2015 IEEE 15th International Conference on Data Mining Workshops

Near Real-time Service Monitoring Using High-dimensional Time Series

Shwetabh Khanduja, Vinod Nair, S. Sundararajan
Microsoft Research, Bangalore, India
{shkhandu, vnair, ssrajan}@microsoft.com

Ameya Raul
University of Wisconsin - Madison, Madison, WI
ameyaraul@gmail.com

Ajesh Babu Shaj
Google, Mountain View, CA
Ajesh.babu@gmail.com

Sathiya Keerthi
Microsoft, Mountain View, CA
keerthi@microsoft.com

***Abstract*—We demonstrate a near real-time service monitoring system for detecting and diagnosing issues from high-dimensional time series data. For detection, we have implemented a learning algorithm that constructs a hierarchy of detectors from data. It is scalable, does not require labelled examples of issues for learning, runs in near real-time, and identifies a subset of counter time series as being relevant for a detected issue. For diagnosis, we provide efficient algorithms as post-detection diagnosis aids to find further relevant counter time series at issue times, a SQL-like query language for writing flexible queries that apply these algorithms on the time series data, and a graphical user interface for visualizing the detection and diagnosis results. Our solution has been deployed in production as an end-to-end system for monitoring Microsoft's internal distributed data storage and computing platform consisting of tens of thousands of machines and currently analyses about 12000 counter time series.**

I. Introduction

Enterprises operating cloud-based services need to closely monitor them to ensure that the quality of service received by customers remains high. Internal hardware and software failures, unexpected changes in user load, etc., can result in slow response, unavailability, and violation of service level agreements, leading to customer dissatisfaction and financial penalties. Therefore detecting and diagnosing service issues quickly and accurately is an important problem.

Services are typically heavily instrumented to collect telemetry data about internal operations so as to use the data for detection and diagnosis. *Performance counters* are one type of telemetry data where the values of variables related to service performance (e.g., latencies of API calls, number of internal failures, usage levels of resources such as memory and CPU, etc.) are aggregated at regular intervals (e.g., averaged over one minute intervals). Service health issues appear as "unusual" patterns in the resulting time series data, e.g., as unusually high latencies. To enable timely detection, performance counters are collected and analyzed in real-time. Once an issue is detected, a service analyst starts a diagnosis procedure to find the root cause and the resolution.

Detection and diagnosis of service issues are both challenging problems for several reasons:
1) High dimensionality: Complex services record thousands of counters, so the solution must scale to high dimensional time series data. For example, the service monitoring application we consider in this demonstration involves monitoring more than ten thousand counters. Information from across multiple counters need to be combined for accurate detection and rapid diagnosis. Achieving this in a scalable manner and without burdening the analysts with too many counters to look at is crucial. **2) Unsupervised:** In practice there may not be many known issues available where all the relevant counters that should be used for detecting and diagnosing them are labelled. This is because a) typically there may not have been any effort to log such labels in the past, and b) expert labeling is expensive. As a result, algorithms that require minimal supervision are needed. **3) Real-time requirements:** Since the counter data is being collected in real-time, the solution must be able to keep up with the incoming data rate so that metrics such as time-to-detection and time-to-resolution are maintained at acceptable levels. **4) Sparse subset:** Only a small subset of counters may be relevant for any given issue. For example, if only one sub-component of the service is involved in an issue, it may affect only those counters related to that sub-component. The solution must be able to discover such small subsets from the full set. Exhaustive manual inspection is impractical beyond 10-20 counters. Conventional dashboards are insufficient as they do not provide any significant automation in finding the subset.

While there is a vast literature on time series anomaly detection [3], service monitoring (e.g., [1], [6], [4], [10], [7]), and visualization of outliers in data (e.g., [2], [5]), none of these papers manage to address all the challenges listed above. In this demonstration we show how we addressed all of these challenges in a real-world service monitoring problem. We have built an end-to-end system for detection and diagnosis, starting from the near real-time collection of counter time series data, a SQL-like query language for writing flexible queries to analyse the data for diagnosis, to a graphical user interface for displaying a small subset of counters to the analyst for diagnosing an issue. The system is currently being used by Microsoft's internal distributed storage and computing service, which consists of tens of thousands of machines and is used heavily by its various product groups to run batch jobs on large-scale, distributed data. The demonstration will show how actual issues affecting this service are detected and diagnosed using our solution.

