Incremental and Parallel Analytics on Astrophysical Data Streams

Dmitry Mishin, Tamás Budavári, Alexander Szalay
Departments of Physics and Astronomy
Johns Hopkins University
Baltimore, Maryland 21218
Email: {dmitry,budavari,szalay}@jhu.edu

Yanif Ahmad
Department of Computer Science
Johns Hopkins University
Baltimore, Maryland 21218
Email: yanif@jhu.edu

***Abstract*—Stream processing methods and online algorithms are increasingly appealing in the scientific and large-scale data management communities due to increasing ingestion rates of scientific instruments, the ability to produce and inspect results interactively, and the simplicity and efficiency of sequential storage access over enormous datasets. This article will showcase our experiences in using off-the-shelf streaming technology to implement incremental and parallel spectral analysis of galaxies from the Sloan Digital Sky Survey (SDSS) to detect a wide variety of galaxy features. The technical focus of the article is on a robust, highly scalable principal components analysis (PCA) algorithm and its use of coordination primitives to realize consistency as part of parallel execution. Our algorithm and framework can be readily used in other domains.**

I. Introduction

Processing data in an online fashion is an integral part of scientific workflows. It facilitates data-driven control and steering, and enables interactive exploration with resource-hungry simulations and analyses amongst other functionality. In recent years, there has been an abundance of research and tools on scalable data analytics from the data management, scientific, and machine learning communities, where our need to parallelize analysis algorithms is driven by rapidly improving instruments and data collection capacity. For example, the Large Synoptic Survey Telescope is expected to generate data at a sustained rate of 160MB per second, nearly a 40-fold increase over the 4.3MB per second generation rate for the Sloan Digital Sky Survey (SDSS). However, much of this work has focused on parallelization in an offline setting, where data movement, synchronization and result delivery can be predetermined, and where scant attention has been paid to in-flight use of partial results.

Many popular analytics techniques including expectation maximization, support vector machines and principal component analysis are extremely well suited to scalable execution on dataflow architectures [1] . Batch parallel processing frameworks such as MapReduce, DryadLINQ and Spark [2] have been successfully used for these algorithms given their heavy use of *partial sums*, and the ease of programming partial sums in the aforementioned tools. Partial sums are central to online aggregation and approximate query processing, and form the basis of incremental and parallel methods that are capable of providing a feed of in-flight results. These early results are invaluable when processing petabytes, with large-scale data movement resulting in execution times ranging from minutes to weeks.

Partial sum computations are straightforward to parallelize in the offline setting, but require coordination and synchronization when used to compute a single result in high fan-in aggregation trees and highly pipelined dataflow architectures. When used on parallel data streams, results are produced at either predefined aggregation points, for example at fixed size time- or tuple-based window boundaries, or via data-driven synchronization where results are produced based on the semantics of the analysis algorithm, for example when local thresholds indicate a significant change in the result as seen in adaptive filtering and related techniques.

This article presents a scalable implementation of a partial sum analytic with data-driven synchronization on a parallel stream processing platform. The technical focus is a statistically robust algorithm that handles the presence of outliers in the streaming dataset to reduce the need for synchronization. Stream processing systems provide a pipelined execution platform for online aggregation, and enable the decoupling of storage and computation systems to fully take advantage of high-throughput sequential I/O and derandomize data access, particularly in the multiquery setting [3]. Our system of choice is IBM Infosphere Streams. The solution was designed and built to be scalable and runnable on a large distributed cluster. Running the implemented algorithm as a service in cloud is a good way to get the flexible scalable system that would satisfy the needs of large data stream realtime processing.

The algorithm we consider is principal components analysis (PCA), a popular algorithm [4] [5] for automatic dataset exploration that finds patterns based on the estimation of an eigensystem of a covariance matrix. PCA can be evaluated in parallel by randomly partitioning a data stream and maintaining independent eigensystem estimates for each substream. Synchronization involves communicating eigensystems to yield a global, accurate estimate given disjoint substreams that potentially contain outliers.

II. Robust Incremental PCA

We present a novel way of running the parallel streaming PCA algorithm, which efficiently estimates the eigensystem

by processing the data stream in parallel and periodically synchronizing the estimated eigensystems. First we discuss the iterative procedure, next the statistical extensions that enable our method applicable to cope with outliers as well as noisy and incomplete data, and the streams synchronization algorithm to run the processing in parallel.

We focus on a fast data stream, whose elements are high-dimensional $\boldsymbol{x}_n$ vectors. Similarly to the recursion formulas commonly used for evaluating sums or averages, e.g., the mean $\boldsymbol{\mu}$ of the elements, we can also write the covariance matrix $\mathbf{C}$ as a linear combination of the previous estimate $\mathbf{C}_{\rm prev}$ and a simple expression of the incoming vector. If we solve for the top p significant eigenvectors that account for most of the sample variance, the $\mathbf{E}_p\mathbf{\Lambda}_p\mathbf{E}_p^T$ product is a good approximation of full covariance matrix, where $\{\mathbf{\Lambda}_p, \mathbf{E}_p\}$ is the truncated eigensystem. The iterative equation for the covariance becomes

$$\mathbf{C} \approx \gamma \mathbf{E}_p\mathbf{\Lambda}_p\mathbf{E}_p^T + (1-\gamma)\boldsymbol{y}\boldsymbol{y}^T = \mathbf{A}\mathbf{A}^T \tag{1}$$

where $\boldsymbol{y}=\boldsymbol{x}-\boldsymbol{\mu}$, and we can also write this as the product of a matrix $\mathbf{A}$ and its transpose, where $\mathbf{A}$ has only $(p+1)$ columns, thus it is potentially much lower rank than the covariance matrix itself [6]. The columns of $\mathbf{A}$ are constructed from the previous eigenvalues λ_k, eigenvectors $\boldsymbol{e}_k$, and the next data vector $\boldsymbol{y}$ as

$$\boldsymbol{a}_k = \boldsymbol{e}_k\sqrt{\gamma\lambda_k}, \qquad \text{for} \quad k=1\ldots p \tag{2}$$

$$\boldsymbol{a}_{p+1} = \boldsymbol{y}\sqrt{1-\gamma} \tag{3}$$

Then the singular-value decomposition of $\mathbf{A}=\mathbf{U}\mathbf{W}\mathbf{V}^T$ provides the updated eigensystem, because the eigenvectors are $\mathbf{E}=\mathbf{U}$ and the eigenvalues $\mathbf{\Lambda}=\mathbf{W}^2$.

