduction
Geo-distributed algorithm

A B S T R A C T

Principal Component Analysis (PCA) is a widely popular technique for reducing the dimensionality of a dataset. Interestingly, when dimensions of the dataset grow too large, existing state-of-the-art methods for PCA face scalability issue due to the explosion of intermediate data. Moreover, in a geographically distributed environment where most of today's data are originally generated, these methods require unnecessary data transmissions as they apply centralized algorithms for PCA and thus are proven to be inefficient. To solve these problems, we take advantage of the zero-noise-limit Probabilistic PCA model, which provably outputs the correct principal components, and introduce a block-division method for it in order to suppress the explosion of intermediate data efficiently. We employ several optimization ideas such as mean propagation for preserving sparsity, dynamic tuning of the number of blocks to automatically adjust to large dimensions, etc. Additionally, in the geo-distributed environment, we propose a communication efficient solution by reducing idle time, passing only the required parameters, and choosing geographically ideal central datacenter for faster accumulation. We refer to our algorithm as TallnWide. Our empirical evaluation with real datasets shows that TallnWide can successfully handle significantly higher dimensional data (**10×**) than existing methods, and offer up to **2.9×** improvement in running time in the geo-distributed environment compared to the conventional approaches. For reproducibility and extensibility of our work, we make the source code of TallnWide publicly available at https://github.com/tmadnan10/TallnWide.

1. Introduction

The frequency of data access at the users' end has increased by a large amount for the past few years. With such rapid generation, data that appear in many applications, such as social networks [1], product ratings [2], web documents [3], etc., often become high-dimensional. Unfortunately, most of the machine learning algorithms cannot operate with such a high number of dimensions [4–7]. As Principal Component Analysis (PCA) is a widely used technique for dimensionality reduction [8–11], it is often desirable to reduce its dimensions (number of columns) using PCA to make it thin (lower number of dimensions), and then apply other machine learning techniques on it. PCA can also be used for lossy-data compression [12], feature extraction [13], and data visualization [14].

There are several state-of-the-art libraries that offer PCA for distributed clusters, namely: Mahout [15], MLlib [16], sPCA [17], and sSketch [18]. However, PCA algorithms implemented in these libraries are not suitable when dimensions (columns) of data grow proportionately with the number of data samples (rows). For example, Twitter Network [1] presents a dataset of $50M$ users' followers, which is a $50M \times 50M$ matrix (dimension, $D = 50M$). These data are quite large concerning both rows (tall) and columns (wide). We refer to such data as *tall and wide big data*. If we run sPCA or Mahout-PCA on it, for computing the first 100 principal components, it takes over **37.25 GB of memory** per worker node to store the parameter alone (by parameter, we mainly indicate the principal components or subspace). For MLlib-PCA the situation is much worse. Even for $D = 100K$, the parameter takes roughly **74.50 GB of memory per worker node**. sSketch-PCA can provide good scalability up to a certain extent, and even then, it faces memory overflow error for datasets with significantly larger dimensions (failed for $D \approx 10M$). In summary, current techniques face **out-of-memory error**, and to the best of our knowledge, there is no existing solution.

On top of data being tall and wide, they are often geographically distributed as almost all the companies (e.g. Twitter, Facebook, etc.) store information in multiple geographic locations to ensure privacy and low latency at the users' end [19–22]. In this setting, we do not have a global view of such distributed data. Therefore, even if we deal with datasets with relatively smaller dimensions for which the parameter fits in the memory, we still need to gather partial datasets that are spread across different geographic locations and run existing methods on this centralized

* Corresponding author.
E-mail address: abdullah.adnan@gmail.com (M.A. Adnan).

https://doi.org/10.1016/j.is.2020.101710
0306-4379/© 2021 Elsevier Ltd. All rights reserved.

data [19,23,24]. Communication over regions is expensive, and some countries even prohibit passing *raw data* across national borders [21,22].

Under these circumstances, at present, we need an analytic method that is capable of (1) handling tall and wide big data, and (2) performing a communication-efficient calculation in the geo-distributed environment. In this work, we propose solutions for each of the cases and present a highly scalable algorithm, namely *TallnWide*, for computing PCA. Our main contributions are as follows:

- To manage tall and wide data, we need a solution that can scale up for any arbitrarily large number of dimensions and mitigate the memory overflow error. Block-division is a right candidate solution for such a problem as it allows the computation to get divided into manageable blocks. However, not all techniques of PCA allow computation to be divided into multiple partitions/blocks. Also, dividing the computation is non-trivial because of the interlinked dependencies between matrices and various steps of matrix operations. To solve these challenges, firstly, we identify a variant (zero-noise-limit) of Probabilistic PCA (PPCA) [25], which has a much simpler dependency graph compared to conventional PPCA and can produce results in fewer steps. Because of its simplicity, it also allows the computation to be easily divided into multiple blocks. In this work, we propose a block-division algorithm for it to scale up PCA for any arbitrarily large number of dimensions and mitigate the memory overflow error. To the best of our knowledge, we are the first to give such a method for PCA.
- For a communication-efficient solution in the geo-distributed environment, we propose a scheme that seeks to minimize idle times in computation and avoid bottlenecks of communication by transmitting only the required parameters, and not the raw data. Also, we give a formula that can choose a geographically ideal central datacenter (DC) for faster accumulation (a central DC is needed to gather partial results from all DCs to produce the final result and transmit it back). Such a scheme saves both computation and communication costs by a significant margin. We refer to the whole algorithm as TallnWide.
- For implementing TallnWide, we consider the popular memory-based distributed framework, Spark [26]. We consider various optimization and effective ideas like tuning the number of blocks dynamically and employing efficient accumulation strategy. Moreover, similar to sPCA [17] and sSketch-PCA [18], we also propagate the mean for sparsity preservation. We run extensive experiments with four real large datasets with varying sizes and values in both geo-distributed (a cluster of DCs from three different regions) and local environments (each DC as a cluster). Our experiment shows that TallnWide is well capable of handling tall and wide big data in a datacenter with commodity computing hardware and can complete execution on datasets with dimensions as high as **50M** while existing methods **fail** to run on datasets where dimensions reach to **10M**. Additionally, in the geo-distributed environment, our approach offers up to **2.9×** improvement in running time in contrast to other alternatives.

The rest of the paper is organized as follows. In the following section, we discuss the primary motivations for proposing TallnWide in details. In Section 3, we introduce a brief technical background of existing PCA methods. Section 4 discusses state-of-the-art implementations of these methods. We also point out their limitations briefly. We present the design of our proposed algorithm in Section 5 with the complexity analysis of it in Section 6. Section 7 shows our experimental setup, while Section 8 illustrates the evaluation. Finally, Section 9 concludes the paper.

2. Motivation

As mentioned in the introduction, there are two main problems which motivate us to propose an efficient solution for PCA. The first one is the ever-increasing number of dimensions of data, which also makes it very sparse. This requires an algorithm which can handle an arbitrary number of dimensions and preserve sparsity at the same time. The second one is dataset being by born geographically distributed, which requires a communication efficient solution. We discuss both of them below with illustrative examples and use-cases.

2.1. High dimensionality with sparsity

With the increasing rate of data generation, the number of features, i.e. attributes per data sample has also been increasing by a large scale during the last decade. This results in the data to be very high dimensional, which eventually contributes to the increment of data sparsity. As we already mentioned, a wide range of fields such as social network [1], health sector [3], e-commerce [2], bio-metric, industries, etc. are often generating high dimensional and consequently sparse data.

As an illustrative example, let us consider the user–item interaction matrix, which is a crucial part of building a recommender system in various web platforms. In a user–item interaction matrix, each row denotes a new user, and each column denotes a new item. Typically, for any web service, such as large e-commerce systems, both the number of users and items are in millions, and both increase as the system sees a new user or adds a new item to inventory. However, each user interacts with only a few items resulting in a very sparse matrix. For instance, Amazon product data provided by [2] has 21M users' ratings on a total of 9.8M products. However, 99.99% of this dataset is sparse as most of the users rate only a few items (more details on this dataset is provided in Section 7.3).

This example shows how dimensions of data can proliferate and how sparsity of the overall dataset can increase along with it. Therefore, efficient data analytic techniques need to be well capable of dealing with the sparsity and higher dimensionality of big data.

2.2. PCA on geo-distributed datasets

Nowadays, data are by born geographically distributed, which has evolved the requirement of developing a geo-distributed or federated learning technique. The main principle of federated learning is simple: learn a shared model across multiple decentralized servers storing local datapoints, without exchanging them. This technique not only allows us to build a robust model with data from multiple sources but also ensures critical issues such as data privacy and security. There are many use-cases for such federated learning in various fields such as healthcare systems [27,28], Financial Services Industry (FSI) [29–31], IoT, telecommunications [32,33], etc.

For example, in the field of the healthcare systems, one significant usefulness of geo-distributed learning is to develop an analytical tool for measuring the effectiveness of particular treatments against groups of people all around the world. The target of such analysis is to identify the characteristic properties in patients demonstrating better and lower response. A similar analysis can be carried out in order to identify adverse drug reactions on the patients located at different geographic locations [27] or to extract significant insights from geo-distributed healthcare dataset [28]. With the help of geo-distributed data analytics, this kind of global benchmarking can analyze the patients' data at a location close to the collection spot within the geographic boundaries defined by regulatory compliance. Similar use-cases exist for Financial Services Industry (FSI) to preserve the security of the raw data [30] or to detect fraudulent activities [31].

As federated learning is an important and sometimes the only technique for geo-distributed data, over the years, it has caught the researchers' attention to reduce communication requirements [34,35], or ensure robustness to differential privacy attacks [36]. Currently, a good number of federated machine learning tools are already available [20,21,37]. Additionally, to meet the new challenges for running machine learning algorithms, and to provide a generalized framework for running the machine learning tools in a federated setup, Gaia [38] and TernGrad [39] were proposed. However, most of these available machine learning tools and frameworks are not suitable to operate on datasets with significantly higher dimensionality [4–7]. Therefore, when it comes to reducing the dimensionality of the data into a manageable one or perform any other pre-processing steps such as feature extraction [13], lossy-data compression [12], and data visualization [14], etc. in such a geo-distributed environment, the requirement of a federated PCA is certain.

Nevertheless, to the best of our knowledge, in such federated setup described above, there is no implementation of PCA, and out of the necessity, we focus on proposing a communication efficient solution for federated PCA. We discuss more related works in geo-distributed analytics in Section 4.5.

3. Technical background

In this section, we briefly go through each of the existing techniques for computing PCA. We consider the following notations: data/matrix[1] $\boldsymbol{Y}$, number of rows N, number of columns or we can say data dimension D, target dimension d (where $d \ll D$), principal component matrix $\boldsymbol{V}$ (the columns of $\boldsymbol{V}$ are the principal components of $\boldsymbol{Y}$). If $\boldsymbol{\mu}$ is the mean vector and $\ominus$ represents the row-wise subtraction of a vector from a matrix, then $\boldsymbol{Y}_c = \boldsymbol{Y} \ominus \boldsymbol{\mu}$ is the mean centering of the data $\boldsymbol{Y}$. Later, we will extend these notations when we discuss our proposal.

3.1. Eigen value decomposition

PCA can be computed from Eigen Value Decomposition (EVD) [40] of a covariance matrix [41]. If $\boldsymbol{C}$ is our covariance matrix, then $\boldsymbol{C} = (\boldsymbol{Y}_{\boldsymbol{c}}^T * \boldsymbol{Y}_c)/(N-1)$. We can get the principal components $\boldsymbol{V}$ and the Eigen values $\boldsymbol{D}$ of $\boldsymbol{C}$ by performing EVD: $[\boldsymbol{V}, \boldsymbol{D}] = evd(\boldsymbol{C})$, (see [42] for details).

3.2. Singular value decomposition

Singular Value Decomposition (SVD) is another method of computing PCA. If $\boldsymbol{U}$ is an $N \times D$ matrix with orthonormal columns ($\boldsymbol{U^T U} = \boldsymbol{I}$), $\boldsymbol{V}$ gives the principal components of mean centered data matrix $\boldsymbol{Y_c}$ which is a $N \times D$ orthonormal matrix ($\boldsymbol{V^T V} = \boldsymbol{I}$), and $\boldsymbol{\Sigma}$ is a $D \times D$ diagonal matrix with positive or zero elements, called the singular values, then SVD decomposes data matrix $\boldsymbol{Y_c}$ into three components:

$$\boldsymbol{Y_c} = \boldsymbol{U \Sigma V^T}$$

[1] Throughout the paper, we use the term 'data' and 'matrix' interchangeably.