978-1-4673-8493-3/15 $31.00 © 2015 IEEE
DOI 10.1109/ICDMW.2015.254

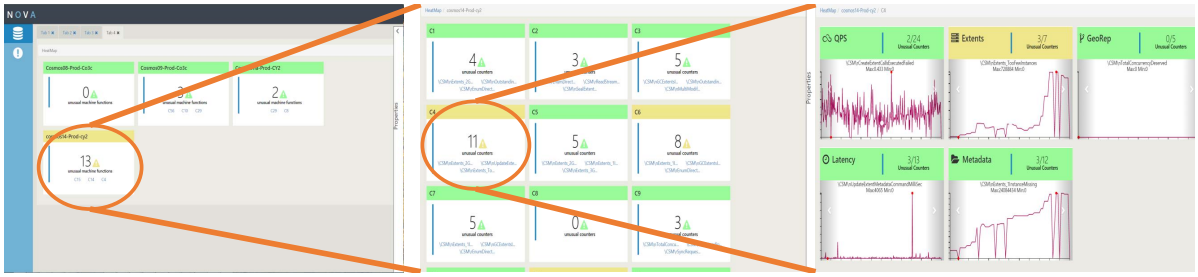

Fig. 1. Heat map view of the service provided by our monitoring system. Tiles in this view represent various service components (left), sub-components (middle), and sub-sub-components (right) with their corresponding counters. Tile colors are set using the output of the hierarchical detection system based on how much unusual behavior is being detected in the corresponding part of the service. In addition to the groupings derived from the service architecture itself, the heat map can also display those *learned* automatically by the hierarchical detection system.

A. Contributions

The main contributions of the paper are:

- An integrated system for detection and diagnosis on near real-time, high-dimensional time series data. Our system currently handles analysis of 12000 counter time series, and can scale to more with additional computational resources.
- Scalable, real-time algorithms developed based on analyst requirements as post-detection diagnosis aids to find relevant counters for an issue.
- End-to-end system deployed in production for a real-world large scale service. It includes a tool for displaying the results of both detection and diagnosis.

II. Integrated Detection and Diagnosis System

The system we demonstrate has two key parts: 1) a hierarchical detection system, and 2) an interactive, query-based diagnosis system. The detection system uses a hierarchy of detectors automatically learned from data for detecting issues affecting the service [8]. This hierarchy is applied in near real-time to the latest counter time series data to detect issues. Once an issue is detected, it can provide the analyst a hierarchical grouping of counters that are suspected to be involved in the issue. This provides the user a starting point for a more detailed investigation using the query-based diagnosis system, which can identify further relevant counters as well as relevant time intervals from the past that can help explain the current issue.

A. Hierarchical Detection System

Our implementation is based on the approach defined in [8]. Due to limited space, we only provide a high-level overview. Please refer to the paper for details.

First, a hierarchy of detectors needs to be learned *offline* from historical counter data. Our system assumes that the learning is already complete and that the detector hierarchy is available to be applied on the new incoming counter time series data. Briefly, the learning algorithm (1) detects "unusual" patterns (defined later) in individual counters independently, (2) automatically discovers groups of related detectors from a large set, and (3) combines the detectors within each group to detect issues characterized by multiple simultaneous unusual patterns in that group. Steps (2) and (3) can be repeated on the time series of outputs produced by detectors themselves to build a hierarchy of detectors. There are many possible choices for defining what an "unusual" pattern is. In this work we have mainly used *change in probability distribution across two consecutive time windows* of a counter, and *low probability values with respect to a probability distribution model* of a counter. Other definitions can also be used without changing the learning algorithm.

The detection system supports a rich exploration experience for the analyst by providing a hierarchical organization of the service based on the learned grouping of counters. With appropriate inputs to the learning algorithm, knowledge about any hierarchical structure in the service architecture itself (i.e., the various service components, sub-components, sub-sub-components, and so on) can be incorporated into the detection system. The results are used to generate a heat map view of the service which lets the analyst visually explore how an issue affects which components and which levels in the hierarchy. We provide a visual representation of the service that shows the components as colored tiles, with green indicating "normal" and red indicating "unusual". The scores from the hierarchical detector determine the color of the tiles. See figure 1 for an example.