A. Robustness against Outliers

The classic PCA procedure essentially fits a hyperplane to the data, where the eigenvectors define the projection matrix onto this plane. If the truncated eigensystem consists of p basis vectors in $\mathbf{E}_p$, the projector is $\mathbf{E}_p\mathbf{E}_p^T$, and the residual of the fit for the nth data vector is

$$\boldsymbol{r}_n = (\mathbf{I} - \mathbf{E}_p\mathbf{E}_p^T)\boldsymbol{y}_n \tag{4}$$

Using this notation, PCA solves the minimization of the average $\sigma^2=\langle r_n^2\rangle$, The sensitivity of PCA to outliers comes from the fact that the sum will be dominated by the extreme values in the data set.

The current state-of-the-art technique was introduced by [7] to overcome these limitations, which is based on a robust M-estimate [8] of the scale, called M-scale. Here we minimize a different σ that is the M-scale of the residuals, and satisfies

$$\frac{1}{N}\sum_{n=1}^{N}\rho\left(\frac{r_n^2}{\sigma^2}\right) = \delta \tag{5}$$

where the robust ρ-function is bound and assumed to be scaled to values between $\rho(0)=0$ and $\rho(\infty)=1$. The parameter δ controls the breakdown point where the estimate explodes due to too much contamination of outliers. By implicit differentiation the robust solution yields a very intuitive result, where the location $\boldsymbol{\mu}$ is a weighted average of the observation vectors, and the hyperplane is derived from the eigensystem of a weighted covariance matrix,

$$\boldsymbol{\mu} = \Big(\sum w_n\boldsymbol{x}_n\Big) \Big/ \Big(\sum w_n\Big) \tag{6}$$

$$\mathbf{C} = \sigma^2\Big(\sum w_n(\boldsymbol{x}_n-\boldsymbol{\mu})(\boldsymbol{x}_n-\boldsymbol{\mu})^T\Big) \Big/ \Big(\sum w_n r_n^2\Big) \tag{7}$$

where $w_n = W(r_n^2/\sigma^2)$ and $W(t) = \rho'(t)$. The weight for each observation vector depends on σ^2, which suggests the appropriateness of an iterative solution, where in every step we solve for the eigenvectors and use them to calculate a new σ^2 scale; see [7] for details. One way to obtain the solution of eq.(5) is to re-write it in the intuitive form of

$$\sigma^2 = \frac{1}{N\delta}\sum_{n=1}^{N} w_n^\star r_n^2 \tag{8}$$

where $w_n^\star = W^\star(r_n^2/\sigma^2)$ with $W^\star(t)=\rho(t)/t$. Although, this is not the solution as the right hand side contains σ^2 itself, it can be shown that its iterative re-evaluation converges to the solution. By incrementally updating the eigensystem, we can allow for the solution for σ^2 simultaneously.

Processing random test data with artficially generated outliers for classical versus robust PCA algorithm shows significant bebefits of the latter (figure 1). Each outlier data point takes over the top eigenvector creating a rainbow effect on the left plot. The eigensystem does not converge and eigenvalues are noisy in the beginning. The robust PCA shows converged eigenvalue plot on the right and is not sensitive to outliers.

B. Algorithm Extensions

When dealing with the online arrival of data, there are several options to maintain the eigensystem over varying temporal extents, including a damping factor or time-based windows. These issues are orthogonal to the parallel PCA computation and can both be applied provided a single source of data that provides a well-ordered stream. Both approaches can be implemented, exploiting sharing strategies for sliding window scenarios, as well as data-driven control and coupling of the duration of the window with the eigensystem size. Our algorithm is also capable of handling missing data and gaps in the data stream.

The recursion equation for the mean is formally almost identical to the classic case, and we introduce new equations to propagate the weighted covariance matrix and the scale,

$$\boldsymbol{\mu} = \gamma_1\boldsymbol{\mu}_{\rm prev} + (1-\gamma_1)\boldsymbol{x} \tag{9}$$

$$\mathbf{C} = \gamma_2\mathbf{C}_{\rm prev} + (1-\gamma_2)\ \sigma^2\boldsymbol{y}\boldsymbol{y}^T/r^2 \tag{10}$$

$$\sigma^2 = \gamma_3\sigma^2_{\rm prev} + (1-\gamma_3)\ w^\star r^2/\delta \tag{11}$$

where the γ coefficients depend on the running sums of 1, w and wr^2 denoted below by u, v and q, respectively [9].

$$\gamma_1 = \alpha v_{\rm prev}/v \quad \text{with} \quad v = \alpha v_{\rm prev} + w \tag{12}$$

$$\gamma_2 = \alpha q_{\rm prev}/q \quad \text{with} \quad q = \alpha q_{\rm prev} + wr^2 \tag{13}$$

$$\gamma_3 = \alpha u_{\rm prev}/u \quad \text{with} \quad u = \alpha u_{\rm prev} + 1 \tag{14}$$

