Real-time analysis and management of big time-series data

A. Biem
H. Feng
A. V. Riabov
D. S. Turaga

The ability to process and analyze large volumes of time-series data is in increasing demand in various domains including health care, finance, energy and utilities, transportation, and cybersecurity. Despite the broad use of time-series data worldwide, the design of a system to easily manage, analyze, and visualize large multidimensional time series, with dimensions on the order of hundreds of thousands, is still a challenging endeavor. This paper describes the Streaming Time-Series Analysis and Management (STAM) system as a solution to this problem. STAM provides the capability to glean actionable information from continuously changing time series with thousands of dimensions, in real time. STAM exploits the IBM InfoSphere® Streams platform and allows for general-purpose large-scale time-series analytics for applications including anomaly detection, modeling, smoothing, forecasting, and tracking. In addition, the system provides user-friendly tools for managing, deploying, and initiating analytics on large-scale data streams of interest, and provides a web-based graphical visualization interface that allows highlighting of events of interest with interactive menus. In this paper, we describe the system and illustrate its use in a large-scale system-monitoring application.

Introduction

In this paper, we use the term *time series* to refer to any numerical data generated sequentially through time and associated with timestamps, either implicitly or explicitly. Examples include measurements obtained from sensors, log records generated by software (and translated to numerical metrics), load metrics obtained from data-center equipment, physiological data from health care-monitoring devices, etc. A *univariate time series* exhibits the value of a single variable through time (e.g., the sequence of daily temperature in New York). A *vector time series* exhibits a list of multiple variables values through time (e.g., a sequence of daily temperature and humidity level in New York). The dimension of that vector is the dimension of the time series. *Big time series* refers to large-dimensional vector time series, with a dimension on the order of hundreds or thousands. Big time-series processing is equivalent to processing hundreds or thousands of univariate time series in parallel. The capabilities to process and analyze such a large volume of time-series data in real time, in order to make use of as much information as possible, are in increasing demand in many industries and applications, including data-center monitoring, health care, finance, transportation, cybersecurity, environmental monitoring, and energy and utilities.

Due to this increasing usage of time-series data worldwide, efficient analysis of high-rate time series with thousands of dimensions requires specially designed algorithms and implementations. To our knowledge, there are scant examples of systems or frameworks that provide an integrated capability to manage, analyze, and visualize very high-dimensional time-series data in real-time derived from multiple sources. Most time-series packages or software provide the capabilities to analyze or manage time series up to few hundred dimensions, mostly applied to non-streaming data. Notable examples include the GEneralized Multimedia INdexIng (GEMINI) [1] system and related algorithms [2, 3]. GEMINI provides retrieval possibilities for time series "at rest" (e.g., information stored in large databases), and its focus is on retrieval and storage.

Digital Object Identifier: 10.1147/JRD.2013.2243551

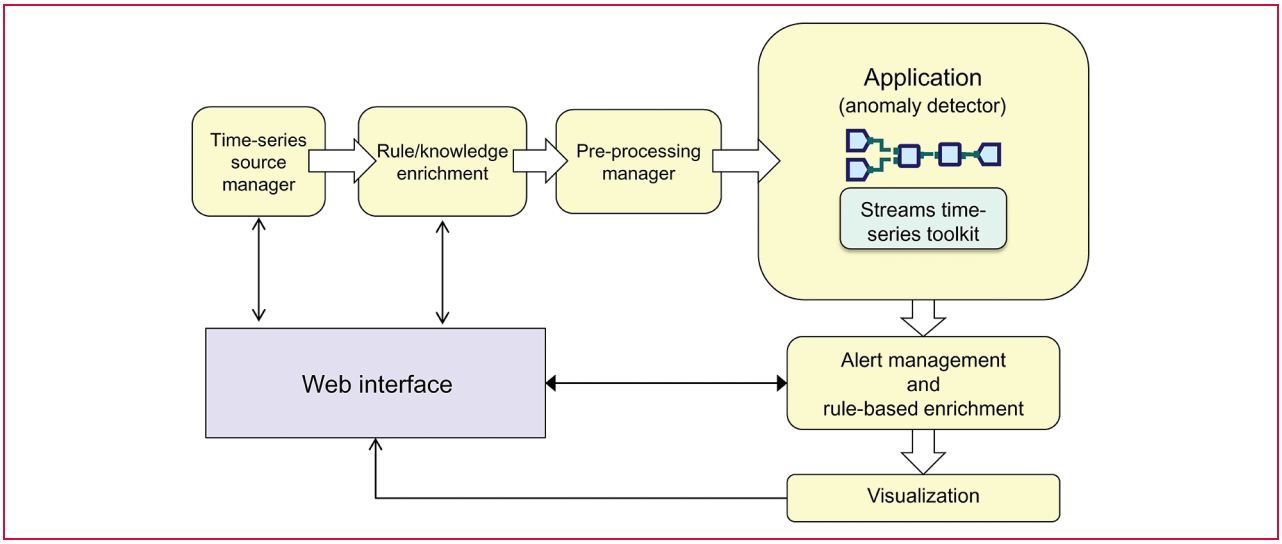

Figure 1

STAM overall architecture. The system involves a source manager that ingests and aligns incoming data. The Application module hosts various time-series applications packaged as a toolkit. The Visualization module interacts with the user through a web interface and provides feedback and alerts.

Others systems, such *StatStream* [4], deal with time series in real time but with a focus on detecting short-term cross-correlations. StatStream is essentially a set of algorithms for estimating cross-correlations in streaming time-series data; it is not a large-scale time-series management system.

This paper describes the Streaming Time-Series Analysis and Management (STAM) system as a more comprehensive solution for real-time time-series management. STAM belongs to a new generation of general-purpose time-series analytics and management platforms for analyzing high-dimensional time series in real time and with scaling capability (i.e., it allows an increase in dimensionality at runtime). It can be used to implement or deploy new or existing state-of-the-art, scalable, multi-dimensional time-series applications including anomaly detection, modeling, forecasting, and tracking. It enables the development and integration of new analysis techniques for real-time processing.