B. Query-based Diagnosis System

The hierarchical detection system provides the analyst with an initial clue for understanding an issue affecting the service. For more detailed investigation, our diagnosis system supports a SQL-like query language that the analyst can use to do more detailed analysis. In addition to supporting primitives such as SELECT, GROUP BY, etc., this language also provides additional *machine learning primitives* that apply machine learning algorithms on the data. To support interactive querying with acceptable time lag in displaying the results (no more than a few seconds), the diagnosis system continually precomputes the results for a few popular time taking queries on latest counter time series data. The results are stored in a database. When a user issues a query via the user interface, the results are either computed or fetched from the database and displayed on the interface. See figure 2 for an example of a query for finding past counter time series time intervals that are similar to the query interval. Due to lack of space, we do not explain the details of the query language here. Based on the requirements of our users, we have implemented three types of diagnosis functionality: 1) finding unusual counters by comparing a time window of values of a counter against its own past values, 2) finding unusual counters by comparing the time window of

```
SELECT SIMILAR
    FOR
    QUERY_SPEC
    apenvironment=="Cosmos08-Prod-Co3c" AND machinefunction=="C13"
    QUERY_INTERVAL
    time >= DateTime.Parse("10/08/2014 09:00:00 PM") AND
    time <= DateTime.Parse("10/08/2014 10:50:00 PM")
    ACROSS
    apenvironment,machinefunction
```

Fig. 2. Example of a query for finding past counter time series time intervals that are similar to the query interval.

a counter across multiple instances of a service component, and 3) finding time intervals that are similar to a query time interval by comparing the counter time series across intervals. We briefly explain the algorithms used. One key constraint in the design of these algorithms is that they must be fast enough to analyze high dimensional time series data so that the results can be made available to the user as quickly as possible. Currently all three algorithms described below finish analyzing 12000 counter time series within about 8 minutes of receiving the data on two machines, each with two 2.2GHz, 16-core CPUs and 128GB RAM.

1) Unusualness with respect to past: Here the goal is to measure how unusual a counter's current values are compared to its past values. We maintain an estimate $P_h(x_c)$ of the probability distribution of the c^{th} counter x_c estimated from a history window h of some fixed size (3 days in this case). Given a new value for the counter, we compute its unusualness score as the negative log probability $s = -log(P_h(x_c))$. To get a score with smoother temporal behavior, we use the average negative log probability score for a window of k hours (k is set to either 3 or 6 by the user). We use a Gaussian kernel density estimate to represent $P_h(x_c)$. The kernel bandwidth parameter is estimated automatically using Least Squares Cross Validation [9]. The average negative log probability score is normalized by converting it to its corresponding percentile over a six month interval and scaling it by a "confidence" factor that accounts for closeness to the scores corresponding to neighboring percentiles. (We omit the details due to lack of space.) The normalized scores are used to rank the counters.

2) Unusualness across component instances: It is common for services to have multiple instances of a component that are performing the same functional role (e.g., a component handling data storage may have many instances to provide more capacity). Due to load balancing, many counters (related to the workload) will usually have only small variance across instances. One common task an analyst performs repeatedly is to compare a counter across instances of the same component to identify outliers. Performing this comparison manually for a large set of counters is impractical. Here we present an algorithm for automating it.

We pose the problem as follows: let x_c^i and x_c^j be the value for the c^{th} counter of the i^{th} and j^{th} instances, respectively. We want to estimate the probability that the value for the c^{th} counter of the i^{th} instance denoted by x_c^i is greater than that of the same counter for the j^{th} instance, denoted by x_c^j, by some margin m. This probability is denoted as the score $s_{ij} = P(x_c^i - x_c^j > m)$. The score for the i^{th} instance is then defined as $s_i = \sum_j s_{ij}$. This score will be high if the