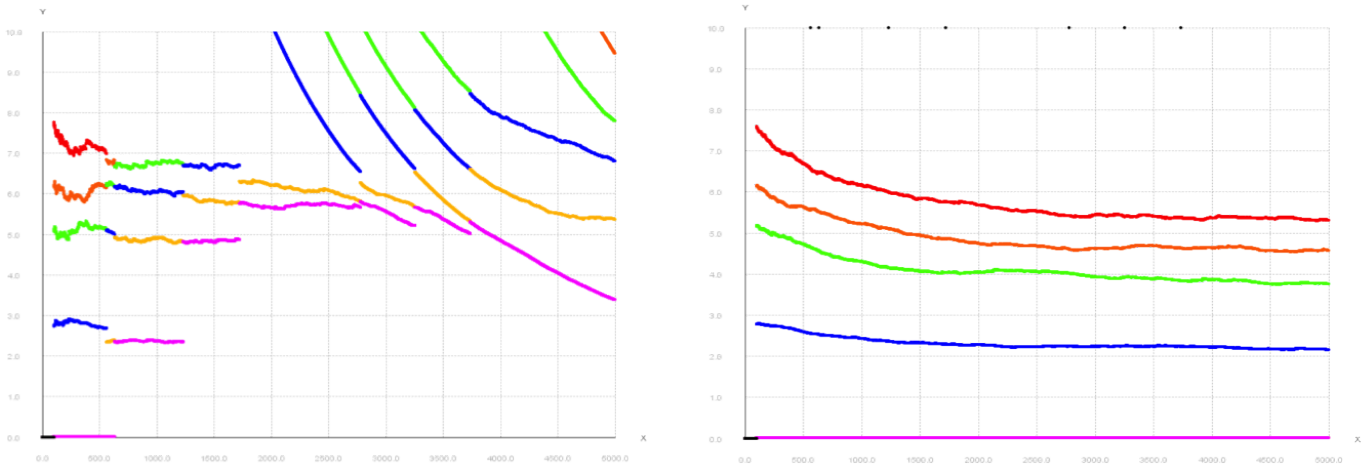

Fig. 1. The plot shows the eigenvalues plots for test data with randomly generated outliers. Comparing the eigenvalues calculated by classical PCA algorithm (left) with robust (right) shows the inability of eigensystem to converge in classical approach when outliers dominate the eigenvectors directions. Robust PCA shows perfect results and fast convergence. The black points on top of the plot mark the detected outliers.

The parameter α introduced here, which takes values between 0 and 1, adjusts the rate at which the evolving solution of the eigenproblem *forgets* about past observations. It sets the characteristic width of the sliding window over the stream of data; in other words, the effective sample size.[1] The value $\alpha = 1$ corresponds to the classic case of infinite memory. Since our iteration starts from a non-robust set of eigenspectra, a procedure with $\alpha < 1$ is able to eliminate the effect of the initial transients. Due to the finite memory of the recursion, it is clearly disadvantageous to put the spectra on the stream in a systematic order; instead they should be randomized for best results.

It is worth noting that robust "eigenvalues" can be computed for any basis vectors in a consistent way, which enables a meaningful comparison of the performance of various bases. To derive a robust measure of the scatter of the data along a given eigenspectrum $\boldsymbol{e}$, one can project the data on it, and formally solve the same equation as in eq.(5) but with the residuals replaced with the projected values, i.e., for the kth eigenspectrum $r_n = \boldsymbol{e}_k \boldsymbol{y}_n$. The resulting σ^2 value is a robust estimate of λ_k.

C. Running in Parallel

There is a strong motivation to parallelize the incremental PCA to process fast streams that a single analysis engine cannot possibly handle. The convenient solution is to drop a given percentage of the stream and process only a fraction of the data. Leaving data items out is, however, undesirable for several reasons. One example is tracking time-dependent phenomena. Keeping all elements is vital to learn the changes in the stream in a timely manner. Dropping 90% of the data would take 10 times longer to yield new solutions. Another example is filtering of data points. Often the goal is to flag outliers for further processing. Dropped items are not even considered.

We tackle very large volumes by partitioning the input stream to multiple nodes. The partitioning is performed by a multithreaded split operator provided by InfoSphere engine. Each new data tuple is being sent to a random running PCA engine which is free to process it. This equally balances the nodes load, although making the order of data points on selected PCA engine unpredictable. These sub-streams are processed independently, and their eigensystems are periodically combined. For example, the common location of two systems is the weighted average $\boldsymbol{\mu} = \gamma_1 \boldsymbol{\mu}_1 + \gamma_2 \boldsymbol{\mu}_2$, where $\gamma_1 + \gamma_2 = 1$. In out case for robust PCA $\gamma_1 = v_1/(v_1 + v_2)$. The covariance becomes

$$\mathbf{C} = \gamma_1 \left(\mathbf{C}_1 + \boldsymbol{\mu}_1 \boldsymbol{\mu}_1^T\right) + \gamma_2 \left(\mathbf{C}_2 + \boldsymbol{\mu}_2 \boldsymbol{\mu}_2^T\right) - \boldsymbol{\mu}\boldsymbol{\mu}^T \qquad (15)$$

When the means are approximately the same, we can further simplify this using the eigensystems

$$\mathbf{C} \approx \gamma_1 \left(\mathbf{E}_1 \boldsymbol{\Lambda}_1 \mathbf{E}_1^T\right) + \gamma_2 \left(\mathbf{E}_2 \boldsymbol{\Lambda}_2 \mathbf{E}_2^T\right) = \mathbf{A}\mathbf{A}^T \qquad (16)$$

which once again can be more efficiently decomposed with the help of a low-rank $\mathbf{A}$ matrix.

The transfer of eigensystems from separate PCA instances is coordinated by the synchronisation controller to follow different synchronization strategies, e.g., peer-to-peer or broadcast. The incremental update formula includes an exponential decay term to phase out old data. This also means that after a while the separate solutions become independent. The nodes verify every time that the eigensystems are statistically independent. The α parameter discussed in section II-B sets the length of exponential decay tail of the data. We can define it as $\alpha = 1 - 1/N$, with N equals to the number of past observations affecting the eigensystem. The synchronisation happens only when the number of observations since the last synchronisation is greater than N, which is checked by each

[1]For example, the sequence u rapidly converges to $1/(1-\alpha)$.