As suggested, from the user perspective, STAM provides the capability to monitor and interact with thousands of constantly changing time series, in real time through an easy-to-use web interface. It comprises a variety of on-demand analytics features with self-learning and self-tuning capabilities. The analytics processes can automatically adapt to data and include automatic data conditioning, transformations, and modeling. Those analytics are designed to be robust with respect to noise and tolerant with respect to low data quality such as missing data or noisy data, a critical challenge in most data-monitoring systems. Feedback to the user is provided by alerts and notifications with customizable severity controls via a web-based graphical visualization system with interactive menus. The intent is to help focus user attention on key observations by providing valuable insights into important changes and trends in data movement, while requiring as little input from the end user as possible. The system will automatically adapt to the characteristics of the data to appropriately modify the analysis. A version of the STAM system is currently being deployed at the IBM Industry Solution Lab in New York as an example of large-scale data processing.

The goal of this paper is to describe STAM in detail, from both the system perspective and the algorithm perspective. The paper is organized as follows. In the first section, we provide an overview of the STAM components. In particular, we consider the data-loading mechanism, application deployment, and visualization. In the second section of the paper, we describe one underlying application available in STAM, namely a generic real-time anomaly-detection application for large-scale time-series data. We include performance evaluation in the third section and conclude the work in the last section.

System overview

The STAM architecture is illustrated in **Figure 1**. As mentioned, the system comprises a web-based user interface that is responsible for the interaction with the user and the transmission of the user's requests to other components. The web interface communicates with a Time-Series Source

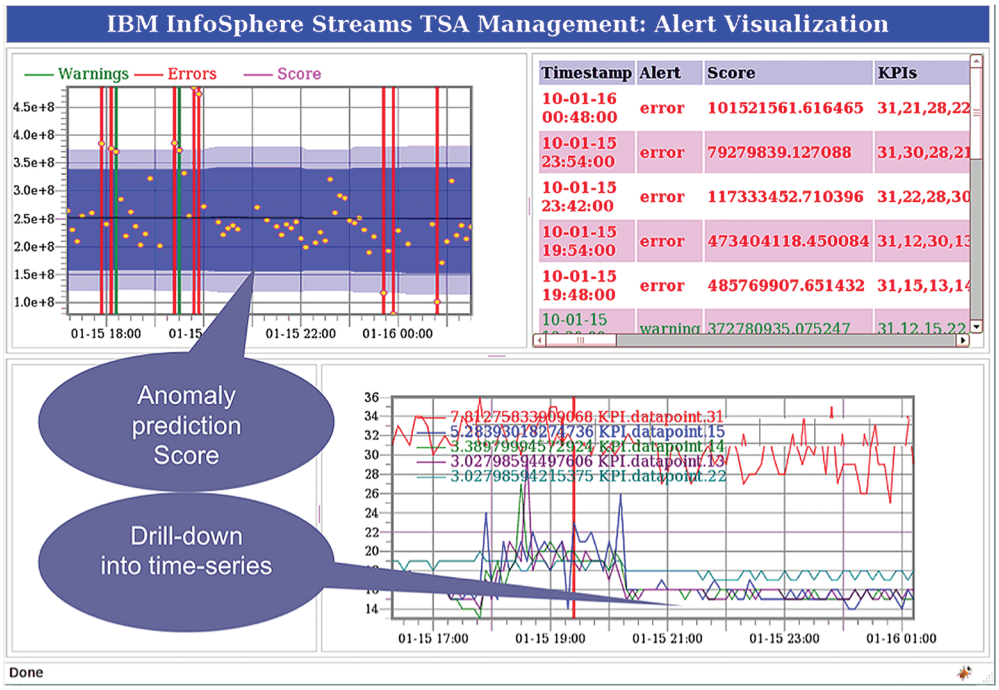

Figure 2

The STAM web-based interface. The upper-right menu shows the state system scores. At upper left, each dot illustrates an anomaly score. A score in the blue area refers to a normal state of the system. A score in the purple area refers to an alarm state and will generate a warning. A score in the white area refers to an anomaly and will generate an alert. The right menu displays the actual value of the score, the alert level (e.g., normal, warning, and error), and the time. The bottom window shows the subset of time series that has triggered the anomaly, along with the related time interval.

Manager, responsible for loading data. The Time-Series Source Manager also uses the service of a preprocessing module responsible for joining and aligning time-series data in real time. The time-series samples are aligned along a single timestamp within a vector to allow multivariate analysis, and time-series data is treated as a multi-dimensional sequence of data. For the alignment process, we adopt a common factor for the unit of time across incoming samples. For instance, for two time series sampled, with a 15-second delay, they necessarily share the same 30-second time interval.

Once aligned and synchronized, the time-series information is represented as a vector time series (a sequence of vectors, where each component in the vector represents the data of a single-variable time series, and all components in the vector share the same timestamp). This data is then passed to the *Application module*. The Application module hosts several real-time time-series processing applications. These applications run on the IBM InfoSphere* Streams middleware [5] as detailed in the next section. The choice of IBM InfoSphere Streams as the underlying computational platform was motivated by the desire to achieve scaling to increasing data volumes in real time, with support for adding new hardware compute nodes when necessary. This point is further explained in the next section. The applications are deployed by means of a *Deployer Services* module, whose role is to schedule and launch candidate applications on the IBM InfoSphere Streams platform.

The applications themselves use algorithms implemented as analytics, deployable on InfoSphere Streams, and are designed for real-time processing. The output of the application is returned to the *Web Interface* via a real-time visualization system as illustrated in **Figure 2**. The Web Interface provides the user with various menus for data source selection, choice of analytics, configuration parameters, and navigation for focusing on areas of interest in the time-series data. Once the choice of analytics is made, the user can initiate the run of these analytics using the *Deployer Services* component (see later sections) and then be able to visualize the output and focus visualization on identified events of interest in real-time using a web browser.

Streaming computing platform

At the core of the proposed system is a real-time processing middleware called IBM InfoSphere Streams [5], or simply

Streams. As mentioned, InfoSphere Streams is an IBM software product that supports high-performance stream processing. It has been used in a variety of sense-and-respond application domains, from environmental monitoring to algorithmic trading. It offers both language and runtime support for improving the performance of sense-and-respond applications in processing data with a high arrival rate. InfoSphere Streams supports structured as well as unstructured data stream processing and can be scaled to a large number of compute nodes.