3.3. Stochastic SVD

We can use randomized sketch matrix to reduce big data to smaller data [43,44] in order to derive principal components. This method is referred to as Stochastic Singular Value Decomposition (Stochastic SVD or simply SSVD) [45]. In SSVD, we first start with sketching the data matrix. To have a sketch matrix, $\boldsymbol{\Omega}$, we can draw $D \times m$ numbers randomly from the Gaussian distribution with *zero* mean and *unit* variance. Here m is the sketching dimension, which is greater than our target rank d, but much less than $D (m \ll D)$ and m controls the approximation [43]. We have to multiply $\boldsymbol{\Omega}$ with $\boldsymbol{Y}_c$ to get sketched data matrix: $\boldsymbol{Z} = \boldsymbol{Y}_c * \boldsymbol{\Omega}$. After having $\boldsymbol{Z}$, we have to perform QR Decomposition [41] on it to get orthonormal basis $\boldsymbol{Q}$: $[\boldsymbol{Q}, \boldsymbol{R}] = qr(\boldsymbol{Z})$. Then we have to perform matrix multiplication $\boldsymbol{Q}^T * \boldsymbol{Y}_c$. Finally we have to perform SVD [41] on $\boldsymbol{Q}^T * \boldsymbol{Y}_c$ to get principal component $\boldsymbol{V}$: $[\sim, \sim, \boldsymbol{V}] = svd(\boldsymbol{Q}^T * \boldsymbol{Y}_c)$. To increase the accuracy of the result, we can take this $\boldsymbol{Q}^T * \boldsymbol{Y}_c$ as new $\boldsymbol{\Omega}$ and repeat the previous steps before performing SVD on the final result. This is known as Power Iteration, and more details can be found in [43].

3.4. Probabilistic PCA

Probabilistic PCA (PPCA) is an example of a linear Gaussian model [25,46]. The observed variable $\boldsymbol{y_c}$ is related to a linear transformation of the latent variable $\boldsymbol{x}$ so that

$$\boldsymbol{y}_c = \boldsymbol{W} * \boldsymbol{x} + \boldsymbol{\sigma}$$

where $\boldsymbol{y}_c$ is a D-dimensional observed or data vector (mean centered), $\boldsymbol{W}$ is a $D \times d$-dimensional principal subspace, $\boldsymbol{x}$ is a d-dimensional Gaussian latent variable, and $\boldsymbol{\sigma}$ is a D-dimensional zero-mean Gaussian distributed noise variable with covariance $ss * \boldsymbol{I}$. So, we can say,

$$\boldsymbol{x} \sim \mathcal{N}(0, \boldsymbol{I}),\ \boldsymbol{\sigma} \sim \mathcal{N}(0, ss * \boldsymbol{I}),\ \boldsymbol{y_c} \sim \mathcal{N}(0, \boldsymbol{W} * \boldsymbol{W}^T + ss * \boldsymbol{I})$$

Given fixed model parameters $\boldsymbol{W}$ and ss, the following can be said about the hidden state $\boldsymbol{x}$, for some observation $\boldsymbol{y_c}$:

$$\begin{aligned} P(\boldsymbol{x}|\boldsymbol{y_c}) &= \frac{P(\boldsymbol{y_c}|\boldsymbol{x}) * P(\boldsymbol{x})}{P(\boldsymbol{y_c})} \\ &= \frac{\mathcal{N}(\boldsymbol{y_c}|\boldsymbol{W} * \boldsymbol{x},\ ss * \boldsymbol{I}) * \mathcal{N}(\boldsymbol{x}|0,\ \boldsymbol{I})}{\mathcal{N}(\boldsymbol{y_c}|0,\ \boldsymbol{W} * \boldsymbol{W}^T + ss * \boldsymbol{I})} \\ &= \mathcal{N}(\boldsymbol{x}|\boldsymbol{\beta} * \boldsymbol{y_c},\ \boldsymbol{I} - \boldsymbol{\beta} * \boldsymbol{W}) \end{aligned} \tag{1}$$

where $\boldsymbol{\beta} = \boldsymbol{W}^T * (\boldsymbol{W} * \boldsymbol{W}^T + ss * \boldsymbol{I})^{-1}$. [46] proposed an Expectation Maximization (EM) algorithm which uses the inference (1) above in the Expectation (E) step to estimate the unknown state and then choose $\boldsymbol{W}$ and the restricted ss in the Maximization (M) step so as to maximize the expected joint likelihood of the estimated $\boldsymbol{x}$ and the observed $\boldsymbol{y_c}$. We can write down corresponding E-step and M-step in matrix formulation as follows at kth iteration:

E-step: $$\boldsymbol{M} = (\boldsymbol{W^k})^T * \boldsymbol{W^k} + ss^k * \boldsymbol{I}; \tag{2}$$

E-step: $$\boldsymbol{X} = (\boldsymbol{Y} \ominus \boldsymbol{\mu}) * \boldsymbol{W^k} * \boldsymbol{M}^{-1}; \tag{3}$$

E-step: $$\boldsymbol{XtX} = \boldsymbol{X}^T * \boldsymbol{X} + N * ss^k * \boldsymbol{M}^{-1}; \tag{4}$$

E-step: $$\boldsymbol{YtX} = (\boldsymbol{Y} \ominus \boldsymbol{\mu})^T * \boldsymbol{X}; \tag{5}$$

M-step: $$\boldsymbol{W^{k+1}} = \boldsymbol{YtX} * \boldsymbol{XtX}^{-1}; \tag{6}$$

M-step: $$ss_1 = \|\boldsymbol{Y} \ominus \boldsymbol{\mu}\|_F^2; \tag{7}$$

$$ss_2 = trace(\boldsymbol{XtX} * (\boldsymbol{W^{k+1}})^T * \boldsymbol{W^{k+1}}); \tag{8}$$

$$ss_3 = \sum_{n=1}^{N} \boldsymbol{X_n} * (\boldsymbol{W^{k+1}})^T * (\boldsymbol{Y} \ominus \boldsymbol{\mu})_{\boldsymbol{n}}^T; \tag{9}$$

$$ss^{k+1} = (ss_1 + ss_2 - 2 * ss_3)/N/D \tag{10}$$

4. Related works

In this section, we give a brief description of various methods which are already implemented to compute PCA. Here we analyze their time and space complexities along with their limitations. We only consider distributed settings, and so we do not discuss any single machine implementation (for example, [47]).

4.1. MLlib-PCA

MLlib [16] is a built-in machine learning library in Spark. For PCA, MLlib computes EVD of the covariance matrix of $\boldsymbol{Y}$ and mainly provides distributed computation of large covariance matrix. We refer to its method as MLlib-PCA.

Complexity: The major computational part of MLlib-PCA is the calculation of the covariance matrix. MLlib-PCA is a deterministic algorithm and does not leverage sparsity of the matrix, so for a data matrix with size $N \times D$, it takes $\mathcal{O}(ND \times min(N, D))$ time to calculate covariance matrix. Also, it creates a large dense matrix of size $D \times D$, which it needs to store, and thus its space complexity is $\mathcal{O}(D^2)$.

Limitations: This method offers the least scalability because both N and D can be very large. For example, even for a data with vertical dimension $D = 128k$, it **faces out-of-memory error**. It is mainly because it needs to store a large intermediate data (covariance matrix) of size $\mathcal{O}(D^2)$, and it quickly faces memory overflow error as the number of dimensions grows.

4.2. Mahout-PCA

Mahout [15] is another popular library that provides the implementation of various machine learning algorithms for distributed computing. Mahout computes PCA using SSVD for big data. We refer to this as Mahout-PCA.

Complexity: In Mahout-PCA, the majority of calculation lies in the matrix multiplication $\boldsymbol{Q}^T * \boldsymbol{Y}_c$ ($m \times N$ multiplied by $N \times D$). Thus its computational complexity is $\mathcal{O}(NDm)$. The algorithm requires to store $N \times m$ dimensional matrix $\boldsymbol{Q}$. So the total intermediate data can be calculated as $\mathcal{O}(Nm)$.

Limitations: This method needs to store a large intermediate data, which is the main bottleneck. As N is large, transmitting the $N \times m$ sized intermediate data can take a very long time in communication and a lot of space in memory of each node. Furthermore, it **fails** to compute PCA on a dataset with dimension larger than $5M$ (see Table 4 for experimental outcome) as it fails to store the parameter of dimension $N \times m$.

4.3. sPCA

[17] presents a scalable implementation of Probabilistic PCA for distributed platforms, which they referred to as sPCA.

To improve the performance of the basic EM algorithm, sPCA incorporates several special features, such as (1) mean propagation to leverage sparsity, (2) minimizing intermediate data, (3) efficient matrix multiplication, (4) efficient Frobenius norm computation, etc.

Complexity: The major part of computation in sPCA is done on calculating $\boldsymbol{X}$ and ss_3 by multiplying $N \times D$ sized mean-centered data $\boldsymbol{Y_c} = (\boldsymbol{Y} \ominus \boldsymbol{\mu})$ by $D \times d$ sized parameter $\boldsymbol{W^k}/\boldsymbol{W^{k+1}}$. So, there are **two** operations which take $\mathcal{O}(NDd)$. However, sPCA preserves sparsity in calculations. Therefore, if $nnz(\boldsymbol{Y})$ represents the non-zero elements of $\boldsymbol{Y}$, the complexity would be $\mathcal{O}(nnz(\boldsymbol{Y}) \times d)$. Also, in each iteration, the parameter $\boldsymbol{W^k}$ has to be passed and stored, resulting in the space complexity to be $\mathcal{O}(Dd)$.

Limitations: By far, sPCA offers the most scalability. However, similar to the case of Mahout-PCA (by requiring to store $\mathcal{O}(Dd)$ parameter), sPCA too **faces out of memory** for higher-dimensional data. On top of that, there is no implementation of PCA on geographically distributed big data.

4.4. sSketch-PCA

sSketch-PCA [18] utilizes the SSVD technique in the computation of PCA which provides a scalable implementation of the Gaussian sketch method and then uses it for PCA. Similar to sPCA, it also uses various optimization ideas, such as (1) mean propagation for sparsity preservation, (2) effective job consolidation, and (3) on-the-fly computation which minimizes the generation of intermediate data, etc.

Complexity: In the sketching phase, in order to generate a Gaussian sketch matrix, $\boldsymbol{\Omega}$ of dimension $D \times d$, numbers are randomly selected from Gaussian distribution with *zero* mean and *unit* variance. The major part of computation in sSketch lies in the generation of the sketched data matrix $\boldsymbol{Z}$ ($D \times d$) by multiplying the sketch matrix with the mean-centered data matrix, $\boldsymbol{Y_c}$ ($N \times D$). Similar to sPCA, as sSketch incorporates sparsity preservation using mean propagation, the computation complexity is $\mathcal{O}(nnz(\boldsymbol{Y}) \times d)$, where $nnz(\boldsymbol{Y})$ denotes the non-zero entries of $\boldsymbol{Y}$. In each iteration, $\boldsymbol{Y}^T * \boldsymbol{Z}$ of dimension $D \times d$ and $\boldsymbol{Z}^T * \boldsymbol{Z}$ of dimension $d \times d$ is stored and transmitted. Therefore, the space complexity is $\mathcal{O}(Dd)$.

Limitations: sSketch-PCA is the most scalable among the techniques which used SVD to compute PCA. By using various modifications, like avoiding both reduce operation and redundant computations, use of accumulators, and preservation of sparsity of both input and intermediate data, it successfully reduces the constant factors in the running time. However, it still requires a space complexity of $\mathcal{O}(Dd)$.

Therefore, similar to the other state-of-the-art methods, as the dimension, D becomes very high, sSketch also incurs **memory overflow error**.

For more details on the complexities and limitations of these state-of-the-art techniques, interested readers are encouraged to read the survey [48].