PCA engine. The result of this is that we do not need to keep track of the contributions of other streams, hence our parallel solution can scale out to arbitrary large clusters. In current implementation we check the number of observations since last synchronization to be greater that $1.5 * N$, which is a good compromise between the speed and consistency of eigensystems.

D. Missing Entries in Observations

The other common challenge is the presence of gaps in the observations, i.e., missing entries in the data vectors. Gaps emerge for numerous reasons in real-life measurements. Some cause the loss of random snippets while others correlate with physical properties of the sources. An example of the latter is the wavelength coverage of objects at different redshifts: the observed wavelength range being fixed, the detector looks at different parts of the electromagnetic spectrum for different extragalactic objects.

Now we face two separate problems. Using PCA implies that we believe the Euclidean metric to be a sensible choice for our data, i.e., it is a good measure of similarity. Often one needs to normalize the observations so that this assumption would hold. For example, if one spectrum is identical to another but the source is brighter and/or closer, their distance would be large. The normalization guarantees that they are close in the Euclidean metric. So firstly, we must normalize every spectrum before it is entered into the streaming algorithm. This step is difficult to do in the presence of incomplete data, hence we also have to "patch" the missing data.

Inspired by [10], [11] proposed a solution, where the gaps are filled by an unbiased reconstruction using a pre-calculated eigenbasis. A final eigenbasis may be calculated iteratively by continuously filling the gaps with the previous eigenbasis until convergence is reached [12]. While [11] allowed for a bias in rotation only, the method has recently been extended to accommodate a shift in the normalization of the data vectors [13]. Of course, the new algorithm presented in this paper can use the previous eigenbasis to fill gaps in each input data vector as they are input, thus avoiding the need for multiple iterations through the entire dataset.

The other problem is a consequence of the above solution. Having patched the incomplete data by the current best understanding of the manifold, we have artificially removed the residuals in the bins of the missing entries, thus decreased the length of the residual vector. This would result in increasingly large weights being assigned to spectra with the largest number of empty pixels. One solution is to calculate the residual vector using higher-order eigenspectra. The idea is to solve for not only the first p eigenspectra but a larger $(p+q)$ number of components and estimate the residuals in the missing bins using the difference of the reconstructions on the two different truncated bases.

III. InfoSphere implementation

IBM InfoSphere Streams is a high-performance computing platform that allows user-developed applications to rapidly ingest, analyze, and correlate structured data streams as it arrives from real-time sources. The data is a stream of structured blocks – tuples, having the data structure specified by the application. The application components logic defines the data flow graph and processing routines applied to the data blocks. The architecture of the application leverages the infrastructure of InfoSphere streams throughout to achieve the scalability on large clusters. The solution is ready to be deployed in a Cloud. Dynamic scalable Cloud cluster would be able to meet the demand of large data streams realtime processing by adding additional nodes to the processing cluster when needed.

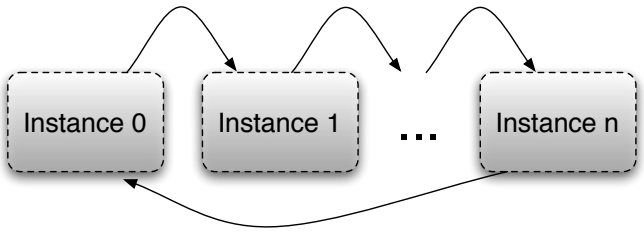

Fig. 3. A simple circular synchronization pattern can achieve reasonable global solutions while minimizing the network traffic.

A. System architecture

The application contains four types of distributed logical blocks connected by three major data streams (Figure 2). Each stream consists of the specific type tuples processed by an application component. To send the tuple between the components located on one node, the tuple memory address can be shared, while sending between nodes involves the data transmission over the network connection.

1) Input data: InfoSphere application is flexible in using the different sources of data. The observations can be artificially generated by the application component for testing purposes, for example, random data matching the predefined properties. Local regular text or binary file with CSV, formated tuples, or a folder of such a files can feed the data. Side service can feed the data using piped stream file, and InfoSphere will lock on the stream end until a new data is streamed through. Network TCP sockets and http URLs are also supported out of the box as a source of data. Additional data sources can be implemented using a custom operator, such as relational database queries. The input data for Streaming PCA application is defined as a time series stream of observations – constant-length vectors of double values.

2) Load balancer: We split the input stream by time to a number of streams that are rerouted to a corresponding PCA engine. The order of target instances is random and is chosen by the splitting component to equally balance and maximize the cluster nodes load. InfoSphere provides the multi-threaded **Signal splitter** component to push the data to multiple targets without blocking the queue on one target. Using this scheme, faster nodes will get more data than slower ones in a period of time, which should be taken into consideration in the eigensystems synchronization algorithm. The number of data