Fundamentally, InfoSphere Streams is a programmable software platform aimed at facilitating the development of stream computing applications. An InfoSphere Streams application can be viewed as directed graphs, where nodes are operators (independent processing units). Each operator has a set of input and output ports, where streaming data is received and produced, respectively. The arcs between nodes represent data flowing between operators. Each operator performs a specific task, according to the data it receives (such as data normalization or filtering) and passes the results to the next operator for processing as constrained by the graph. In addition, the InfoSphere Streams platform provides special operators that are used to form an interface with the external environment to either collect data in a streaming fashion (i.e., the source operators) or externalize the output made by the system (i.e., the sink operators). The InfoSphere Streams runtime can execute a large number of long-running jobs (queries) distributed across a variety of machines, because operators composing an application graph can be easily distributed across different nodes. Scalability is also easily achieved by increasing the number of nodes for computation without an explicit need for new software development [6, 7].

Programming InfoSphere Streams is greatly facilitated with a declarative programming language called the *InfoSphere Streams Processing Language* (SPL) [5], which shields developers from the complexity of the internal InfoSphere Streams application programming interfaces (APIs). InfoSphere Streams and SPL provide a rapid application development front-end for streaming data and offers the following features. It provides an intermediate language for the flexible composition of parallel and distributed data-flow graphs. In addition, it offers a standard toolkit of built-in processing operators supporting relational operators with rich windowing semantics, including *Functor* (functional and filtering operations), *Join* (union of separate data streams), *Aggregate* (summarizations and basic statistics), *Barrier* (synchronization of data streams), and much more. A broad range of stream adapters are used to ingest data from outside sources and publish data to outside destinations, such as network sockets, relational and XML (Extensible Markup Language) databases, and file systems.

Additionally, the standard toolkit may be extended with user-defined operators, called *Primitive Operators*, programmable in either C++ or Java**. Developing Primitive Operators is facilitated the IBM InfoSphere Streams integrated development environment. This allows for the development of toolkits as a library of algorithms developed for InfoSphere Streams. Our core analytics in the STAM system are implemented as a toolkit.

Application Deployer service

Applications developed for the IBM InfoSphere Streams middleware platform are compiled from source code written in SPL and are deployed for execution using the InfoSphere Streams scheduler and runtime services. The *scheduler* makes placement decisions, deploys individual operators to cluster nodes, and establishes data streams.

We have developed an additional service for STAM, the *Application Deployer*, which provides a web-based interface for parameter setting and applications deployment for end users. The deployed applications are constructed from templates provided by developers, and the end users do not need to write code or learn the IBM InfoSphere SPL. Via the web interface, end users can select and activate data sources and parameterize and deploy time-series analytics. As mentioned, using the same interface, end users can also view the results in real time, as data is being ingested from the sources and processed.

New data sources can be easily added to the *Application Deployer* by providing a configuration file. In a typical deployment, STAM can be used as soon as it is connected to sources, and our configuration-based approach makes this operation significantly easier than developing a new stream processing application.

Time-series visualization

As part of the Application Deployer, as mentioned, we have developed a real-time web-based visualization for results of time-series analysis. Our web-based visualization framework consists of a server that accumulates data streams and provides access to resulting ring buffers via HTTP (Hypertext Transfer Protocol) and a browser-based client library written in JavaScript**.

Data received from time-series analytics in InfoSphere Streams is accumulated in the memory of the visualization server. We had to address the issues associated with accumulating and delivering data to the browser, as well as dynamically updating browser-based visuals. Each deployed analytic creates a new ring buffer. The maximum size of the ring buffer is fixed, and when the buffer is completely filled, the new records replace the oldest ones. At high data rates, the limited rate at which data is written to buffers may create a "backpressure," slowing down processing; thus, we have created safeguards that reduce the rate by resampling data if the server rate limit is reached.

Large-scale, real-time system monitoring

Background

Large systems such as cloud infrastructures and enterprise data centers require constant supervision to ensure that they are continuously operational. The monitoring objective is to ensure high availability of the center and enable uninterrupted service. Monitoring such systems and detecting anomalous behavior from large collected data streams is not a trivial task: the large number of servers (sometimes spanning diverse geographical areas), their traffic, the density of computational clusters, and the complex interaction among systems components pose serious monitoring and management challenges. The large size of the system makes it vulnerable to a wide range of anomalies, including performance bottlenecks, software errors, memory leaks, firmware or middleware challenges (e.g., networks problems), or hardware failures (e.g., disk malfunctions).

Real-time anomaly detection refers to the process of finding areas in incoming data streams that do not conform to expected behavior. For example, in the data-center monitoring scenario, an anomaly is a deviation of the normal operation of the center and could be short-lived (a few seconds or minutes), long-lived (a few hours or days), or catastrophic (system stops functioning). A real-time data-center monitoring system should have the following characteristics: scalability, adaptability, and the ability to correlate multiple time series simultaneously (multivariate analysis). Scalability is a key element and is required to accommodate the growth in the number of sensors at runtime. For example, in data-center monitoring or patient monitoring in health care, it is typical to add new sensors to existing ones, and doing this without requiring a halt to the whole system is a desirable capability. This also means that the number of univariate time series for a large and complex infrastructure can grow to hundreds of thousands, which is beyond the reach of most standard monitoring systems.

Adaptability is necessary in order to account for the changing statistics of the data through time. Multivariate analysis is preferred to account for multiple correlations among time series, such as correlation of metrics within machines or within a group of machines. In addition, the underlying algorithm should be self-learning. Self-learning is desirable because supervised techniques are usually not possible in such contexts, as the conditions change constantly, creating a discrepancy between the data gathered at training and the one used for testing. Furthermore, examples of anomalies are usually not available to allow for a supervised approach.

Finally, robustness with respect to data quality (missing data or noisy data) is desirable: sensors may not produce reliable data, leading to a rather sparse or noisy time series.