4.5. Geo-distributed analytics and large parameter

Recently, geo-distributed analytics has gained much attention to avoid bottlenecks that are incurred for pulling all geographically distributed data to a central location. For example, [21] proposed Geode for running SQL queries efficiently in the geo-distributed environment. Similarly, to reduce query response time, a system for low latency geo-distributed analytics, namely Iridium, was proposed in [20]. To meet new challenges for running machine learning algorithms, Gaia and TernGrad was proposed in [38] and [39], respectively. Nevertheless, for PCA, to the best of our knowledge, there is no solution in geo-distributed settings. All the methods mentioned above for PCA run on aggregated data in a central DC. Additionally, to overcome scalability challenges for the large parameter in other machine learning algorithms, parameter servers have been introduced in [49,50] for single DC. However, in a geo-distributed environment, dedicating one DC as a parameter server is impractical. Therefore, in this paper, we propose a novel solution for handling large parameter for PCA in a single DC as well as in an environment of geographically distributed ones.

5. Our proposed algorithm: TallnWide

In this section, we discuss our proposed algorithm, TallnWide, in details. For a better explanation of our works, we extend the notations we have used in the previous sections. Terms and notations are as follows:

- S is the total number of DCs. d is our target dimension.
- N_s represents number of rows of the data residing in sth DC. So, the total number of rows is $N = \sum_{s=1}^{S} N_s$.

- I is the total number of divided blocks of the parameter.
- D_i represents the number of columns in the ith block of the data. So, total number of columns $D = \sum_{i=1}^{I} D_i$.
- $\boldsymbol{Y_s}$ represents the dataset in sth DC, $\boldsymbol{Y_{s,i}}$ is the ith block of the dataset residing in sth DC (size $N_s \times D_i$). $\boldsymbol{Y}$ is the overall geo-distributed data, so we have:

$$\boldsymbol{Y} = \left[\begin{array}{c|c|c} \boldsymbol{Y_{11}} & \dots & \boldsymbol{Y_{1I}} \\ \hline \dots & \dots & \dots \\ \hline \boldsymbol{Y_{S1}} & \dots & \boldsymbol{Y_{SI}} \end{array}\right]$$

We denote such **vertical concatenation by Ξ** and **horizontal concatenation by $\prod$** so that

$$\boldsymbol{Y} = \mathop{\Xi}_{s=1}^{S} \boldsymbol{Y_s} = \mathop{\Xi}_{s=1}^{S} \prod_{i=1}^{I} \boldsymbol{Y_{s,i}}$$

By **concatenation**, we do not mean any **accumulation** or **summation** (for which we use $\sum$), rather we mean stacking on top/side of each other. Here, $\boldsymbol{Y_{s,i}}$ is of $N_s \times D_i$ size.
- $\boldsymbol{\mu} = \prod_{i=1}^{I} \boldsymbol{\mu_i}$ is the mean vector of all columns.
- $\boldsymbol{W^k} = \Xi_{i=1}^{I} \boldsymbol{W_i^k}$ is the $D \times d$ size principal components, i.e. our parameter, at kth iteration. Here, $\boldsymbol{W_i^k}$ is the ith block of $\boldsymbol{W^k}$ (size $D_i \times d$) at each DC at kth iteration.
- $\boldsymbol{X} = \Xi_{s=1}^{S} \boldsymbol{X_s}$ is the $N \times d$ size latent data, where $\boldsymbol{X_s}$ (size $N_s \times d$) is the latent data at sth DC.
- In the scenario of tall and wide big data analysis, generally the data $\boldsymbol{Y}$ with dimension $N \times D$ is large and sparse and the parameter $\boldsymbol{W}$ with dimension $D \times d$ is large and dense.

5.1. Handling tall and wide big data

By *Tall and Wide Big Data*, we mean both N and D are quite large, and as $D \approx N$, parameters for PCA increase proportionately and thus they will not fit into the memory of the slave machines in a single DC. To overcome this challenge in scalability, we divide the parameter into manageable chunks/blocks to divide/distribute computations among the working nodes in order to get faster result and scale the computation. However, splitting the parameter into blocks to perform PCA is a non-trivial task. For example, when we try to isolate matrix computations of EVD for PCA, we reach a dead-end because EVD requires the $D \times D$ covariance matrix as a whole for decomposition [42]. Similarly, SSVD requires storing of $N \times m$ dimensional matrix $\boldsymbol{Q}$, and so it too has a high memory requirement. Compared to these techniques, we find that PPCA holds significant promise since it has relatively low memory footprint [48] and there is no additional decomposition involved. Second part of the challenge is to reduce steps of the algorithm in simple matrix–matrix and/or matrix–vector operations. We have to make sure that the division does not cause unnecessary overheads or bottlenecks. We also have to divide other intermediate data into blocks as well and resolve inter-dependencies. We should emphasize that due to these various challenges, we have not seen any block-division algorithm for PCA in a distributed platform.

In PPCA, our initial goal is to generate $\boldsymbol{W^k}$ first, for kth iteration. $\boldsymbol{W^k}$ has to reside at each DC as it is the central parameter for the whole algorithm. Since we only want to calculate principal components, we can ignore the noise and only run the EM algorithm with zero-noise (considering the value of noise to be zero) to refine the parameter $\boldsymbol{W^k}$. We refer to this as *zero-noise-limit PPCA*. Later, we will provide proof that zero-noise-limit PPCA indeed outputs the correct principal components. As noise is zero, we do not need (7)–(10) and steps of PPCA, (2)–(6) are changed as follows:

$$\begin{aligned}
\mathbf{E-step}:\quad & \boldsymbol{M} = (\boldsymbol{W^k})^T * \boldsymbol{W^k}; \\
& \boldsymbol{X} = (\boldsymbol{Y} \ominus \boldsymbol{\mu}) * \boldsymbol{W^k} * \boldsymbol{M}^{-1}; \\
& \boldsymbol{XtX} = \boldsymbol{X}^T * \boldsymbol{X}; \\
& \boldsymbol{YtX} = (\boldsymbol{Y} \ominus \boldsymbol{\mu})^T * \boldsymbol{X}; \\
\mathbf{M-step}:\quad & \boldsymbol{W^{k+1}} = \boldsymbol{YtX} * \boldsymbol{XtX}^{-1}
\end{aligned}$$

Now, we can easily divide the computations into smaller chunks. This proves to be significant and crucial when handling parameter of a bigger size. At a time, sth DC works with the ith block of the parameter that fits in the memory, i.e. at kth iteration, only $\boldsymbol{W_i}^{\boldsymbol{k}}$ fits. When $k = 1$, $\boldsymbol{W^1}$ is initialized randomly. Now, block-wise division of computation for $\boldsymbol{M}$ looks like this:

$$\mathbf{E-step}: \quad \boldsymbol{M} = (\boldsymbol{W^k})^T * \boldsymbol{W^k} = \sum_{i=1}^{I} (\boldsymbol{W_i^k})^T * \boldsymbol{W_i^k}$$

This calculation is illustrated as the left figure in Fig. 1. Each DC generates $(\boldsymbol{W_i^k})^T * \boldsymbol{W_i}^{\boldsymbol{k}}$ at a time, and all the results from blocks have to be added locally for getting full result $\boldsymbol{M}$. Note that each DC has the same $\boldsymbol{M}$ after this operation.

Now, we divide the computation of $\boldsymbol{X}$ as follows:

$$\begin{aligned}
\mathbf{E-step}: \quad \boldsymbol{X} &= \mathop{\Xi}_{s=1}^{S} \boldsymbol{X_s} = \mathop{\Xi}_{s=1}^{S} (\boldsymbol{Y_s} \ominus \boldsymbol{\mu}) * \boldsymbol{W^k} * \boldsymbol{M}^{-1} \\
&= \Big(\mathop{\Xi}_{s=1}^{S} \sum_{i=1}^{I} \big(\boldsymbol{Y_{s,i}} * \boldsymbol{W_i}^{\boldsymbol{k}} \ominus \boldsymbol{\mu_i} * \boldsymbol{W_i}^{\boldsymbol{k}} \big) \Big) * \boldsymbol{M}^{-1} \\
&= \Big(\mathop{\Xi}_{s=1}^{S} \sum_{i=1}^{I} \boldsymbol{Z_{s,i}} \Big) * \boldsymbol{M}^{-1} = (\mathop{\Xi}_{s=1}^{S} \boldsymbol{Z_s}) * \boldsymbol{M}^{-1}
\end{aligned}$$

where, $\boldsymbol{Z_{s,i}} = \big(\boldsymbol{Y_{s,i}} * \boldsymbol{W_i}^{\boldsymbol{k}} \ominus \boldsymbol{\mu_i} * \boldsymbol{W_i}^{\boldsymbol{k}}\big)$ and $\boldsymbol{Z_s} = \big(\boldsymbol{Y_s} * \boldsymbol{W^k} \ominus \boldsymbol{\mu} * \boldsymbol{W^k}\big)$.

Computation of $\boldsymbol{Z_s}$ is illustrated as the middle figure in Fig. 1. Interestingly, we need not form full $\boldsymbol{X}$, we need only $\boldsymbol{Z_s}$ at each DC and multiply it by $\boldsymbol{M}^{-1}$ (which each DC already has) for computing $\boldsymbol{XtX}$ (illustrated as the right figure in Fig. 1):

$$\begin{aligned}
\mathbf{E-step}: \quad \boldsymbol{XtX} &= \boldsymbol{X}^T * \boldsymbol{X} \\
&= \Big(\mathop{\Xi}_{s=1}^{S} \boldsymbol{X_s}^T \Big) * \Big(\mathop{\Xi}_{s=1}^{S} \boldsymbol{X_s} \Big) \\
&= \sum_{s=1}^{S} \boldsymbol{X_s}^T * \boldsymbol{X_s} \\
&= \boldsymbol{M}^{-1} * \Big(\sum_{s=1}^{S} \boldsymbol{Z_s}^T * \boldsymbol{Z_s} \Big) * \boldsymbol{M}^{-1}
\end{aligned}$$

For the last stage of the expectation, we need not form $\boldsymbol{YtX}$ explicitly. Instead, we can go directly to the maximization stage and derive the parameter:

$$\begin{aligned}
\mathbf{M-step}: \ & \boldsymbol{W^{k+1}} \\
&= \Big(\big(\mathop{\Xi}_{s=1}^{S} (\boldsymbol{Y_s} \ominus \boldsymbol{\mu}) \big)^T * \big(\mathop{\Xi}_{s=1}^{S} \boldsymbol{X_s} \big) \Big) * \boldsymbol{XtX}^{-1} \\
&= \Big(\big(\mathop{\Xi}_{s=1}^{S} \prod_{i=1}^{I} (\boldsymbol{Y_{s,i}} \ominus \boldsymbol{\mu_i}) \big)^T * \big(\mathop{\Xi}_{s=1}^{S} \boldsymbol{Z_s} \big) \Big) * \boldsymbol{M}^{-1} * \boldsymbol{XtX}^{-1} \\
&= \sum_{s=1}^{S} \Big(\mathop{\Xi}_{i=1}^{I} (\boldsymbol{Y_{s,i}^T} * \boldsymbol{Z_s} * \boldsymbol{MXtX} - \boldsymbol{\mu_i^T} * \boldsymbol{z_s} * \boldsymbol{MXtX}) \Big)
\end{aligned}$$

where vector $\boldsymbol{z}_s$ denotes the sum of all rows of $\boldsymbol{Z}_s$ and $\boldsymbol{MXtX} = \boldsymbol{M}^{-1} * \boldsymbol{XtX}^{-1}$. Notice that, since subtracting the mean vector from the data matrix results in creating a dense matrix, at each step, we consider operations with mean vector $\boldsymbol{\mu}$ separately for

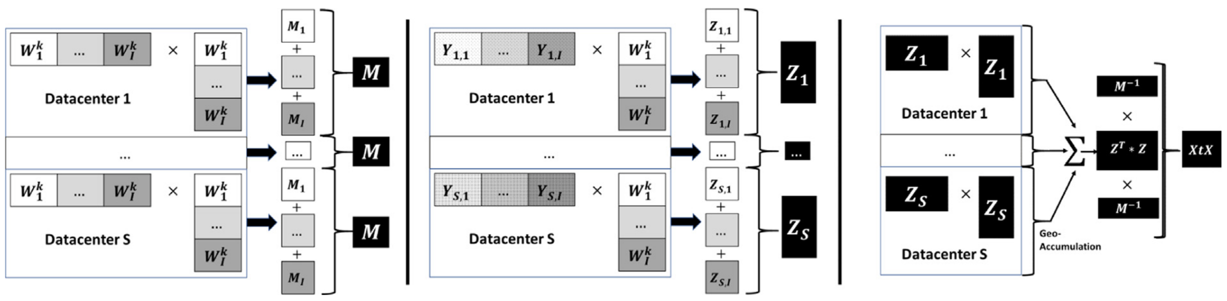


Fig. 1. Generation of $\boldsymbol{M}$ (left), $\boldsymbol{Z}$ (middle), and $\boldsymbol{XtX}$ (right) in parallel fashion among all DCs. For simplicity, operations for mean vector, $\boldsymbol{\mu}$, are not shown in the middle figure.

preserving sparsity in the input matrix $\boldsymbol{Y}$. We refer to this as *mean propagation* [48].