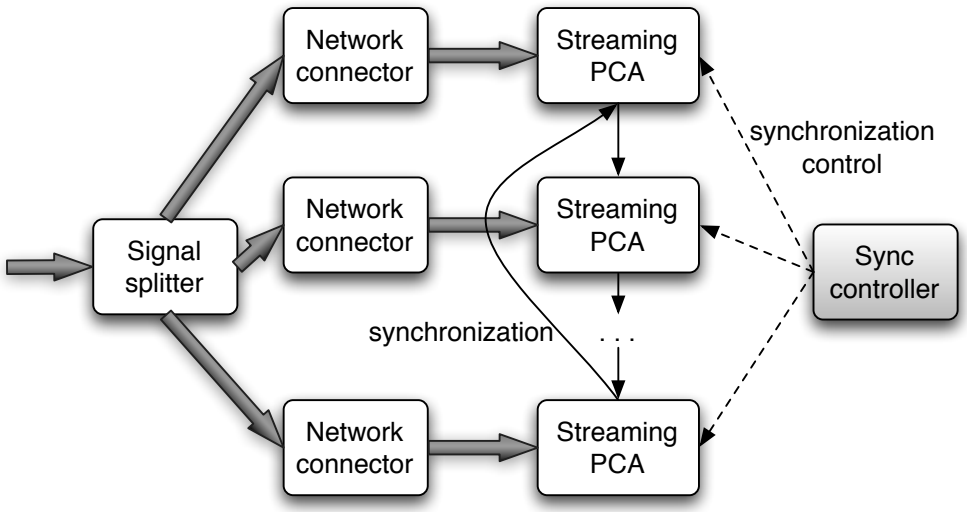

Fig. 2. The analysis graph uses a threaded split operator to scale to the large volume of incoming data. It distributes the inputs to match the processing capacity of each PCA engine. The synchronization messages are also implemented in the same framework, which trigger the communication among the independent instances to achieve a global solution.

streams can be adjusted to match the the rate of incoming stream tuples and the size of data vector allowing the realtime processing of the data and efficient load-balancing of available cluster nodes. The split data stream comes to a corresponding PCA engine for processing using a **Network connector**.

Although the InfoSphere SPL language provides a set of tools for maintaining the data stream and performing basic data processing, the support of efficient arrays and matrices manipulation in SPL is limited. Implementing complex algorithms dealing with matrix operations required the custom C++ **streaming PCA** operator to be implemented. This enables the freedom of choosing a matrix manipulation library having needed features, performance and scalability. Currently we use the Eigen library for all matrix operations and SVD decomposition. SPL provides the developer with API to receive the incoming tuple data to the operator and send back the results of processing to be converted to SPL structures. The C++ operator is refactored into internal InfoSphere code during the project build and distributed in a cluster for running along with other components used. The stateful Streaming PCA operator stores the eigenvalues and eigenvectors (the eigensystem) as well as other state variables as a class members. Upon receiving a new input tuple, its internal states are continuously updated by computationally inexpensive algebraic operations. The stateful operator cannot be moved between the nodes during application execution and therefore needs the optimal placement configuration before running. Thread synchronization for PCA instances is handled by using a mutex class from InfoSphere API in the operator's process method.

Replaceable application components and flexible data flow management make it easy enough to include different partial sum analytics algorithms beyond streaming PCA into the application workflow. The implemented parallelization system will allow distributing the components on multiple nodes to process the incoming data stream in parallel. However, the synchronization criteria for parallel instances must be developed independently in each case. The synchronization schemes (token ring, broadcast, group-based) can be used or new ones can be implemented by the **Sync controller**.

In practice, the application's cluster configuration significantly affects the overall performance. The analysis graph can be partitioned in many ways across the cluster nodes, which optimally minimize data transfer while keeping balanced loads on the processors. To avoid unnecessary packet latency among the graph nodes, we reduce the network traffic by fusing nodes locally such they exchange data in local memory where possible.

B. Synchronization

While load balancing sends the tuples to the instances in random manner, some instances can have the eigensystem values different to the rest of the instances. This may be caused by improper application initialization process when an outlier is in the data, some unusual pattern of incoming data and so forth. Also we want to detect outlier values arriving at each PCA instance. The idea is to keep the eigensystems in sync across all nodes, so that the resulting eigensystem can be obtained from any node. To achieve this, we implement synchronization messages using the same InfoSphere framework, which run periodically. Different synchronization scenarios can be implemented depending on the input data and frequency of processed results reading from the application. For example, circular synchronization involves each instance sending its state to a next one, with the n^{th} instance sending the state to the first (Figure 3). Synchronization signals are generated by a synchronization manager component. The component is

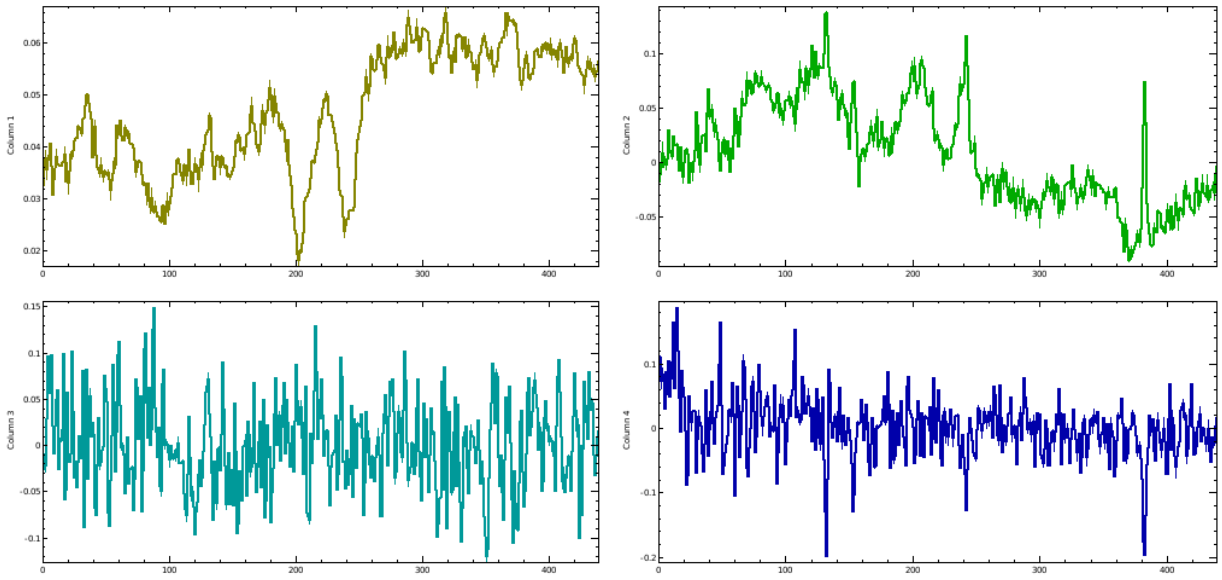

Fig. 4. The plot shows first four eigenvectors of galaxy spectra in the beginning of computation. These are noisy to start with. Two of the first 4 most significant vectors barely show any (spectral) features.