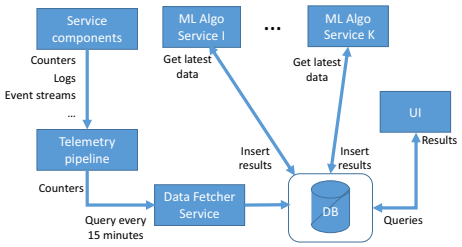

Fig. 3. Architecture of the integrated detection and diagnosis system. It performs three main steps: 1) collect performance counters in near real-time, 2) apply machine learning algorithms for detection and diagnosis, and 3) display the results on the user interface.

i^{th} instance has differences greater than the margin over many other instances, indicating that it is unusual. For each counter, we can rank instances using s_i. Again, we can represent the distribution $P(x_c^i - x_c^j > m)$ efficiently using a histogram for the differences $x_c^i - x_c^j$ for each pair (i, j). The histogram is computed from a fixed size time window, e.g., 12 hours to gain robustness against temporary large differences between instances. The margin m is set to be $k * \sigma$ where σ is the standard deviation of the c^{th} counter across instances within the time window. k is a parameter set by the user (e.g., 3).

3) Time series similarity search: Here the goal is to take a user-specified time interval as input and find other time intervals in the data that are similar to it. This is useful for identifying past instances of similar service issues, which can inform the user about the corrective actions taken in those cases. The unusualness scores for counters computed by the algorithm in subsection II-B1 are used to find past intervals with similar scores. We use the vector of scores for the counters for a time interval as its representation and compute a distance between intervals using an appropriate metric. Jaccard distance between the set of top-K unusual counters between two time intervals is used to measure dissimilarity between two intervals. K is set to 10. This allows finding time intervals with similar sets of counters behaving unusually.

Evaluation: We evaluated the algorithms in sections II-B1 and II-B2 on a set of 21 past issues which affected the service being monitored over a one year interval. For 20 out of 21 issues, the algorithms ranked the relevant counters as the highest ranked result, giving a Precision@1 of 0.9524. A more detailed evaluation is in progress.

III. System Architecture

Figure 3 shows the overall architecture of the end-to-end system. We make use of the telemetry data pipeline of the service being monitored to collect performance counters in near real-time and perform temporal aggregation (e.g., average a counter's values over one minute intervals). We have implemented a Data Fetcher Service which gets the latest aggregated counter values from the pipeline once every 15 minutes. These values are persisted in a database.

Once new data arrives at the database, the services which run the detection and diagnosis algorithms are called. The results on the latest data given by the algorithms are pushed into the database. Both the database and the algorithm services run on two machines, each with two 2.2GHz, 16-core CPUs and

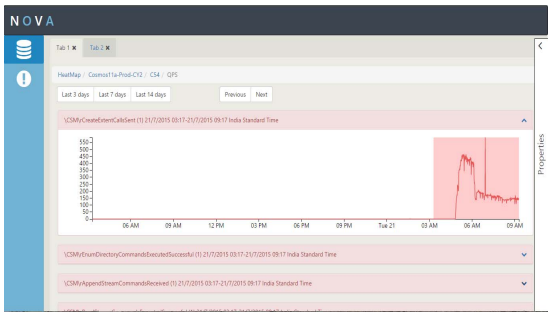

Fig. 4. Example of ranking counters by unusualness with respect to past. The counter time series shown above ranks the highest among the counters for a particular sub-sub-component of the service when the latest six hour window (shaded in red) is compared against the preceding three days (unshaded interval to the left of the red interval) using the algorithm in section II-B1. Figure needs to be magnified on screen for clarity.

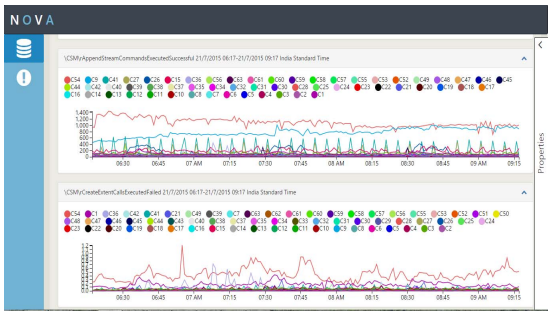

Fig. 5. Example of ranking counters by unusualness across component instances. The figure shows two counters for 64 component instances for which the user-specified instance is the most unusual during the shown time interval as determined by the algorithm in section II-B2. Figure needs to be magnified on screen for clarity.