5.2. Communication efficient calculation

The order at which we perform the matrix operations for our block-division plays a significant role in communication efficiency. In the block-division method, the partial results generated at each DC will be gathered by a central DC, and the aggregated results will be transmitted back to all other DCs. We refer to this process as **Geo-Accumulation** or simply **accumulation**. We now present various approaches of geo-accumulation to transfer our intermediate results.

Approach 1: Trivial Order. At the current order of computing $\boldsymbol{W^{k+1}}$ mentioned in the **M Step**, the following happens:

$$\boldsymbol{W^{k+1}} = \underbrace{\sum_{s=1}^{S}\Big(\overbrace{\mathop{\Xi}_{i=1}^{I}(\boldsymbol{Y}_{s,i}^{T} * \boldsymbol{Z_s} * \boldsymbol{MXtX} - \boldsymbol{\mu}_i^{T} * \boldsymbol{z_s} * \boldsymbol{MXtX})}^{\text{Partial results of all blocks from all DCs}}\Big)}_{\text{Geo-Accumulation of partial results}} \tag{11}$$

As $\boldsymbol{W^k}$ has been horizontally partitioned into I blocks, in this order, each DC will generate partial results for each block. After that, all the partial results from every DC will be accumulated by the central DC and multiplied by $\boldsymbol{MXtX}$ to get the final result $\boldsymbol{W^{k+1}}$. In the current order, raw data need not be transmitted, but it creates a tremendous amount of idle time in worker nodes. To see why let us imagine the case for a single DC. As at a time only one block fits into the memory, the worker nodes of a single DC produces partial results for $i = 1$ first, and save it to the secondary storage, for making room for next segments (i.e., $i = 2$) in memory. In this fashion, we first have to generate (incurring CPU time) and store (incurring Disk I/O) all partial results for each block in each DC. When it is finished, we have to retrieve (which has a Disk I/O) each block from each DC for accumulation (which has a Network I/O) by the central DC. After accumulation, the aggregated result of each block will be transmitted back to each DC and stored again. This is shown as Approach 1 in Fig. 2.

Approach 2: Efficient Order. Now, we consider the reversed order of the computation, as mentioned in (11) above. The reversed order follows:

$$\boldsymbol{W^{k+1}} = \underbrace{\mathop{\Xi}_{i=1}^{I}\Big(\overbrace{\sum_{s=1}^{S}(\boldsymbol{Y}_{s,i}^{T} * \boldsymbol{Z_s} * \boldsymbol{MXtX} - \boldsymbol{\mu}_i^{T} * \boldsymbol{z_s} * \boldsymbol{MXtX})}^{\text{Geo-Accumulation for full result ofith block}}\Big)}_{\text{Full result for all blocks}} \tag{12}$$

In this order, we can accumulate partial results for each block from each DC after they are generated. While we accumulate for one block, we can start the calculation for the next block because

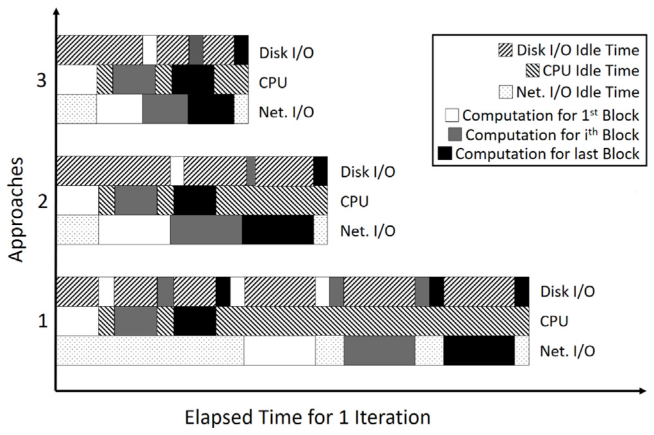


Fig. 2. Comparison of idle time among different approaches. Approach 1: Trivial Order, Approach 2: Efficient Order, and Approach 3: Efficient Order w/ Ideal Central DC.

there is no dependency. This is shown as Approach 2 in Fig. 2. In contrast to Approach 1 from Fig. 2, we can see that Approach 2 reduces the idle time significantly. This kind of highly efficient ordering technique which enables parallelism in computation and transmission is known as Grouped Aggregate Pushdown [51] and very popular in the database research community.

Approach 3: Efficient Order w/ Ideal Central DC. In addition to Approach 2, as every DC has to send their partial results to a central DC for accumulation, we should choose such a DC as the central one for which the slowest link is maximum among all the candidate DCs. Mathematically, let us first assume that we have a symmetric B/W matrix $\boldsymbol{B}$ where each cell $\boldsymbol{B_{uv}}$ denotes B/W between DC u and DC v ($u, v \in \{1, \dots, S\}$). For every DC u, we first determine the slowest B/W link from its connections to all other DCs v ($v \neq u,\ v \in \{1, \dots, S\}$). Then from a set of such B/Ws for every DC u, we can select the DC for which the following is true:

$$\underset{u \in \{1,\dots,S\}}{\operatorname{argmax}}\Big(\min\{\boldsymbol{B_{uv}} \mid v \neq u,\ v \in \{1, \dots, S\}\}\Big) \tag{13}$$

If there are multiple DCs, we can select any of them. We refer this selected central DC for accumulation as ideal central DC. This is shown as Approach 3 in Fig. 2, and theoretically by using this ideal central DC the time required to collect the partial results from other DCs and the redistribution task should be the minimum, and thus it offers the fastest accumulation. Later, we will validate the merit of this final approach through our experiment.

5.3. Validation of zero-noise-limit probabilistic PCA

In this section, we validate zero-noise-limit PPCA over the conventional PPCA according to [25]. We first show that zero-noise-limit PPCA produces the correct principal components.

Lemma 5.1. *PCA is a limiting case of the linear-Gaussian model in (1) as the covariance of the noise $\boldsymbol{\sigma}$ becomes infinitesimally small and equal in all directions, i.e. $\boldsymbol{\sigma} = \lim_{ss\to 0} ss * \boldsymbol{I}$.*

Proof. For $\boldsymbol{\sigma} = \lim_{ss\to 0} ss * \boldsymbol{I}$, inference (1) becomes:

$$P(\boldsymbol{x}|\boldsymbol{y_c}) = \mathcal{N}(\boldsymbol{x}|\boldsymbol{\beta} * \boldsymbol{y_c},\ \boldsymbol{I} - \boldsymbol{\beta} * \boldsymbol{W});$$

$$\boldsymbol{\beta} = \lim_{ss\to 0} \boldsymbol{W}^T * (\boldsymbol{W} * \boldsymbol{W}^T + ss * \boldsymbol{I})^{-1}$$

$$P(\boldsymbol{x}|\boldsymbol{y_c}) = \mathcal{N}(\boldsymbol{x}|(\boldsymbol{W}^T * \boldsymbol{W})^{-1} * \boldsymbol{W}^T * \boldsymbol{y_c},\ 0)$$

$$= \delta(\boldsymbol{x} - (\boldsymbol{W}^T * \boldsymbol{W})^{-1} * \boldsymbol{W}^T * \boldsymbol{y_c})$$

Since the noise has become infinitesimal, the posterior over states collapses to a single point, and the covariance becomes zero. This has the effect of making the likelihood of a point $\boldsymbol{y_c}$ dominated solely by the squared distance between it and its reconstruction $\boldsymbol{W} * \boldsymbol{x}$. But the directions of the columns of a matrix which minimize this error are known as the *principal components*. As columns of $\boldsymbol{W}$ have this property, we have our desired output. □

This version of PPCA is what we have referred to as *zero-noise-limit PPCA*. And we just have designed the block-division EM-algorithm for this version. For more details and correctness of the EM-steps, interested readers are encouraged to read [25].

This zero-noise-limit PPCA has significant advantage over conventional PPCA. To explain why, let us consider the scenario where we calculate noise ss in our steps according to Eqs. (7)–(10). For ss, we have to calculate ss_1 (which needs to be calculated only once), ss_2 and ss_3. Among them, the calculation of ss_3 is the most complicated:

$$\boldsymbol{ss_3} = \sum_{n=1}^{N}\sum_{s=1}^{S}\Big(\boldsymbol{X}_{\boldsymbol{s,n}}^T * (\prod_{i=1}^{I}(\boldsymbol{W}_{\boldsymbol{i}}^{k+1})^T * \mathop{\boldsymbol{\Xi}}_{i=1}^{I}(\boldsymbol{Y}_{\boldsymbol{s,i,n}}^T \ominus \boldsymbol{\mu}_{\boldsymbol{i}}^T))\Big)$$

We can see that the calculation of ss_3 in kth iteration is dependent on the calculation of entirely accumulated updated parameter for the next iteration, i.e., $\boldsymbol{W^{k+1}}$. This unnecessarily complicates the computation and introduces inter-dependency between steps and reduces the scope for maximization of parallelism. To illustrate, let us contrast the dependency diagram of our zero-noise-limit PPCA and the conventional PPCA. From Fig. 3, it is evident that conventional PPCA has additional complications while zero-noise-limit PPCA requires simple and fewer steps. With this validation, we conclude our overall design for the algorithm TallnWide.

6. Algorithm and complexity analysis

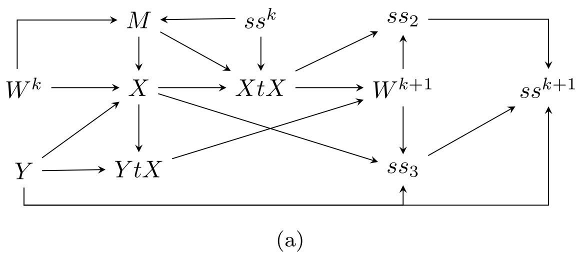
In this section, firstly, we give a detailed description of our two main algorithms, namely TallnWide, and Geo-Accumulation, which are shown in Algorithm 1, and 2, respectively. Algorithm TallnWide shows the necessary steps of performing PCA on tall and wide big data in a geographically distributed environment. On the other hand, Algorithm Geo-Accumulation performs the distributed accumulation task using the detected ideal DC. It should be noted that TallnWide algorithm can be run in two different modes: (i) On a single cluster by setting the number of servers s to zero. We refer to this mode of execution as the *standalone mode*. (ii) On geographically distributed clusters where each datacenter holds its own version of data. TallnWide is capable of handling tall and wide big data in both standalone and in geo-distributed modes. The computational, communication, and space complexities are analyzed in the latter part of this section.