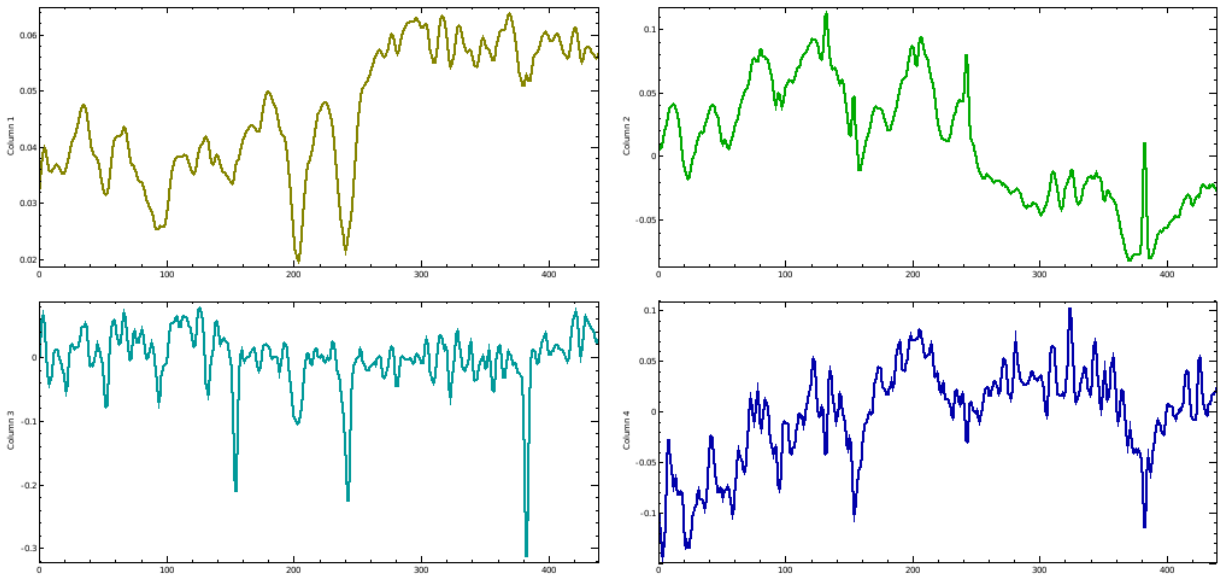

Fig. 5. The plot shows the eigenspectra after processing a significant number of observations. The eigenvectors improve significantly as a function of iteration and develop physically meaningful features. The smoothness of these curves is a sign of robustness as PCA has no notion of where the pixels are relative to each other.

initialised on start-up with the number of threads and generate the controlling signals for each of the threads. The synchronizer sends the signals using a control port (Figure 2). When a control signal is received, the PCA component shares the current eigensystem state with a set of other instances defined in the control message and continues the data processing. The target instances syncronize their state with the shared value by solving the eigenproblem of updated C matrix as described in section II-C. After synchronization is completed, the PCA instance continues receiving the stream of tuples and process it with updated eigensystem values.

The synchronization control subsystem contains the C class generating the sequence of output tuples with sender and receiver number. In our basic case of circular synchronization, receiver number = sender number + 1. When the largest sender number is reached, it sends the tuple with maximum sender and loops the cycle to receiver = 0. Then the loop continues.

Another important synchronization component is standard SPL "Throttle" operator. One controls the rate of synchronization tuples from the control component to the listening PCA engines. The rate of signals in milliseconds can be

adjusted depending on the data processing speed, size of data vectors and the nodes number if the cluster. The synchronization implies the computation time overhead caused by solving the eigenproblem of joined matrices, which is the most computation-intensive operation of the algorithm. Adjusting the Throttle operator timing helps finding the balance between the overall cluster performance and eigensystems consistency. For larger data sets and slower rate, the synchronization timing could be longer, while the need for near-realtime performance for high rates needs more frequent synchronization signals.

The synchronization makes the eigensystem streaming updates smoother. The local eigensystem becomes combined with remote one(s), significantly increasing its weight. If the eigensystem is not converged, the synchronization helps to converge faster.

C. Data processing

First our implementation accumulates a given number of incoming vectors and initializes the eigensystem. The initial set is kept small to minimize the computational requirements. Then the iterative procedure automatically engages. Every time a new input vector is received, the eigensystem is updated by solving the SVD of the low-rank $\mathbf{A}$ matrix, as discussed before. Figure 4 shows the convergence in case of the astronomical galaxy spectra. We see the first four most significant eigenvectors that are noisy to start with and their spectral lines hardly distinguishable. As the engine processes more data, the spectral lines appear more clearly as seen in Figure 5. The intermediate calculation results are periodically saved to the disk for future reference. We frequently see fast convergence way before getting to the last galaxy, which can speed up the scientific analysis. The reason is primarily that inherently low-rank galaxy manifold, which means the galaxies are redundant in good approximation.

D. System performance

The application was tested on the cluster of 10 computing nodes. The hardware was identical that consisted of quad-core Intel Xeon E31230 @ 3.20GHz with 16 GB of RAM and 1G ethernet.

Our preliminary performance results are very promising.

In this section we describe the experiments and the test cases with special attention to the software configurations.