Prior work

A large amount of research on anomaly detection has dealt with data "at rest" (e.g., data residing in a database or file) and rarely with data in motion (i.e., streaming) [8]. The technique typically uses the entire set of data or a partial set of data, or may require that the entire dataset be placed in memory [9].

The work by Lakhina et al. [10, 11] has popularized the use of the subspace method in anomaly detection. Their method projects the time series into subspaces, typically generated by a principal component analysis (PCA) [12]. A PCA performs a linear transformation of the original data into subspaces of principal components where data is less correlated. Higher indices in the principal component subspace correspond to normal behavior, and the remainders of subspace models correspond to abnormal behaviors (residuals). An anomaly is flagged when the magnitude of the residuals violates a Q-statistic threshold [10]. This algorithm, however, requires a central processing unit (CPU)-extensive estimation of the covariance matrix Σ based on the entire dataset and, as such, is not amenable to real-time usage.

For a real-time scenario, the algorithm should operate under the constraint that all of the data is not available, and incoming data may be received at a rate faster than can be processed. An online version of the algorithm has been proposed [11], which uses a sliding window to identify normal and abnormal subspaces. However, the results are very sensitive to the size of window, and there is no guideline as to which size is optimal, given a task. The SPIRIT algorithm [13] does not use a window of data but directly implements an online estimation of the principal subspace, where principal components are incrementally estimated over time. Likewise, the KOAD (Kernel-based Online Anomaly Detection) approach [14] uses a kernel-based recursive least-square regression technique to detect anomalies. The regression is incremental. The incoming data are iteratively projected in a kernel-based subspace on which an online recursive least-square regression is performed. The distance to the subspace is used as a score to detect anomalies in real time. All those algorithms were devised in the context of computer network-based anomaly detection.

Our aim is a generic real-time multivariate anomaly detection algorithm, based on a plug-and-play philosophy whereby the user simply provides the data to be monitored, and the system performs the detection. Necessarily, we assume the non-availability of a training set or exemplar set of anomalies. Our system should be able to infer the normal state of the system without explicit examples of abnormal events. Our anomaly detection algorithm is called the Adaptive Classification of Anomalies (ACA). ACA is designed to overcome various obstacles including high data volumes, scalability

issues, changing data statistics, lack of training data, and variable data quality. Our ACA algorithm is time-synchronous, lightweight (e.g., requiring minimal computational cost and memory requirements), adaptive, and unsupervised; and it is well suited to monitor hundreds and thousands of univariate time-series data in real time and able to detect anomalies in large and complex systems.

The basic assumption of ACA is that early data is assumed to be non-anomalous, enabling the algorithm to bootstrap itself efficiently from this early data. The algorithm assumes that the time-series data is collected from the sensors and arrives in streaming fashion, one sample (vector of data) at a time. The output of the algorithm is the classification of each incoming sample as anomalous or normal. This is the detection process. In addition, once an anomaly has been flagged, the algorithm identifies the components within the current sample that are the cause of the abnormal behavior. This is the diagnostic process. This means that the algorithm is specially designed to deal with a large number of time-series data derived from various monitoring sensors from a complex system while also identifying areas in the system that are the cause of the anomaly. ACA processes data in a reasonable time to enable the system administrator to take preventive measure whenever anomalies are flagged. ACA was developed and tested in a variety of projects and domains, including the IBM Tivoli Performance Monitoring product, cybersecurity, and health care.

We assume that time-series data streams contain values of multiple time series sharing the timestamp t and represented as a sequence of vector $\mathbf{x}_t = (x_t(1), x_t(2), \ldots, x_t(n))$, where t increases indefinitely through time, and the dimension n represents the total number of time series being monitored at a given time. The components of $\mathbf{x}_t$ are heterogeneous because each sensor collects various data from various parts of the system. $\mathbf{x}_t$ represents the output of the Time-Series Source Manager (as explained in the previous section). ACA processes one vector at the time, meaning that no time-series windowing is required. The main analysis steps are illustrated in **Figure 3**.

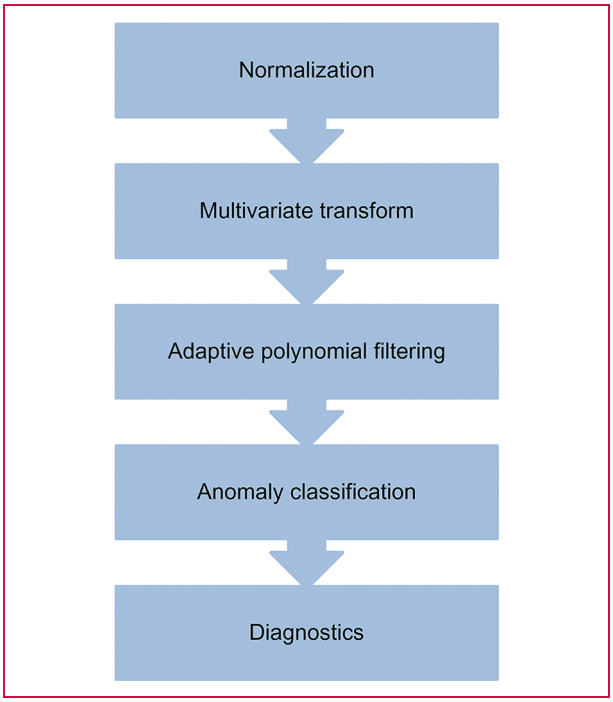

Figure 3

The ACA processing steps. Incoming time series are normalized and then passed to a multivariate transform. An adaptive polynomial filter tracks data movements on the transformed space and produces residuals. The residual statistics are then used to detect anomalies. Diagnostics are performed to signal the faulty time series.