Algorithm 1: TallnWide

```
Input : Data matrix $\boldsymbol{Y}$ of dimension $N \times D$,
        target dimension $d$,
        number of servers $S$,
        B/W matrix $\boldsymbol{B}$
Output: Principal Component $\boldsymbol{W}$
1  $\boldsymbol{W^1}$ = normRand($D, d$)                                  ▹ $\mathcal{O}_c(1)$
2  $iDC$ = detIdealDC($\boldsymbol{B}$)
3  $I$ = detNumBlock( )
4  $\boldsymbol{\mu}$ = mean($\boldsymbol{Y}$)                                  ▹ $\mathcal{O}_c(D)$
5  $\boldsymbol{M}$ = $newMatrix(d, d)$
6  $\boldsymbol{Z}_s$ = $newMatrix(N_s, d)$
7  for each iteration $k \leftarrow 1$ to $K$ do
8    for each DC $s \in \{1, \ldots, S\}$ do
9      for each block $i \leftarrow 1$ to $I$ do
10       $\boldsymbol{W_i^k} = load(\boldsymbol{W^k}, i)$
11       $\boldsymbol{M} \mathrel{+}= (\boldsymbol{W_i^k})^T * \boldsymbol{W_i^k}$                ▹ $\mathcal{O}_t(D_i d^2)$
12       $\boldsymbol{Z_m} = \boldsymbol{\mu_i} * \boldsymbol{W_i^k}$
13       $Z_s \mathrel{+}= \boldsymbol{Y_{s,i}} * \boldsymbol{W_i^k} \ominus \boldsymbol{Z_m}$           ▹ $\mathcal{O}_t(nnz(\boldsymbol{Y_{s,i}})d)$
14     end
15     $\boldsymbol{ZtZ}$ = Geo-Accumulation($\boldsymbol{Z_s}^T * \boldsymbol{Z_s}, iDC$)     ▹ $\mathcal{O}_c(d^2)$
16     $\boldsymbol{MXtX} = \boldsymbol{M}^{-1} * (\boldsymbol{M}^{-1} * \boldsymbol{ZtZ} * \boldsymbol{M}^{-1})^{-1}$       ▹ $\mathcal{O}_t(d^2)$
17   end
18   for each block $i \leftarrow 1$ to $I$ do
19     for each DC $s \in \{1, \ldots, S\}$ do
20       $\boldsymbol{W_{s,i}^{k+1}} = \boldsymbol{Y_{s,i}}^T * \boldsymbol{Z_s} * \boldsymbol{MXtX} - \boldsymbol{\mu_i}^T * \boldsymbol{z_s} * \boldsymbol{MXtX}$   ▹ $\mathcal{O}_t(nnz(\boldsymbol{Y_{s,i}})d)$
21       $\boldsymbol{W_i^{k+1}}$ = Geo-Accumulation($\boldsymbol{W_{s,i}^{k+1}}, iDC$)   ▹ $\mathcal{O}_c(D_i d)$ and$\mathcal{O}_s(D_i d)$
22     end
23   end
24   if $converged$ then
25     $\boldsymbol{W} = \boldsymbol{W^{k+1}}$
26     $stop$
27   end
28 end
29 return $\boldsymbol{W}$
```

6.1. Algorithm description

We summarize the basic steps of TallnWide in Algorithm 1. To reduce communication time for the first iteration, we only send the seed for a random number generator so that each DC can generate the same $\boldsymbol{W^1}$ locally. In Line 1, "normRand(D,d)" produces random Gaussian matrix of size $D \times d$ with the given seed. In Line 2 and 3, "detIdealDC(B)" and "detNumBlock( )" respectively determines the ideal central DC "iDC" from B/W matrix $\boldsymbol{B}$ and number of blocks I. The steps of determining the ideal central DC are designed according to Eq. (13). In the next line, "mean($\boldsymbol{Y}$)" outputs the mean matrix $\boldsymbol{\mu}$. Line 5 and 6 respectively initialize the two matrices $\boldsymbol{M}$ (size $d \times d$) and $\boldsymbol{Z_s}$ (size $N_s \times d$) with zero values. "load($\boldsymbol{W^k}$, i)" loads ith block from $\boldsymbol{W^k}$ in Line 10. We derive necessary intermediate data, namely $\boldsymbol{M}$, $\boldsymbol{Z}$, $\boldsymbol{ZtZ}$ and $\boldsymbol{MXtX}$ (see Fig. 3(b)), for calculating $\boldsymbol{W^{k+1}}$ in Line 11, 13, 15 and 16. Line 18 to Line 23 refer to necessary operations for (12). "Geo-Accumulation(Matrix $\boldsymbol{A}$, iDC)" performs geo-distributed accumulation of any data matrix $\boldsymbol{A}$ using "iDC" in Line 15 and 21. Finally, we check convergence in Line 24. It should be mentioned that this TallnWide algorithm shows the steps done in a generalized platform-independent environment. However, we describe the detailed computation of two of our significant steps in Spark distributed environment in the following section. Algorithm 2 shows the process of accumulating partial results in details. If the DC itself is the ideal central DC, then it accumulates



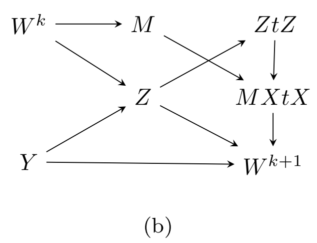


Fig. 3. Matrix dependency diagram of two variations: (a) conventional PPCA (implemented in sPCA) and (b) zero-noise-limit PPCA (implemented in TallnWide).

Algorithm 2: Geo-Accumulation

```
Input : Partial result $\boldsymbol{A}$ in Matrix form,
        $iDC$ which denotes the ID of Ideal DC
1  $myID$ = loadID( )
2  if ($myID == iDC$) then
3    $DClist = \{1, \dots, S\} - \{myID\}$
4    while (not done for every $s \in DClist$) do
5      if $notifiedFrom(s)$ then
6        addNewProcess($\boldsymbol{A} = \boldsymbol{A} + \boldsymbol{A_s}$)
7        remove $s$ from $DClist$
8      end
9    end
10   for each DC $s \in DClist$ do
11     sendData($\boldsymbol{A}$, $s$)
12   end
13 end
14 else
15   sendData($\boldsymbol{A}$, $iDC$)
16   notifyMaster($iDC$)
17 end
```

all the partial results that are already received from the other DCs. The "notifiedFrom(s)" function determines whether any **Done** notification is received from DC s or not. However, upon receiving $\boldsymbol{A_s}$ (the partial result of full data $\boldsymbol{A}$ in matrix form received from any DC s), the ideal DC starts a new process to accumulate it and remove s from the consideration list. At the end of the accumulation process, the ideal central DC redistributes the final result to each of the DCs. On the other hand, if the DC is not the ideal central DC itself, then it concludes its accumulation task by sending the partial result $\boldsymbol{A}$ which is to be accumulated to the central DC and by notifying (sending a **Done** notification) it about the task.

6.2. Complexity analysis

We denote time, space and communication complexities by $\mathcal{O}_t$, $\mathcal{O}_s$, and $\mathcal{O}_c$, respectively. Now we show the line-wise complexity in Algorithm 1. Since we are only sending the seed for a random number generator, Line 1 has a communication complexity of $\mathcal{O}_c(1)$. Line 4 sends the partial mean value at each datacenter, and the aggregated mean generation demands the use of geo-distributed bandwidth which has the communication complexity of $\mathcal{O}_c(D)$. Both $\boldsymbol{M}$ and $\boldsymbol{MXtX}$ are generated redundantly in each of the datacenters with computational complexities of $\mathcal{O}_t(D_id^2)$ and $\mathcal{O}_t(d^2)$, respectively.

However, only major computation in each iteration is the full calculation of $\boldsymbol{Y}{*}\boldsymbol{W}$ and $\boldsymbol{Y}^T{*}\boldsymbol{Z}$ (aggregated result from Line 13 and Line 20) which takes $\mathcal{O}_t(nnz(Y)\times d)$ time (we **preserve sparsity** in multiplication in Line 13 and 20 by mean propagation). Similarly, in each iteration, only major data for storing is the block of the PCA parameter, i.e. $\boldsymbol{W}_i$ (size $D_i \times d$) in Line 21. So the space complexity is $\mathcal{O}_s(D_id)$. Note that $D_i < D$. Finally, accumulation

Table 1
Comparison of complexities among all methods. Since we divide the parameters into blocks, our space complexity is less than others. Also, despite having the same time complexity w.r.t. sPCA and sSketch-PCA, we reduce the constant factors in running time of TallnWide significantly.

Method	Time complexity	Space complexity
MLlib-PCA	$\mathcal{O}_t(ND \times min(N, D))$	$\mathcal{O}_s(D^2)$
Mahout-PCA	$\mathcal{O}_t(NDm)$	$\mathcal{O}_s(Nm)$
sPCA	$\mathcal{O}_t(nnz(\boldsymbol{Y}) \times d)$	$\mathcal{O}_s(Dd)$
sSketch-PCA	$\mathcal{O}_t(nnz(\boldsymbol{Y}) \times d)$	$\mathcal{O}_s(Dd)$
TallnWide	$\mathcal{O}_t(nnz(\boldsymbol{Y}) \times d)$	$\mathcal{O}_s(D_id)$

time of $\boldsymbol{W}^{k+1}$ in Line 21 dominates others, so communication complexity is $\mathcal{O}_c(D_id)$. As other methods are not geo-distributed, we only compare time and space complexities of TallnWide in Table 1. Note that, as we do not calculate noise, we reduce the constant factors in computation time by a significant fraction. Also, because of block-division, our space complexity is less than others, and consequently, we do not face the out-of-memory error.

7. Experimental design

In this section, first we provide a detailed discussion of Spark implementation of our algorithm. After that, we discuss the setup that we use to run our experiments. In the following section, we discuss the experimental outcome.

7.1. Spark implementation

The main goal of our algorithm is to calculate PCA in a geographically distributed setup. To achieve a distributed environment, we consider the distributed framework Spark [26]. In this section, we show spark implementation based computation steps to generate necessary intermediate data during the lifetime of our main TallnWide algorithm (Algorithm 1).

Algorithm 3 shows how we achieve the task of generating the intermediate data $\boldsymbol{Z}$ (Line 13 from Algorithm 1) in Spark environment. In Line 1, the *zip* operation combines the data matrix $\boldsymbol{Y}$ with the intermediate matrix $\boldsymbol{Z}$ of the previous iteration (in case of the first iteration, the input $\boldsymbol{Z}$ is the zero-filled initial matrix). The map operation in Line 3, which runs for each ith row of $\boldsymbol{Y_nZ}$ can be carried out in parallel by each of the worker nodes. For each row of $\boldsymbol{Y_nZ}$, in Line 4, the temporary variable Y_i retrieves the corresponding row from data matrix $\boldsymbol{Y}$. Only those values that fall within the horizontal dimension range from *start* to *end* are extracted. On the other hand, at Line 5, the temporary variable Z_i retrieves the corresponding row from the previous intermediate data matrix $\boldsymbol{Z}$.

After that, Y_i is multiplied by the principal subspace matrix $\boldsymbol{W^k}$ of the previous iteration to generate a quantity *dotRes*. Finally, in Line 7, this *dotRes* is added to the variable Z_i, while the corresponding row from $\boldsymbol{Z_m}$ is subtracted. When all the worker

Algorithm 3: SegmentedZJob

```
Input  : Principal Subspace Matrix $\boldsymbol{W^k}$;
         Data Matrix $\boldsymbol{Y}$;
         start which denotes the starting Index for Block I;
         end which denotes the ending Index for Block I;
         Intermediate Matrix $\boldsymbol{Z}$ from previous iteration;
         $\boldsymbol{Z_m}$ which denotes the mean($\boldsymbol{Y}$) multiplied by $\boldsymbol{W}$
Output: Updated Intermediate Matrix $\boldsymbol{Z}$
1  $\boldsymbol{Y_nZ} = \boldsymbol{Y}.zip(\boldsymbol{Z})$
2  $\boldsymbol{ZSum} = accumultor(newMatrix(N_s, d))$
3  $\boldsymbol{Y_nZ}.map\{(Y_nZ)_i \Rightarrow$            ▷ Runs for each $i^{th}$ row of $\boldsymbol{Y_nZ}$;
4      $Y_i = (Y_nZ)_i.arg0().range(start, end)$
5      $Z_i = (Y_nZ)_i.arg1()$
6      $dotRes = Y_i \times \boldsymbol{W^k}$
7      $\boldsymbol{ZSum}.add(Z_i + dotRes - (Z_m)_i)$
8  }
9  $Z = \boldsymbol{ZSum}.value()$
10 return $\boldsymbol{Z}$
```

nodes are done with the map operation, a new version of the intermediate data $\boldsymbol{Z}$ is generated and returned.

Generation of our parameter $\boldsymbol{W}$ (Line 20 from Algorithm 1) can be divided into two parts: computing $(\boldsymbol{Y}_{s,i}^T * \boldsymbol{Z_s} - \boldsymbol{\mu_i}^T * \boldsymbol{z_s})$ which we refer to as generating $\boldsymbol{YtZ}$ and multiplying it with previously generated $\boldsymbol{MXtX}$. Algorithm 4 provides the mechanism of generating $\boldsymbol{YtZ}$ and $\boldsymbol{ZtZ}$ (a part of $\boldsymbol{MXtX}$) in Spark environment. Similar to Algorithm 3, here the map operation is run for each ith row of $\boldsymbol{Y_nZ}$ and the operation is carried out by each of the worker nodes for their corresponding data segment which has been distributed by Spark. Note that, $\boldsymbol{ZtZ}$ is generated only for the first iteration while new $\boldsymbol{YtZ}$ is generated at every iteration.