We used gaussian random data artificially enriched with additional signals to test the performance of the Streaming PCA engine. The maximum rate of data generated was tested to be higher than processing rate and did not affect the overall processing performance. The observations processing rate was measured as the number of output tuples at the operator splitting the stream to multiple parallel processors averaged in 30 seconds after about 5 minutes of processing. The synchronization throttle rate was set to 0.5 seconds and $N = 5000$. These settings caused the synchronization for all profiling configurations to happen once the eigensystem become independent as described in section II-C.

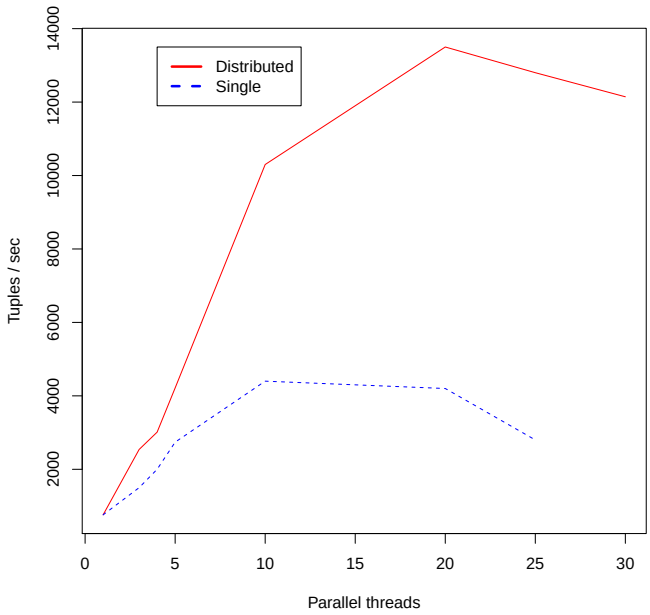

Fig. 6. The performance of the distributed Streaming PCA system processing the tuples with 250 dimensions for 1-30 instances running in parallel. The lines show the difference in placing all the instances on single node versus distributed multiple nodes. We can see the increased performance when using multiple nodes, but with number of threads increasing the network latency becomes the limit and performance of distributed system degrades. The optimum number is 2 instances per node, or 20 instances per 10 nodes in our case.

IBM InfoSphere Streams provides a set of tools for profiling the application. The profiling tool measures the performance of each component and the data channels traffic. To optimize the high tuples rate routes or large sized tuples between operators, it is preferred to use the "Fusion" operators. Those are operators passing data by pointer as a variable in memory instead of using a network. This gives significant decrease of latency and increase in throughput. The optimisation component analyses the logs of profiler and fuses the operators together for optimized data throughput. The optimized code can be run with a profiler again to collect more information about the application data flow. Several steps are usually necessary to optimally layout the components of the application for optimal performance. Profiling for the plots above was performed using the default operators fusion and shows the performance that should be further optimized for specific task. The optimal components placement scheme would change depending on the number of nodes, data vector dimensions number and hardware configuration. It makes it hard trying to tune the application for any possible task. The performance shown gives the guaranteed performance without application-specific optimisation and shows the promising results that can be used in multiple applications.

The network latency and throughput limit was tested by placing multiple parallel engines on single node versus the distributed allocation (Figure 6). We were running the test with

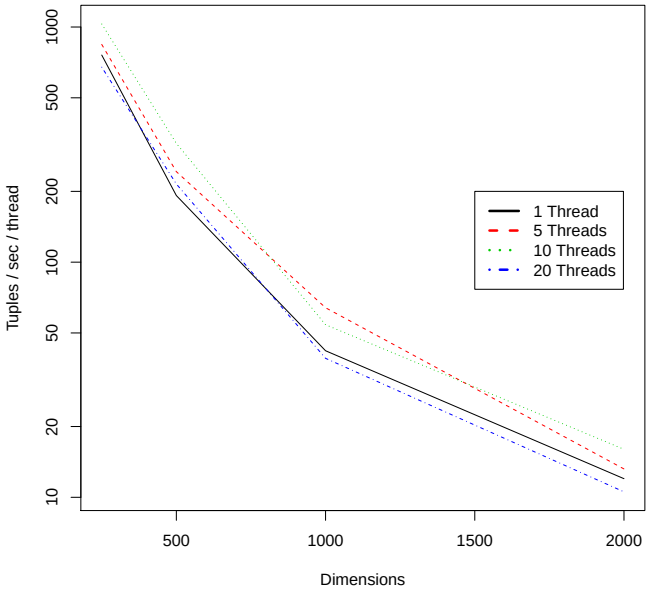

Fig. 7. The logarithmic plot shows the dependency of distributed system performance in tuples / second / thread on the dimensionality of the incoming data stream. The tests were performed for a data stream being split to 1,5,10 and 20 parallel synchronized PCA engines running on 10 computing nodes. We can see good scaling capabilities for 5 and 10 parallel threads. However, single thread underperformance is most likely caused by the non optimal distribution of components in the cluster and interconnect overhead, and 20 threads are saturating the nodes interconnect.

relatively small number of data dimensions (250) to decrease the influence of SVD computation speed on the results. At the same time the speed of PCA processing was always lower than the limit of random data generation rate. While distributed placement of parallel instances gives significant advantage as the number of nodes increases, it has a ceiling of 2 threads per node or 20 threads for 10-node cluster. The performance for distributed scheme degrades for 30 parallel threads. The single-placed instances are using the advantage of operators fusion and capable of processing the data in multiple threads without performance degrading (although not giving any significant advantage either). Increasing the number of threads above 20 for single-allocated configurations makes the system unstable and unresponsive and clearly shows the need for distributed computation. Increasing the number of nodes allows the system to scale for needed observation size and data rate. Enabling the faster network interconnect and using the system profiler and components optimizer will help to further increase the performance of data processing.