128GB RAM. The system currently analyzes 12000 counters collected at one minute intervals.

We have implemented a browser-based user interface. The interface fetches data and algorithmic results from the database based on user requests and displays them. Multiple simultaneous user sessions are supported.

IV. Demonstration Overview

The demonstration consists of two parts: 1) a walk-through of recent issues which affected the service monitored using our system, and 2) an interactive session where audience members can try out their own queries on the time series data we have.

Walk-through of issues: We have a collection of recent issues which have affected the service that is monitored using our system. In this part of the demo, we describe how our system is used in detecting and diagnosing those issues. The starting point of the demo is an alert message generated by the detection system indicating that an issue is affecting the service. Next, the service components, sub-components, and counters involved in the issue are shown using the hierarchical view generated by our system. We explain how this hierarchical view helps the user localize the issue to a small part of the service.

Once an initial set of relevant counters is identified using the hierarchical view, we show how the diagnosis algorithms described in section II-B are used to identify further relevant counters and past time intervals for understanding the issue at hand. We show use cases for 1) computing unusualness with respect to past values of counters, 2) computing unusualness across component instances, and 3) finding similar time intervals from the past. For example, figure 4 shows a screenshot of the top ranking counters selected by the algorithm in section II-B1 during the time interval of a service issue. The top ranking counters in this case do turn out to be relevant for understanding the incident. Figure 5 shows a screenshot of the top ranking component instances selected by the algorithm in section II-B2. This is an example where the component instances relevant for a particular service issue is correctly identified by the algorithm.

Interactive query-based exploration of time series data: In this part of the demo, we ask the audience members to try out their own queries to better understand the diagnosis functionality provided by our system and the interactivity it supports. We provide some example queries that the audience members can modify to form new ones which can be run on the time series data we have. The idea is for them to run various queries, visually evaluate the quality of the results returned, and thus get an intuitive understanding of the system's functionality. This does not require a detailed understanding of our time series data or the service from which the data is collected. From our experience, the diagnosis algorithms tend to produce intuitive results which can be understood in terms of visual unusualness or visual similarity of the time series patterns.

References

[1] P. Bodík, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *EuroSys 2010*, pages 111–124, 2010.

[2] L. Cao, Q. Wang, and E. A. Rundensteiner. Interactive outlier exploration in big data streams. *VLDB*, 7(13):1621–1624, 2014.

[3] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3), 2009.

[4] Q. Fu, J. Lou, Q. Lin, R. Ding, D. Zhang, Z. Ye, and T. Xie. Performance issue diagnosis for online service systems. In *SRDS*, pages 273–278, 2012.

[5] D. Georgiadis, M. Kontaki, A. Gounaris, A. N. Papadopoulos, K. Tsichlas, and Y. Manolopoulos. Continuous outlier detection in data streams: an extensible framework and state-of-the-art algorithms. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1061–1064. ACM, 2013.

[6] D. Lin, R. Raghu, V. Ramamurthy, J. Yu, R. Radhakrishnan, and J. Fernandez. Unveiling clusters of events for alert and incident management in large-scale enterprise it. In *KDD*.

[7] J. Lou, Q. Lin, R. Ding, Q. Fu, D. Zhang, and T. Xie. Software analytics for incident management of online services: An experience report. In *IEEE/ACM ASE*, 2013.

[8] V. Nair, A. Raul, S. Khanduja, V. Bahirwani, S. Sellamanickam, K. Selvaraj, S. Herbert, and S. Dhulipalla. Learning a hierarchical monitoring system for detecting and diagnosing service issues. In *Knowledge Discovery and Data Mining (KDD)*. ACM, 2015.

[9] B. Park and B. Turlach. Practical performance of several data driven bandwidth selectors. Technical report, Universit Catholique de Louvain, Center for Operations Research and Econometrics (CORE), 1992.

[10] S. Roy, A. C. Konig, I. Dvorkin, and M. Kumar. Perfaugur: Robust diagnostics for performance anomalies in cloud services. In *ICDE, 2015*, pages 111–124, 2015.