Algorithm 4: SegmentedYtZnZtZJob

```
Input  : Iteration number $k$;
         Data Matrix $\boldsymbol{Y}$;
         $\boldsymbol{\mu}$ which denotes mean($\boldsymbol{Y}$);
         $D_i$ which denotes the No of Rows in $i^{th}$ segment of $\boldsymbol{W}$;
         Target Dimension $d$;
         Intermediate Matrix $\boldsymbol{Z}$;
         start which denoted the starting Index for Block I;
         end which denotes the ending Index for Block I
Output: Intermediate Data $\boldsymbol{ZtZ}$;
         Intermediate Data $\boldsymbol{YtZ}$
1  $\boldsymbol{Y_nZ} = \boldsymbol{Y}.zip(\boldsymbol{Z})$
2  if $(k == 1)$                         ▷ Iteration for the $1^{st}$ Segment of W
3  then
4      $\boldsymbol{YtZSum} = accumultor(newMatrix(D_i, d))$
5      $\boldsymbol{ZtZSum} = accumultor(newMatrix(D_i, d))$
6      $\boldsymbol{Y_nZ}.map\{(Y_nZ)_i \Rightarrow$         ▷ Runs for each $i^{th}$ row of $\boldsymbol{Y_nZ}$
7          $Z_i = (Y_nZ)_i.arg1().range(start, end)$
8          $(YtZ)_i = Y_i^T \times Z_i - \mu^T \times Z_i$
9          $(ZtZ)_i = Z_i^T \times Z_i$
10         $\boldsymbol{YtZSum}.add((YtZ)_i)$
11         $\boldsymbol{ZtZSum}.add((ZtZ)_i)$
12     }
13     $\boldsymbol{YtZ} = \boldsymbol{YtZSum}.value()$
14     $\boldsymbol{ZtZ} = \boldsymbol{ZtZSum}.value()$
15     return $\boldsymbol{YtZ}, \boldsymbol{ZtZ}$
16 end
17 else
18     $\boldsymbol{YtZSum} = accumultor(newMatrix(D_i, d))$
19     $\boldsymbol{Y_nZ}.map\{(Y_nZ)_i \Rightarrow$         ▷ Runs for each $i^{th}$ row of $\boldsymbol{Y_nZ}$
20         $Y_i = (Y_nZ)_i.arg0().range(start, end)$
21         $Z_i = (Y_nZ)_i.arg1()$
22         $(YtZ)_i = Y_i^T \times Z_i - \mu^T \times Z_i$
23         $\boldsymbol{YtZSum}.add((YtZ)_i)$
24     }
25     $\boldsymbol{YtZ} = \boldsymbol{YtZSum}.value()$
26     return $\boldsymbol{YtZ}$
27 end
```

7.2. Algorithms compared

For computing principal components, we compare five methods:

- TallnWide: Our implementation of the algorithm TallnWide.
- sPCA: A scalable implementation of PPCA [17].
- Mahout-PCA: Mahout implementation of PCA [15].
- MLlib-PCA: PCA implementation in MLlib [16].
- sSketch-PCA: PCA implementation using a scalable sketching technique [18].

The implementations of Mahout-PCA, as well as MLlib-PCA, are highly optimized in their respective platforms. On the other hand, the sPCA implementation on Spark has added some advanced features and has been quite successful in outperforming most of the close competitors. Finally, sSketch-PCA utilizes a scalable implementation of Gaussian sketching method along with various optimization techniques in order to achieve high scalability. Therefore, we find the algorithms as mentioned above to be the best options for comparing with TallnWide to establish its merit. For uniform comparison, we make all PCA algorithms to compute the top 10 principal components.

7.3. Datasets

We use four real datasets. All of them are quite diverse in terms of size, dimensions, sparsity, and data values. Moreover, we generate various subsets of the four datasets so that we can assess the scalability and performance of comparing PCA algorithms with increasing data sizes. The following list describes the datasets by showing their source, dimension and sparsity. If the number of rows and columns of a dataset are N and D respectively, then the sparsity of that dataset is measured as follows: sparsity = # of zero elements/$(N * D)$.

- PubMed: PubMed is a sparse, bag-of-word representation of medical documents from the U.S. National Library of Medicine. [3] provides a matrix from this dataset where the rows represent the documents, and the columns represent the words (values are either 0 or 1). Its size is $8,200,000 \times 141,043$. The number of non zero elements in this dataset is 483,450,157, and the sparsity is 0.9995. We also consider a small subset of this dataset with a dimension of 2000, referred to as PubMed2K.
- AmazonRating: Amazon product data provided by [2] contains product reviews and meta-data from Amazon. It is a sparse matrix of size $21M \times 9.8M$, and the values are between 0 and 5. The value at the cell $[i][j]$ represents the rating given by the user i to the product j. The number of non zero elements in this dataset is 82,676,840, and the sparsity is 0.9999. We also consider two subsets: AmazonRating2K (size $6.6M \times 2K$) and AmazonRating50K (size $6.6M \times 50K$).
- SiftFeature: We download images from ImageNet [52] and from each image, we extract SIFT (Scale-Invariant Feature Transform) [53] features. It is a dense matrix of size 4455091 $\times$ 128 (with sparsity value of 0.2272), and each element is a real value.
- Twitter: We take a $50M \times 50M$ dataset from [1] which is a sparse matrix (with sparsity value of 0.9999) of social

Table 2
Bandwidth (in MB/s) among geo-distributed DCs.

DC/DC	Ireland	N. Virginia 1	N. Virginia 2	Oregon 1	Oregon 2
Ireland	–	11.35 MB/s	10.95 MB/s	9.25 MB/s	11.60 MB/s
N. Virginia 1	11.35 MB/s	–	81.30 MB/s	18.05 MB/s	18.20 MB/s
N. Virginia 2	10.95 MB/s	81.30 MB/s	–	17.50 MB/s	17.70 MB/s
Oregon 1	9.25 MB/s	18.05 MB/s	17.5 MB/s	–	126.95 MB/s
Oregon 2	11.60 MB/s	18.20 MB/s	17.70 MB/s	126.95 MB/s	–

Table 3
Effect of ρ on the choice of different number of blocks.

	AmazonRating			Twitter10M		
ρ	Number of blocks	Block size (MB)	Time per iteration (s)	Number of blocks	Block size (MB)	Time per iteration (s)
10	1	2918	**Failed**	2	2213	**Failed**
30	2	1628	591.70	3	1811	597.54
40	3	1105	742.47	4	1425	877.31
60	4	917	1444.22	5	1007	1635.71

network (values are 0 and 1) of Twitter users. From this dataset, we primarily consider two subsets: Twitter1K (size $59670 \times 1K$) and Twitter10M (size $10M \times 10M$). We also make several other low dimensional subsets as needed (not mentioned as separate names).

7.4. Cluster configuration

We run our TallnWide algorithm in two different environments: (i) firstly, in a single cluster (*standalone mode*) to assess the capability of handling arbitrarily large dimensional datasets; (ii) secondly, in geo-distributed clusters where partial results generated by each cluster need to be aggregated and redistributed. The first experiment mainly focuses on computational capability, which can provide efficient memory utilization. On the other hand, the later one needs efficient inter datacenter communication scheme. In this subsection, we describe the cluster configuration to create the above mentioned two environments.

Single DC Configuration for Standalone Mode: We run the experiments for a single DC (*standalone mode*) by creating a cluster on the Amazon EMR. Our cluster consists of 8 Amazon EC2 m3.xlarge instances, each of which contains eight vCPU, 15 GB of memory, and 80 GB SSD storage. Each node has the following software installed: emr-5.7.0, Hadoop 2.7.3, Spark 2.1.1, Ganglia 3.7.2, and Mahout 0.13.0. MLlib library is included in Spark.

Geo-distributed DCs Configuration: For simulating geo-distributed environment, we consider three geographical regions, namely North Virginia, Oregon, and Ireland. In North Virginia and Oregon, we consider two separate zones, which are referred to as North Virginia 1 & 2, and Oregon 1 & 2. In Ireland, we only consider one zone referred to as Ireland. That means we consider five different geographic locations. Table 2 shows the inter-DC B/Ws. In each zone, to create a cluster, we take 3 Amazon EC2 m3.2xlarge instances each of them having 16 vCPU, 30 GB of memory, and 160GB SSD and install following software: Hadoop 2.6, Spark 2.0.0, and Ganglia 3.7.2. Depending on the implementation, we deploy different kinds of geo-distributed clusters on these 15 instances (described later).

7.5. Performance metrics

In case of *standalone mode* evaluation, we consider three performance metrics: running time to achieve the desired accuracy, scalability, and intermediate data size. In addition, we also show the effect of the number of blocks on the running time and scalability. For checking convergence, we use the following criteria:

$$\Delta \boldsymbol{W} = \max\Big(|\boldsymbol{W^k} - \boldsymbol{W^{k+1}}|/\big(\epsilon + \max(|\boldsymbol{W^{k+1}}|)\big)\Big)$$

where $\boldsymbol{W}$ is our principal component. Here $|.|$ denotes the absolute value of the term, and max returns the maximum element of a matrix. When we say 5% tolerance for convergence, we mean changes in elements of $\boldsymbol{W}$ or $\Delta\boldsymbol{W}$ is less than or equal to 5%. Here ϵ is a small number to avoid any division-by-zero error. On the other hand, in case of geo-distributed evaluation, we mainly focus on communication efficiency. We show the changes in running time in our various accumulation strategies. Then we show the communication efficiency among different approaches mentioned in Section 5.2.

8. Experimental evaluation

In this section, we present an in-depth comparison of TallnWide with other algorithms based on the performance metrics mentioned in the preceding section.

8.1. Evaluation for a single DC

We first show the efficiency of our block-division method in TallnWide algorithm for a single DC. We show that TallnWide is both fast and scalable compared to the other methods.

Effect of Number of Blocks: In our implementation, we do not fix the number of blocks for the division. Instead, we dynamically choose the number of blocks by this formula:

$$I = \lceil (\rho * D * d * 8)/(minMem * 1024^2) \rceil$$

where I is the number of blocks, ρ is the tuning parameter, *minMem* is the least free memory (in MB) among working nodes. Table 3 shows the effect of ρ on the number of blocks for AmazonRating and Twitter10M datasets. Depending on ρ, the number of blocks varies. In this table, the sizes of the blocks are shown in approximate values. These sizes are measured during the runtime of the spark code by comparing the approximate free and utilized memories observed before and after the generation of a single block of the PCA parameter, $\boldsymbol{W}$. One point to be noted that these block sizes do not provide any direct indication of the failure or success in execution rather they directly impact the size of intermediate data generation. In order to ensure that memory overflow error does not occur, we keep the block size manageable. To do so, we keep the number of blocks to be larger than a certain threshold value. However, if we work with more blocks, we have to encounter additional overheads for Disk I/O. Therefore, our goal is to minimize the number of blocks as much

Table 4
Comparison of running time (in sec) for TallnWide on different datasets against state-of-the-art library functions: MLlib-PCA, Mahout-PCA, and sPCA. For iterative methods, we consider running time to reach convergence (5% tolerance). For full Twitter dataset, we consider 1 iteration.