Real-time data normalization

The values within a time-series $\mathbf{x}_t$ are usually heterogeneous, representing data from various sensors and having different ranges. Some components in the time-series vectors are bounded, such as data representing percentile (e.g., CPU usage or system usage rate). Others, such as heap size or memory consumption, can have a greater range of values. The normalization process is a preconditioning step that transforms the original data into a multivariate time series with approximately zero mean and unit variance. In our real-time setting, the mean and variance are approximated by estimating them through early data using an incremental process as follows: Each incoming sample $\mathbf{x}_t$ is normalized in real time as

$$\mathbf{z}_t = \Sigma_t^{-1}(\mathbf{x}_t - \mu_t).$$

The mean μ_t nd covariance matrix Σ_t at time t are also incrementally estimated as

$$\mu_t = \mu_{t-1} + \frac{(\mathbf{x}_t - \mu_{t-1})}{t}$$

$$\Sigma_t = \frac{t}{t-1}\Sigma_{t-1} + \left[\frac{(\mathbf{x}_t - \mu_t)(\mathbf{x}_t - \mu_{t-1})^{\mathrm{T}}}{t}\right],$$

where T is the transpose operator. Given our focus on a lightweight algorithm, the covariance matrix is constrained to be diagonal during the estimation process, meaning that each time series is individually normalized. The process of normalization starts as soon as data becomes available from sensors. We typically use one week of data for estimating the means and variance.

This process is implemented as an operator on the IBM InfoSphere Streams platform. In addition, some noise removal is performed using a simple exponential filter.

Multivariate transform

Multivariate analysis treats the input vector as a single entity by taking into account the correlation between components. This process is typically performed by a vector transform that translates the original time series into new space that captures the correlation between time series. The transform could also be embedded in the modeling scheme as performed when using a multivariate regression model. Optimally, the transform should de-correlate the data, so that the transformed time series represent the system through the prism of various "lenses" that capture the cross-correlation between time series. Anomalies are thus easier to track in the transformed space than in the original ones. As stated previously, PCA is the standard technique used for this process but requires an offline estimate of the covariance matrix. Furthermore, it does not easily accommodate new time series being added at runtime.

Online alternatives, such as the above-mentioned SPIRIT or KOAD algorithms [13, 14], implement an incremental estimate of the covariance matrix but are prone to under-optimality and sensitivity to noise. In addition, similar to offline techniques, online estimation of covariance matrix cannot cope with time series that exhibit increasing dimensionality in real time.

Our solution is to use a vector-by-vector transform that focuses on fusing the original time series into a new time series, where individual data movements relative to each other are emphasized. The normalized data is transformed as follows:

$$\mathbf{y}_t = \mathbf{T}(\mathbf{z}_t)$$

at each timestamp t. Note here that the transform $\mathbf{T}$ is not estimated from data, as with the PCA technique, but provided a priori. This ensures that the transformation can accommodate the increasing dimension of the time series since it acts on the sample-by-sample basis.

Reasonable choices for the transform include the discrete cosine transform (DCT) [15] and the discrete wavelet transform (DWT) [16]. These algorithms were implemented as operators running on IBM InfoSphere Streams. Per default, the ACA algorithm uses the DCT operator with the DWT operator available as an option. Our choice for both derived from their excellent compression properties. In particular, DCT has been proven to generate less-correlated transformed data and, as such, is a good alternative to PCA [15].

Adaptive polynomial filtering

Time-series anomalies are deviations from normalcy. Normalcy could be provided in the form of examples when operating under supervised mode. However, this approach is not possible for large-scale anomaly detection where abnormal events can occur in various shapes and forms. It is easier to model the normal state and then classify as abnormal any state of the system that is far outside a tolerable range of that normal state. To do this, we make use of an adaptive filter that tracks the expected normal state of the system.

Adaptive filtering is the class of filtering algorithms that can self-learn and can adapt their internal parameters to the changing data statistics. They are typically used for tracking, smoothing, or one-step prediction. Typical examples include the Kalman filter [17] and the Particle filter [18]. Adaptive filtering is the fundamental component that enables ACA to be adaptive and self-learning.

As mentioned, because our focus is on a lightweight and scalable predictor (i.e., with minimal computational cost and memory requirements), we avoid the use of CPU-costly standard tracking filters or auto-regressive approaches, and instead make use of the Recursive Fading-Memory filter [19], implemented as an IBM InfoSphere Streams operator.

Recursive polynomial filters [20] provide a time-synchronous fit to the time series by approximating its trajectory by a polynomial of a specified order and can be used to estimate and update the parameter of the filter in real time at an extremely minimal computational cost. The filters are recursive because they do not operate in batch mode, but work in a time-synchronous fashion; they only need the result from the previously predicted sample and the new sample to be able to predict the next sample. Furthermore, these recursive filters are ideally suited to pipeline-type data processing such as the one provided by the InfoSphere Streams platform.

The recursive fading-memory polynomial filters locally represent the time series as a weighted sum of discrete Laguerre polynomial of a given order [19]. The weights are estimated incrementally to minimize a weighted mean square error. The filters are called fading-memory because older data samples are progressively weighted less than recent samples. This gives the filter the ability to adapt to changing time-series statistics. The filters are compact and therefore very fast: only a few additions and multiplications are needed at each time step.

An FMPFilter of order 1, as used in this paper, is described as follows. Let y_t represent one of the components of the multivariate vector $\mathbf{y}_t$ at time t. Let ε_t be the prediction error measured at time t for component y_t, then $\varepsilon_{t-1} = y_{t-1} - \hat{y}_{t-1}$, where $\hat{y}_{t-1}$ is the predicted sample at time $t-1$ after observing data up to time $t-2$. Let dy_t be the first-order derivative of y_t, $d\hat{y}_t$ its predicted derivative, and θ a time constant. The predicted value at time t, $\hat{y}_t$, is estimated as follow:

$$\begin{bmatrix} \hat{y}_t \\ d\hat{y}_t \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} y_{t-1} \\ dy_{t-1} \end{bmatrix} + \begin{bmatrix} 2(1-\theta) \\ (1-\theta)^2 \end{bmatrix} \varepsilon_{t-1}.$$

The filter produces an estimate of the expected value $\hat{y}_t$, based on previously estimated errors. Likewise, the filter

also produces an estimate of the variance, σ_t, around the predicted value by making use of the past N samples up to timestamp t, and the residual ε_t, where N is a parameter specified by the user.