Another set of tests measures the performance of the system with distributed placement of streaming PCA components running in parallel. The cumulative performance was measured for various data dimensionality (250-2000) and 1,5,10 and 20 parallel threads synchronized as described above. The most computation-intensive PCA components performing the SVD of truncated matrices were required to run on separate computing nodes. For 20 threads the PCA components were grouped by 2 on all distributed computing nodes evenly.

The logarithmic plot in Figure 7 shows the performance in tuples per second per one thread when running multiple parallel threads. We can see the system reaching the optimal scalability for 5 and 10 parallel threads for 10 nodes cluster. However running a single thread on distributed system shows the decrease of performance. This is caused by the overhead of network connectivity of the application components running on distributed nodes. At the same time running 20 threads on 10 nodes saturates the cluster interconnect in case of default unoptimised components placement and underperforms compared to the smaller number of threads.

IV. Conclusion

Parallelization of statistical learning algorithms is often difficult. In this article we demonstrated a robust PCA algorithm that can process fast streams by distributing the incoming data and combining the partial results from independent analysis engines. Three main contributions are made by the article. We combine the robust and streaming approaches to PCA processing. The stream can be processed in parallel with jobs synchronization that balances the nodes performance with eigensystems consistency. The scalable prototype implementation is developed and described.

When the eigensystem vector locations of the components are close to each other, an approximation becomes possible that speeds up the synchronization step and allows for frequent evaluations even for the high-dimensional input data. Results from our initial parallel implementation in IBM InfoSphere Streams meet the expectations and yield a faster convergence than the individual components by themselves. The performed synthetic data tests show the good performance of the application configuration both for smaller and higher dimensional data streams. The higher-dimensional data processing performance can be improved by using a multithreaded SVD processing algorithm to distribute the computation load to all the node processor cores. The further scaling can be achieved by increasing the number of nodes in the cluster which was shown in the tests. Further improvements of the elements placement are needed to get the more predictable performance results and to improve the components communication pattern. Using the IBM Low Latency Messaging can also significantly improve the overall computations performance and scaling factor of the cluster.

Using IBM InfoSphere Streams as a platform for Streaming PCA engine enables flexible scaling the system on large clusters for high loads having control of distributed components placement. When deployed to a Cloud cluster, the benefit of dynamic cluster scaling allows flexible adapting the available computing power to the data volume demand. The system is able to process a virtually infinite incoming data stream on the fly eliminating the need for data storage. This becomes a benefit for Cloud deployment and makes the system easy to deploy and scale. The rich statistics of components

performance allow avoiding the bottlenecks in the architecture and provide essential information for further increasing the system performance. InfoSphere standard import components enable flexibility in choosing the data source. Connecting import, processing and output components to each other in the workflow adjusts the system to the specific task.

The examples of data streams that can be monitored using this approach include astronomic data such as observed galaxies spectra and remote sensors data such as environment or cluster health monitoring. Using streaming PCA for analyzing the spectra of galaxies allows scalable processing of large volumes of data on a cluster in realtime without running a long-period jobs to calculate large matrices, and allowing the flexible feeding of interesting objects to the data with immediate retrieving the result of analysis. The system is useful for monitoring the modern cluster installations that include thousands of servers, each having multiple parameters monitored, including the computation components temperature, hard drive parameters, cooling fans RPMs and so on. Wireless sensors deployed in the server room provide additional information that also should be taken into consideration. Monitoring such a system is a complex task and our streaming PCA algorithm can indicate latent features and correlations in cluster health, where a significant eigensystem deviation could indicate a hardware failure. Prediction and timely reaction to the changes can significantly improve the cluster reliability and decrease the system downtime.

REFERENCES

[1] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," in *NIPS*, 2006, pp. 281–288.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, Apr. 2012.

[3] K. Kanov, E. A. Perlman, R. C. Burns, Y. Ahmad, and A. S. Szalay, "I/o streaming evaluation of batch queries for data-intensive computational turbulence," in *SC*, 2011, p. 29.

[4] I. Jolliffe, *Principal Component Analysis*. Springer Verlag, 1986.

[5] J. Sun, D. Tao, S. Papadimitriou, P. S. Yu, and C. Faloutsos, "Incremental tensor analysis: Theory and applications," *TKDD*, vol. 2, no. 3, 2008.

[6] Y. Li, L. qun Xu, J. Morphett, and R. Jacobs, "An integrated algorithm of incremental and robust pca," in *Proceedings of IEEE International Conference on Image Processing*, 2003, pp. 245–248.

[7] R. Maronna, "Principal components and orthogonal regression based on robust scales," *Technometrics*, vol. 47, no. 3, pp. 264–273, Aug. 2005. [Online]. Available: http://dx.doi.org/10.1198/004017005000000166

[8] P. J. Huber, *Robust statistics*. Wiley, New York, 1981.

[9] T. Budavari, V. Wild, A. S. Szalay, L. Dobos, and C.-W. Yip, "Reliable eigenspectra for new generation surveys," 2008.

[10] R. Everson and L. Sirovich, "Karhunen-lo'eve procedure for gappy data," pp. 12, 8, 1995.

[11] A. Connolly and A. Szalay, "A robust classification of galaxy spectra: dealing with noisy and incomplete data," *Astron.J.*, vol. 117, pp. 2052–2062, 1999.

[12] C.-W. Yip *et al.*, "Spectral classification of quasars in the Sloan Digital Sky Survey: Eigenspectra: redshift and luminosity effects," *Astron.J.*, vol. 128, pp. 2603–2630, 2004.

[13] V. Wild, G. Kauffmann, T. Heckman, S. Charlot, G. Lemson, J. Brinchmann, T. Reichard, and A. Pasquali, "Bursty stellar populations and obscured active galactic nuclei in galaxy bulges," *Monthly Notices of the Royal Astronomical Society*, vol. 381, no. 2, pp. 543–572, 2007. [Online]. Available: http://dx.doi.org/10.1111/j.1365-2966.2007.12256.x