Datasets	Size	MLlib-PCA	Mahout-PCA	sPCA	sSketch-PCA	TallnWide
PubMed2K	$8.2M \times 2K$	94.54	57.27	35.32	24.21	21.11
PubMed	$8.2M \times 141K$	**Fail**	19491.51	634.90	585.37	533.91
SiftFeature	$4.5M \times 128$	107.09	1801.52	556.10	182.10	194.93
AmazonRating2K	$6.6M \times 2K$	58.58	63.54	10.88	12.99	5.731
AmazonRating50K	$6.6M \times 50K$	**Fail**	2042.49	102.343	64.33	48.76
AmazonRating	$21M \times 9.8M$	**Fail**	**Fail**	**Fail**	**Fail**	6339.77
Twitter1K	$50K \times 1K$	26.63	22.86	1.18	1.12	1.10
Twitter10M	$10M \times 10M$	**Fail**	**Fail**	**Fail**	**Fail**	2987.69
Twitter (1 iteration only)	$50M \times 50M$	**Fail**	**Fail**	**Fail**	**Fail**	14687.62

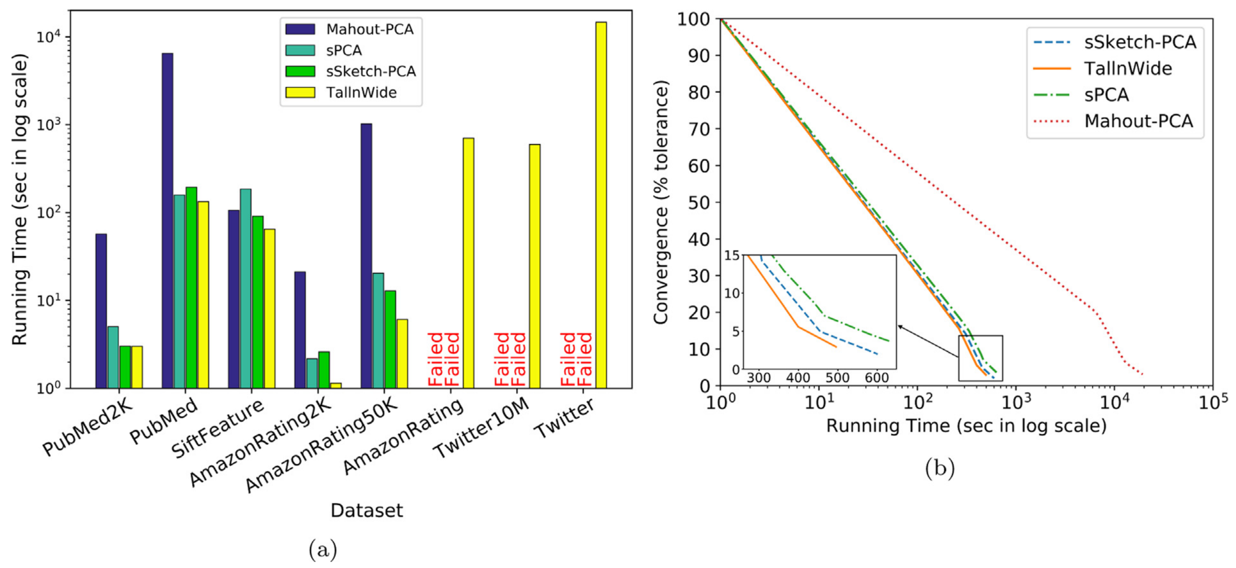


Fig. 4. (a) Per iteration running time for all iterative methods. (b) Comparison of running time of iterative methods to converge (5% tolerance) for PubMed dataset.

as possible. In our observation, $\rho = 30$ yields the best result, so we have used $\rho = 30$ for TallnWide. However, determining the efficient value of ρ to keep the count of blocks minimum while ensuring the mitigation of overflow error is out of the scope of this paper and is kept as future work.

Running Times in a Single DC: To compare running times of our method against all other techniques, we first show that as an iterative method TallnWide takes less time to finish one iteration, compared to other iterative methods. Later, we show that when all the iterative techniques converge, TallnWide takes the least amount of time as well. For each method, to get results faster and reduce monetary cost, we consider deriving top 10 principal components. Point to be noted that all the methods we consider for comparison involve randomization in their approaches. This may result in either a decent starting point, which tends to converge earlier or any average starting point, which needs more iterations to converge. For this reason, in order to maintain fairness among the methods, we use per iteration time comparison.

Fig. 4(a) shows per iteration running times for all the iterative methods for all datasets (except Twitter1K for which logarithmic running time is negative). In all cases, TallnWide takes the least amount of time. For example, for AmazonRating50K dataset, TallnWide takes only 6.10 s while Mahout-PCA, sPCA, and sSketch-PCA take 1021.25, 20.47, and 12.86 s respectively. For other datasets, the results are similar. Furthermore, when the number of dimensions is too high, Mahout-PCA, sPCA, and sSketch-PCA fail to run. For example, for AmazonRating ($D = 9.8M$), Twitter10M ($D = 10M$), and Twitter datasets ($D = 50M$), they fail, whereas TallnWide performs smoothly, and takes 742.47, 597.54, and 14687.62 s per iteration, respectively.

We now show that TallnWide takes less time for convergence. In Fig. 4(b), running times of iterative methods to converge (5% tolerance) for PubMed dataset is shown. We can observe that TallnWide takes the least amount of time (evident from the zoomed segment) to converge compared to others (533.91 s vs 585.37, 634.90, and 19491.51 s) and offers **1.1 − 37×** better performance. Table 4 shows full running times of all the methods for all datasets. For Twitter, we show the results of the first iteration only (to reduce the monetary cost). Observe that, due to the high dimension, MLlib-PCA fails to run on full PubMed dataset. If we take a smaller dataset, TallnWide still takes less time. For example, for PubMed2K MLlib-PCA, Mahout-PCA, sPCA, and sSketch-PCA take 94.54, 57.27, 35.32, and 24.21 s to finish while TallnWide takes only 21.11 s (a factor of **1.2 − 4.5×** improvement). Results for other datasets are similar, and TallnWide offers better performance (**1.1 − 42×**) in running time in most of the cases. However, only for the dense matrix SiftFeature, MLlib-PCA takes the least time. It is because MLlib-PCA is a deterministic algorithm, and it does not have any overhead for sparsity preserving calculation. However, here the dimension is relatively small, which is not a case with big data, for which MLlib-PCA fails to perform.

Scalability: TallnWide does not face out-of-memory error as dimensions of data increase arbitrarily. To show this, we work with different dimensions of Twitter datasets such as $1K$, $2K$, $4K$, $6K$, $8K$, $10K$, $100K$, $1M$, $5M$, $10M$, $20M$ and $50M$, and record the memory consumption using Ganglia [54] for each method. Fig. 5 shows the result. As expected MLlib-PCA faces out-of-memory error after $6K$ dimensions. Mahout-PCA and sPCA too face such error after handling up to $5M$ dimensions while sSketch-PCA

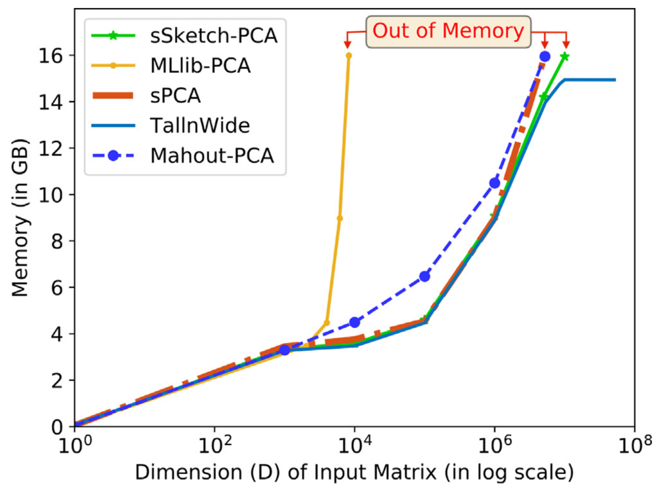


Fig. 5. Memory consumption in one working node for all methods for Twitter dataset.

can prolong its scalability further but eventually fails when the dimension reaches close to 10M (from Table 4, we can see that sSketch-PCA fails for datasets with dimension 9.8M, 10M, and 50M). However, due to the merit of block-division, TallnWide does not face such a problem. Thus, from this figure and Table 4, we can say that TallnWide successfully handles up to **10×** dimensions which bears the testimony of our claim that it can handle an arbitrarily large number of dimensions.

Intermediate Data Size: We measure intermediate data size generated by all the comparing algorithms for PubMed, AmazonRating50K, and SiftFeature datasets. We choose these datasets because they allow other methods to run without facing out-of-memory errors. Table 5 shows the experimental results. Along with the zero-noise model, which reduces the generation of intermediate data, our TallnWide offers a significantly lower memory utilization by providing a block division of the main parameter (principal subspace $\boldsymbol{W}$). Although we may incur some computational overhead due to partitioning, the total running time is not impacted by that much as we showed earlier. From the table, we can see, in case of MLlib-PCA, the intermediate data size grows very quickly and eventually goes beyond 150 GB for our all higher-dimensional datasets. It exceeds the total aggregated memory of the cluster. Since the intermediate data size for Mahout-PCA depends on the input row size N, which is typically very big in our case, it also generates a lot of intermediate data. For the PubMed dataset, Mahout-PCA, sPCA and sSketch-PCA generate 3.65GB, 198MB and 151MB of intermediate data respectively, whereas TallnWide generates only 131 MB, a factor of **29×**, **1.5×**, and **1.2× reductions** respectively. In case of SiftFeature which has a row size of about 4 million but only has 128 dimensions, we see our method offers a huge reduction of memory compared to Mahout-PCA (0.67 GB vs 128 KB) and also generates a slightly lower amount of intermediate data than the closest competitors sPCA (143 KB vs 128 KB) and sSketch-PCA (140 KB vs 128 KB).

8.2. Evaluation for geo-distributed DCs

We first describe three different accumulation strategies for the implementation of Geo-Accumulation method in Algorithm 1.

- *gSpark-Accumulation:* This is a naïve strategy where we implement the accumulation of the partial parameters in a centralized approach using the out-of-the-box solution provided by Spark. For this, we deploy Spark in the geo-distributed environment, which we call gSpark. Our single cluster for this accumulation scheme consists of 1 master along with 14 slaves. The slaves are located at different (distant) geo-distributed locations. This accumulation method demands the passing of raw data among the slaves in order to achieve parallel computational capability.
- *gHDFS-Accumulation:* To avoid passing raw data, we discard the centralized approach, and consider 3 instances from each zone as a single DC. In each DC, we set up a Spark cluster (1 master and 2 slaves). In addition, we set up another Spark cluster taking all the master instances from each DC, i.e., a *cluster of masters*, in order to accumulate the parameter only. After that, we deploy HDFS in a geo-distributed fashion, which we call gHDFS. In this setting, each DC stores partial results in gHDFS and notify the master of the cluster of masters, or *master of masters*, to start accumulation. Master of masters reads partial results from and stores the final result in the same gHDFS. Since gHDFS is accessible from all the DCs, each DC can read the accumulated parameter at the beginning of the next iteration.
- *TallnWide-Accumulation:* In gHDFS, all DCs have one shared file system which does not depict the real scenario of a geo-distributed environment. So, now in our approach, we discard gHDFS and cluster of masters. To do so, we make the assumption that the masters from each DC are able to communicate among themselves in a real application via *SSH* for passing parameter only. We provide our custom accumulation using the native file system, and *Linux* shell script. All of the instructions for gSpark and gHDFS, and code for all strategies are publicly available in the provided *GitHub* link.

Table 5
TallnWide is space-efficient. It produces less intermediate data than state-of-the-art methods. For a fair comparison, we compare results with only those data where other methods do not face memory issues.

Technique	PubMed	AmazonRating50K	SiftFeature
MLlib-PCA	≥150GB	≥150GB	896KB
Mahout-PCA	3.65GB	1.05GB	0.67GB
sPCA	198MB	66MB	143KB
sSketch-PCA	151MB	53MB	140KB
TallnWide	131MB	45MB	128KB

Table 6
Comparison of running times (in sec) among different strategies (for **single iteration**): gSpark-Accumulation, gHDFS-Accumulation, and TallnWide-Accumulation.

Dataset	gSpark-Accumulation	gHDFS-Accumulation	TallnWide-Accumulation
AmazonRating	1547.14	1195.44	525.56
PubMed	212.45	180.13	137.34

Running Times Among Accumulation Strategies: To keep costs low, we use two datasets, namely AmazonRating and PubMed, to validate the merit of TallnWide-Accumulation scheme. Table 6 shows the results of the strategies for the datasets. The results are shown for running one iteration only. Notice that since gSpark has to transmit and shuffle raw data, gSpark-Accumulation is slow. gHDFS has additional overhead for initialization and replication, and therefore gHDFS-Accumulation takes longer time too. However, our final strategy, TallnWide-Accumulation, operates faster because it does not need to pass raw data and does not have any additional overhead and offers **1.3 − 2.9×** better performance than the other mentioned strategies.

Table 7
Average running time (in sec) needed for taking different DC as center using TallnWide-Accumulation strategy. Statistics is shown for AmazonRating and PubMed datasets.