Anomaly classification

It is standard practice to estimate an anomaly score using the χ^2 statistics [21] defined as

$$\chi^2 = \sum_{i=1}^{n} \frac{\varepsilon_t^2(i)}{\hat{y}_t(i)}.$$

This statistic, however, is not robust with respect to change in the dimension of the time series at runtime. As an alternative, we propose the following anomaly score:

$$s^2 = \frac{2}{n(n+1)} \sum_{i=1}^{n} i \frac{\varepsilon_t^2(i)}{\sigma_t(i)}.$$

This statistic weights the variance-normalized prediction errors by their indices and is robust with respect to dimensional changes.

This statistic represents the *state score* of the entire system at one point in time, and it is the value displayed in the STAM monitor screen (see Figure 2). This statistic is passed to an inter-quartile operator (IQR). The IQR operator [22] produces a real-time histogram and first-order statistics of scores in real time per time bin (specified prior to analysis). The statistics computed are as follows: the smallest non-outlier, the lowest quartile (first quartile), the median, the upper quartile (third quartile), and largest non-outlier, all estimated incrementally through time for each time bin. For instance, statistics can be estimated per hour, yielding a distribution for each hour of day that describes the expected behavior of the system within each hour of the day. This flexibility in granularity of analysis is one of the strengths of the ACA algorithm.

The inter-quartile interval *iqr* defined as the distance between the first and third quartile is computed in real time and is the basis for producing the confidence measure of an anomaly as follows. First, we perform a variable transform as follows:

$$\phi(s) = \frac{\pi(s - s_{\text{med}})}{iqr},$$

where s_{med} is the median value of the score, and the *iqr* is the inter-quartile interval derived from the statistics. The confidence measure is the error function defined on the transformed statistics as

$$c(\phi) = \frac{2}{\sqrt{\pi}} \int_0^{\phi} e^{-\tau^2} d\tau.$$

An incoming sample is classified as an anomaly when $c(\phi)$ exceeds a specified threshold between 0 and 1. A threshold of 0.95 is typically used.

Diagnostics

Once anomalies have been flagged, the next step is to perform a diagnosis that yields the list of faulty metrics. Faulty metrics are displayed as indices and time of occurrence in the original time series responsible for the anomaly, as illustrated in Figure 2. To accomplish this, we estimate the *rate of change* for each time series of index i as

$$r(i) = \frac{(z(i) - \hat{z}(i))}{\sigma^2(i)},$$

where $\sigma^2(i)$ is the estimated variance as provided by the polynomial filter, and $\hat{z}(i)$ is the i-th component of inverse transform $\hat{\mathbf{z}} = \mathbf{T}^{-1}(\mathbf{y})$, representing an estimate of the normalized time-series at index i, derived by performing a reverse inversion of the transformed time $\mathbf{T}^{-1}$.

The values are then ranked according to $r(i)$ in decreasing order, and a top list is selected from a specified threshold, yielding the most likely time-series at fault at that particular timestamp.

Performance evaluation

Detection performance

We evaluated our approach on synthesized anomalies, generated for data collected from a data center of large financial company. We gathered 514 time series representing data from the data centers. The data was collected every six minutes for 28 days. During the simulation, the data was replayed at a faster rate. Various testing sets were devised by randomly selecting a subset of time series from the data-center data (ranging from 8 to 20). Depending on the test, various levels of white noise were added to the time series, or a subset of time series made of pure noise was included as a new time series. The goal was to make the testing as challenging as possible.

Typical data-center anomalies were introduced within a subset of time series. We devised nine categories of anomalies typically found in the data center or cluster of servers. For each category, we devised a set of tests, resulting in 225 tests (approximately 25 tests per category). For each category, the algorithm is run on each test and is intended to classify each incoming sample as anomalous or not. Since this is a completely unsupervised process, there is no separate training phase versus testing phase. Instead, the algorithm uses the first samples to bootstrap itself, including filter initialization, the estimation of the number of time series, and the estimation of the sampling rate based on the timestamps. Once these meta-parameters have been

Table 1 Summary of performance results across all tests. The term *specificity* refers to how useful a test is in avoiding false alarms, and it is computed from the ratio of the number of true negatives over the sum of true negatives and false positives.

Performance metrics	*Results*
True positive	53,638 samples
False positive	12 samples
True negative	1,736,934 samples
False negative	25,976 samples
Precision	100%
Recall	67%
Specificity	100%
Accuracy	99%
F score	80%

estimated, the algorithm learns, adapts, and scores incoming data in real time as described in the previous section.

The categories of anomalies devised for the test include the following:

- *Decreasing gradual*—A subset of the time series gradually decreases at some point before returning to normal. In other words, there are multiple time series running in parallel, and a subset of those time series are exhibiting values that decrease. This simulates a loss in performance within a node and could be a precursor to a potential shutdown.
- *Decreasing sharp*—A sudden decrease in values within a subset of time series. This type of anomaly typically suggests hardware problems and can shut down an entire system.
- *Drop sharp*—A set of time series values suddenly decrease in value. This may be due to a power failure in one of the system components.
- *Increasing sharp*—A subset of time series slowly increases in values. This is typical of memory leaks.
- *Increasing square*—A subset of time series has a sudden increase in value, reaching a plateau for a certain period of time, and then dropping suddenly. This suggests a system overload.
- *Increasing top*—A sudden increase in a subset of time series, followed by a sudden decrease. This is typical of a power surge.
- *Loss*—A sudden disappearance of a subset of time series. This could either suggest a failure in the sensing infrastructure or a component shutting down unexpectedly.
- *Spikes*—Outlier spikes may suggest an unusual behavior of the system that needs to be monitored.
- *Spike gradual*—A gradually increasing and decreasing of a subset of time series.
- *No anomalies*—No anomaly is in the set. This is to test the ability to accurately detect false alarms.

Table 1 shows the overall results of the algorithm for the entire test set. In the table, *true positive* indicates the detection of anomalous samples, whereas *false positive* indicates the incorrect classification of samples as anomalous. Likewise, *true negative* indicates the correct classification of samples as normal, and *false negative* indicates the incorrect classification of samples as anomalous.

We achieved an accuracy of 99% (proportion of correctly classified samples). However, raw accuracy is not a reliable performance measure for anomaly detection. Instead, we should examine the *precision* (proportion of the correctly detected anomalies among retrieved anomalies) and the *recall* (proportion of detected anomalies among all available anomalies). In particular, the F score, which makes use of precision and recall, is a more robust measure of detection accuracy. The F score is defined as

$$F = 2\,\frac{precision \times recall}{precision + recall}.$$

It is the harmonic means between precision and recall. A higher precision or recall yields a higher F score and vice versa.

We achieved an almost 100% precision for all tests and all categories, meaning that most samples categorized as anomalous were actually anomalous. An F score of 80% is remarkable performance for a detection algorithm. This provides a high degree of specificity to the algorithm.

A recall of 67% can be better understood when analyzing the results for each category as summarized in **Table 2**. Table 2 shows the results of our proposed algorithm for various types of anomalies. Our proposed algorithm is able to detect most anomalies within each category with high efficiency except for the *Increasing Top* category, characterized by a week-long anomalous region. When anomalies have such a longer duration, our system adapts

Table 2 Performance results per type of test. The detection latency refers to the number of first undetected anomalous samples when getting anomalous data. For instance, a detection latency of 2.6 means that algorithm failed to detect the first 2.6 anomalous samples an average within the anomalous region.

Anomaly Category	*Number of tests per category*	*F1 Score*	*Average detection latency (in samples)*
Decreasing gradual	27	83%	3.6
Decreasing sharp	27	87%	0
Drop sharp	27	92%	0
Increasing sharp	27	86%	2.6
Increasing square	27	92%	0
Increasing top	27	68%	2.57
Loss	27	83%	4.5
Spikes	27	92%	0
Spike gradual	27	84%	2.8
No anomaly	9	100%	0

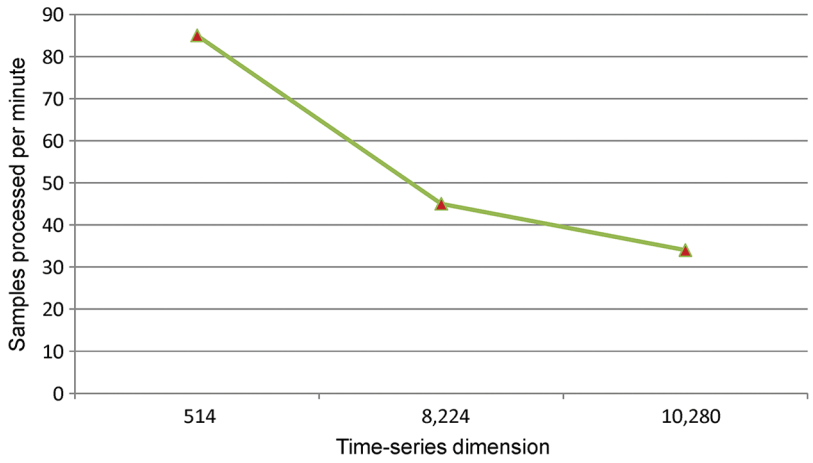

Figure 4

Scalability performance. The *x* axis displays the dimension of the time series (the total number of univariate time series). The *y* axis shows the number of samples (vectors) processed per minute.

to that status, and the region is considered less anomalous as it should be, leading to a false positive.

Scalability

We performed scalability tests using an x86 64-bit 3-GHz Intel Xeon** Quad-Core server. To test scalability, the original data-center set was concatenated 16 times to produce a multivariate time series of dimension 8,224, and then 20 times for multivariate time series of dimension 10,280.

Figure 4 shows the results of the test. Evidently, our system is able to process thousands of data samples with a minor decrease in the processing rate: a 20 times increase in the number of the time series results in only a twofold decrease in processing rate.

Conclusion

We described STAM as a domain-agnostic, multi-component, generic time-series analysis and management system and illustrated its capabilities through experiments on a real-time, large-scale anomaly detection application and synthesized tests.

The STAM system is designed with the particular focus on well-defined characteristics. STAM is a generic plug-and-play system. It provides the ability to handle multi-dimensional time-series data of virtually any type at

any scale. The user simply submits the data source, and the system performs the processing. STAM emphasizes ease of use: the system requires minimal tuning from the user and minimal configuration to start, except for adding data sources. It also allows control (e.g., sensitivity adjustment and parameter selection) for advanced users. STAM analytics are scalable solutions based on the IBM InfoSphere Streams platform, thus being able to process hundreds and thousands of time-series data in real time. In addition, STAM handles data in real time with negligible latency for monitoring and alerts. The user is able to load multiple time series, select any particular class of analysis, and obtain results in real time. One application illustrated in this paper was a large-scale generic anomaly detection in which the user is able to visualize alerts generated by the system. Finally, the STAM analytics are adaptive, as they automatically adapt to changes in data movements. The underlying algorithms are capable of self-learning and do need an offline training set.

The next step in the STAM development concerns several more applications in the STAM Application module: a generic forecaster with short-term and long-term forecasting capabilities, a generic noise removal and smoother tool, and a pattern detector.

Acknowledgments

We thank various colleagues from IBM Research, in particular Mitch Cohen of IBM Research, for providing early feedback in the design of the ACA algorithm, Eric Bouillet for developing an early draft of the IQR operator, and Olivier Verscheure who first initiated this work. We thank the following members of the IBM Tivoli team for their help for the requirements of the ACA algorithm: Robert McKeown, Edie Stern, Alex Pikovski, Ian Manning, and Eric Thiebaut-Georges. Finally, we particularly thank Natalia Udaltsova for help in setting up the testing framework for the ACA algorithm.