Dataset	Approach 2				Approach 3
	Ireland	North Virginia 1	North Virginia 2	Oregon 1	Oregon 2 (Ideal)
AmazonRating	594.34	537.57	542.56	546.45	**525.56**
PubMed	158.31	138.55	141.53	149.44	**137.34**

Communication Efficiency Across Different Approaches: Using TallnWide-Accumulation strategy, we now establish the merit of using Eq. (13) for Approach 3 (Efficient Order w/ Ideal Central DC) over Approach 2 (Efficient Order) in Fig. 2 mentioned in 5.2. We do not experimentally validate Approach 1 (Trivial Order) because of straightforward observation and cutting down the monetary cost of running EC2 instances. Table 2 shows the B/W (a symmetric matrix) between every pair of DCs. From this table and Eq. (13), we derive the slowest B/Ws for all masters from each DC:

$$B/W_{min} = \{9.25, 11.35, 10.95, 9.25, 11.60\}$$

For Oregon 2, we get the maximum from this set which is 11.60MB/s. So, using Oregon 2 as the central DC for accumulation should yield a faster result. From Table 7, we see that this is exactly the case i.e. for both the datasets, Oregon 2 takes the least amount of time.

To summarize, our results show that TallnWide offers high scalability and better performance than its competitors.

9. Conclusions

In this paper, we have proposed a new algorithm, referred to as TallnWide, to meet the challenges of geo-distributed tall and wide big data. We have devised a block-division EM algorithm for PCA, and we have demonstrated that compared to the state-of-the-art techniques, our method is highly scalable (handles **10×** higher dimension) and offers better performance (**1.1** − **42×** faster). We also give a better solution in our algorithm for the geo-distributed environment. We show that our accumulation strategy, coupled with our communication efficient calculation yields up to **2.9×** faster result than other alternatives. In future, we intend to give an optimal block-partition scheme and provide a technique for perturbation of values of the parameter for privacy preservation. For reproducibility and extensibility of our work, we make the source code of TallnWide available at https://github.com/tmadnan10/TallnWide.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was conducted at the Department of Computer Science and Engineering (CSE), Bangladesh University of Engineering and Technology (BUET) and supported by CASR Research Grant from Bangladesh University of Engineering and Technology (BUET). The authors also thank the reviewers for their valuable comments for improving the presentation and strenght of the paper.

References

[1] Haewoon Kwak, Changhyun Lee, Hosung Park, Sue Moon, What is twitter, a social network or a news media? in: Proceedings of the 19th international conference on World wide web, 2010, pp. 591–600.
[2] Julian McAuley, Amazon product data. (2014), 2014.
[3] Moshe Lichman, et al., UCI machine learning repository, 2013.
[4] Jianqing Fan, Fang Han, Han Liu, Challenges of big data analysis, Natl. Sci. Rev. 1 (2) (2014) 293–314.
[5] Xiao Fu, Kejun Huang, Evangelos E. Papalexakis, Hyun-Ah Song, Partha Pratim Talukdar, Nicholas D. Sidiropoulos, Christos Faloutsos, Tom Mitchell, Efficient and distributed algorithms for large-scale generalized canonical correlations analysis, in: 2016 IEEE 16th International Conference on Data Mining (ICDM), IEEE, 2016, pp. 871–876.
[6] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, Samee Ullah Khan, The rise of big data on cloud computing: Review and open research issues, Inf. Syst. 47 (2015) 98–115.
[7] Cho-Jui Hsieh, Si Si, Inderjit S. Dhillon, Communication-efficient distributed block minimization for nonlinear kernel machines, in: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2017, pp. 245–254.
[8] Christopher M. Bishop, Pattern Recognition and Machine Learning, springer, 2006.
[9] Chris Ding, Xiaofeng He, K-means clustering via principal component analysis, in: Proceedings of the twenty-first international conference on Machine learning, 2004, p. 29.
[10] Baolin Guo, Chenping Hou, Feiping Nie, Dongyun Yi, Semi-supervised multi-label dimensionality reduction, in: 2016 IEEE 16th International Conference on Data Mining (ICDM), IEEE, 2016, pp. 919–924.
[11] Qiang Wang, Vasileios Megalooikonomou, A dimensionality reduction technique for efficient time series similarity analysis, Inf. Syst. 33 (1) (2008) 115–132.
[12] Qian Du, James E. Fowler, Hyperspectral image compression using jpeg2000 and principal component analysis, IEEE Geosci. Remote Sensing Lett. 4 (2) (2007) 201–205.
[13] Josep M. Porta, Jakob J. Verbeek, Ben J.A. Kröse, Active appearance-based robot localization using stereo vision, Auton. Robots 18 (1) (2005) 59–80.
[14] Alexander N. Gorban, Balázs Kégl, Donald C. Wunsch, Andrei Y. Zinovyev, et al., Principal Manifolds for Data Visualization and Dimension Reduction, vol. 58, Springer, 2008.
[15] Apache Mahout, What is apache mahout?, copyright© 2014 the apache software foundation, licensed under the apache license, version 2.0.
[16] Xiangrui Meng, Mllib: Scalable machine learning on spark, in: Spark Workshop April, 2014.
[17] Tarek Elgamal, Maysam Yabandeh, Ashraf Aboulnaga, Waleed Mustafa, Mohamed Hefeeda, spca: Scalable principal component analysis for big data on distributed platforms, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015, pp. 79–91.
[18] Md Mehrab Tanjim, Muhammad Abdullah Adnan, ssketch: A scalable sketching technique for pca in the cloud, in: Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining, 2018, pp. 574–582.
[19] George Lee, Jimmy Lin, Chuang Liu, Andrew Lorek, Dmitriy Ryaboy, The unified logging infrastructure for data analytics at twitter, 2012, arXiv preprint arXiv:1208.4171.
[20] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, Ion Stoica, Low latency geo-distributed data analytics, ACM SIGCOMM Comput. Commun. Rev. 45 (4) (2015) 421–434.
[21] Ashish Vulimiri, Carlo Curino, P. Brighten Godfrey, Thomas Jungblut, Jitu Padhye, George Varghese, Global analytics in the face of bandwidth and regulatory constraints, in: 12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15), 2015, pp. 323–336.
[22] Ashish Vulimiri, Carlo Curino, Philip Brighten Godfrey, Thomas Jungblut, Konstantinos Karanasos, Jitendra Padhye, George Varghese, Wanalytics: Geo-distributed analytics for a data intensive world, in: Proceedings of the 2015 ACM SIGMOD international conference on management of data, 2015, pp. 1087–1092.
[23] Aditya Auradkar, Chavdar Botev, Shirshanka Das, Dave De Maagd, Alex Feinberg, Phanindra Ganti, Lei Gao, Bhaskar Ghosh, Kishore Gopalakrishna, Brendan Harris, et al., Data infrastructure at linkedin, in: 2012 IEEE 28th International Conference on Data Engineering, IEEE, 2012, pp. 1370–1381.
[24] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, Hao Liu, Data warehousing and analytics infrastructure at facebook, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, 2010, pp. 1013–1020.

[25] Sam T. Roweis, Em algorithms for pca and spca, in: Advances in Neural Information Processing Systems, 1998, pp. 626–632.
[26] Apache Spark, Apache spark: Lightning-fast cluster computing, 2016, pp. 2168–7161, URL http://spark.apache.org.
[27] Olivia Choudhury, Yoonyoung Park, Theodoros Salonidis, Aris Gkoulalas-Divanis, Issa Sylla, et al., Predicting adverse drug reactions on distributed health data using federated learning, in: AMIA Annual Symposium Proceedings, vol. 2019, American Medical Informatics Association, 2019, p. 313.
[28] Jie Xu, Fei Wang, Federated learning for healthcare informatics, 2019, arXiv preprint arXiv:1911.06270.
[29] European Union Directive, Payment services (PSD 2) - directive, 2014, URL https://ec.europa.eu/info/law/payment-services-psd-2-directive-eu-2015-2366_en.
[30] Kewei Cheng, Tao Fan, Yilun Jin, Yang Liu, Tianjian Chen, Qiang Yang, Secureboost: A lossless federated learning framework, 2019, arXiv preprint arXiv:1901.08755.
[31] Wensi Yang, Yuhang Zhang, Kejiang Ye, Li Li, Cheng-Zhong Xu, Ffd: A federated learning based method for credit card fraud detection, in: International Conference on Big Data, Springer, 2019, pp. 18–32.
[32] Shiva Raj Pokhrel, Towards efficient and reliable federated learning using blockchain for autonomous vehicles, Comput. Netw. (2020) 107431.
[33] Ahmet M. Elbir, Sinem Coleri, Federated learning for vehicular networks, 2020, arXiv preprint arXiv:2006.01412.
[34] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, Blaise Aguera y Arcas, Communication-efficient learning of deep networks from decentralized data, in: Artificial Intelligence and Statistics, 2017, pp. 1273–1282.
[35] Jakub Konečnỳ, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, Dave Bacon, Federated learning: Strategies for improving communication efficiency, 2016, arXiv preprint arXiv:1610.05492.
[36] Robin C. Geyer, Tassilo Klein, Moin Nabi, Differentially private federated learning: A client level perspective, 2017, arXiv preprint arXiv:1712.07557.
[37] Syeda Nahida Akter, Muhammad Abdullah Adnan, Weightgrad: Geo-distributed data analysis using quantization for faster convergence and better accuracy, in: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2020, pp. 546–556.
[38] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, Onur Mutlu, Gaia: Geo-distributed machine learning approaching {LAN} speeds, in: 14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17), 2017, pp. 629–647.
[39] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, Hai Li, Terngrad: Ternary gradients to reduce communication in distributed deep learning, in: Advances in Neural Information Processing Systems, 2017, pp. 1509–1519.
[40] Simo Puntanen, George P.H. Styan, Jarkko Isotalo, Eigenvalue decomposition, in: Matrix Tricks for Linear Statistical Models, Springer, 2011, pp. 357–390.
[41] Gene H. Golub, Charles F. Van Loan, Matrix Computations, Vol. 3, 2012.
[42] Jonathon Shlens, A tutorial on principal component analysis, 2014, arXiv preprint arXiv:1404.1100.
[43] Nathan Halko, Per-Gunnar Martinsson, Joel A. Tropp, Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions, SIAM Rev. 53 (2) (2011) 217–288.
[44] Ninh Pham, Rasmus Pagh, Fast and scalable polynomial kernels via explicit feature maps, in: Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining, 2013, pp. 239–247.
[45] Nathan P. Halko, Randomized Methods for Computing Low-Rank Approximations of Matrices (Ph.D. thesis), University of Colorado at Boulder, 2012.
[46] Michael E. Tipping, Christopher M. Bishop, Probabilistic principal component analysis, J. R. Stat. Soc. Ser. B Stat. Methodol. 61 (3) (1999) 611–622.
[47] Facebook Research, Fast randomized svd, 2014, URL https://research.fb.com/fast-randomized-svd.
[48] Tarek Elgamal, Mohamed Hefeeda, Analysis of pca algorithms in distributed environments, 2015, arXiv preprint arXiv:1503.05214.
[49] Alexander Smola, Shravan Narayanamurthy, An architecture for parallel topic models, Proc. VLDB Endow. 3 (1–2) (2010) 703–710.
[50] Yun Seong Lee Lee, Markus Weimer, Youngseok Yang, Gyeong-In Yu, Dolphin: Runtime optimization for distributed machine learning, in: Proc. of ICML ML Systems Workshop, 2016.
[51] Benoit Dageville, Bhaskar Ghosh, Rushan Chen, Thierry Cruanes, Mohamed Zait, Parallel partition-wise aggregation, August 17 2010, US Patent 7, 779, 008.
[52] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, Li Fei-Fei, Imagenet: A large-scale hierarchical image database, in: 2009 IEEE Conference on Computer Vision and Pattern Recognition, IEEE, 2009, pp. 248–255.
[53] David G. Lowe, Object recognition from local scale-invariant features, in: Proceedings of the Seventh IEEE International Conference on Computer Vision, vol. 2, IEEE, 1999, pp. 1150–1157.
[54] Matthew L. Massie, Brent N. Chun, David E. Culler, The ganglia distributed monitoring system: design, implementation, and experience, Parallel Comput. 30 (7) (2004) 817–840.