References

1. R. Agrawal, C. Faloutsos, and A. Swami, "Efficient similarity search in sequence databases," in *Proc. Found. Data Org. Algorithms*, vol. 730, *Lecture Notes in Computer Science*, 1993, pp. 69–84.
2. R. Agrawal, K. Lin, H. Sawhney, and K. Shim, "Fast similarity search in the presence of noise, scaling, and translation in time-series databases," in *Proc. 21st Int. Conf. Very Large Databases*, Zurich, Switzerland, Sep. 1995, pp. 490–501.
3. C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast subsequence matching in time-series databases," in *Proc. ACM SIGMOD Int. Conf. Management Data*, Minneapolis, MN, USA, May 1994, pp. 419–429.
4. Y. Zhu and D. Shasha, "StatStream: Statistical monitoring of thousands of data streams in real time," in *Proc. 28th Int. Conf. Very Large Data Bases*, 2002, pp. 358–369.
5. IBM Corporation, InfoSphere Streams. [Online]. Available: http://www-01.ibm.com/software/data/infosphere/streams/
6. L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani, "SPC: A distributed, scalable platform for data mining," in *Proc. Workshop DM-SSP*, 2006, pp. 27–37.
7. B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "SPADE: The system S declarative stream processing engine," in *Proc. ACM SIGMOD*, 2008, pp. 1123–1134.
8. V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surveys*, vol. 41, no. 3, p. 15, Jul. 2009.
9. D. Yankov, E. Keogh, and U. Rebbapragada, "Disk aware discord discovery: Finding unusual time series in terabyte sized datasets," *Knowl. Inf. Syst.*, vol. 17, no. 2, pp. 241–262, Nov. 2008.
10. A. Lakhina, K. Papagiannaki, M. Crovella, C. Diot, E. Kolaczyk, and N. Taft, "Structural analysis of network traffic flows," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 1, pp. 61–72, Jun. 2004.
11. A. Lakhina, M. Crovella, and C. Diot, "Mining anomalies using traffic feature distributions," in *Proc. Conf. Appl., Technol., Archit., Protocols Comp. Commun.*, 2005, pp. 217–228.
12. I. Jolliffe, *Principal Component Analysis*, 2nd ed. New York, NY, USA: Springer-Verlag, 2002.
13. S. Papadimitriou, J. Sun, and C. Faloutsos, "Streaming pattern discovery in multiple time-series," in *Proc. 31st Int. Conf. Very Large Databases*, 2005, pp. 697–708.
14. T. Ahmed, M. Coates, and A. Lakhina, "Multivariate online anomaly detection using kernel recursive least squares," in *Proc. 26th IEEE INFOCOM*, 2007, pp. 625–633.
15. K. R. Rao and P. Yip, *Discrete Cosine Transform*. New York, NY, USA: Academic, 1990.
16. C. Chui, *An Introduction to Wavelets*. San Diego, CA, USA: Academic, 1992.
17. A. Harvey, *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge, U.K.: Cambridge Univ. Press, 1990.
18. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for on-line nonlinear/non-Gaussian Bayesian tracking," *IEEE Trans. Signal Process.*, vol. 50, no. 2, pp. 174–188, Feb. 2002.
19. N. Morisson, *Introduction to Sequential Smoothing and Prediction*. New York, NY, USA: McGraw-Hill, 1969.
20. E. Brookner, *Tracking and Kalman Filtering Made Easy*. Hoboken, NJ, USA: Wiley, 1998.
21. S. Subramaniam, T. Palpanas, D. Papadopoulos, V. Kalogeraki, and D. Gunopulos, "Online outlier detection in sensor data using non-parametric models," in *Proc. 32nd VLDB*, 2006, pp. 187–198.
22. A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, "IBM InfoSphere Streams for scalable, real-time, intelligent transportation services," in *Proc. SIGMOD Conf.*, 2010, pp. 1093–1104.

Received July 27, 2012; accepted for publication August 22, 2012

Alain Biem *IBM Research Division, Thomas J. Watson Research, Yorktown Heights, NY 10598 USA (biem@us.ibm.com)*. Dr. Biem is a Research Staff Member at the IBM T. J. Watson Research Center. He received the Diplome D'ingenieur in telecommunications in 1991 and a Ph.D. degree in computer science from the University of Paris, France, in 1997 with *summa cum laude* honors. He was a Research Scientist working on speech recognition at the Advanced Telecommunication Research (ATR) Institute in Kyoto, Japan, before joining the IBM T. J. Watson Research Center in 2000, where he has been involved in various projects on statistical pattern recognition, streaming analytics, and business modeling. He received an IBM Outstanding Technical Accomplishment Award in 2009 for his

contribution to IBM Component Business Modeling (CBM), and an IBM Research Division Award in 2010 for his contribution to online health care analytics. He is the author of numerous papers on signal processing, machine learning, and business modeling. Dr. Biem is a member of the Institute of Electrical and Electronics Engineers (IEEE), the Acoustical Society of Japan, and the International Speech and Communication Association.

Hanhua Feng *(hanhua@gmail.com).* Dr. Feng received his M.S. and Ph.D. degrees in computer science from Columbia University. He joined the IBM Thomas J. Watson Research Center in 2007. He is now working at a startup company. His research interests include modeling and performance analysis, queuing theory, optimization, computer networking, and storage techniques.

Anton V. Riabov *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (riabov@us.ibm.com).* Dr. Riabov is a Senior Software Engineer and Manager of the Automated Component Assembly Middleware department at the IBM T. J. Watson Research Center. He received a B.S. degree in computer science from Moscow University in 1997 and a Ph.D. degree in operations research from Columbia University in 2004. He subsequently joined IBM at the IBM T. J. Watson Research Center, where he has worked on algorithms and systems for automated assembly of software applications. In 2011, he received an IBM Corporate Award for his work on stream processing middleware. He is author or coauthor of 18 patents and 31 technical papers.

Deepak S. Turaga *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (turaga@us.ibm.com).* Dr. Turaga received a B.Tech. degree in electrical engineering from Indian Institute of Technology, Bombay, in 1997, and M.S. and Ph.D. degrees in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, in 1999 and 2001, respectively. He is currently a Research Staff Member and Manager at the IBM T. J. Watson Research Center. His research interests lie primarily in statistical signal processing, multimedia coding and streaming, machine learning, and data mining applications. In these areas, he has published more than 50 papers. He is an Associate Editor of the *Journal of Visual Computing and Imaging* and was an Associate Editor for the *IEEE Transactions on Multimedia*, *IEEE Transactions on Circuits and Systems for Video Technology*, and *Journal for the Advances in Multimedia*.