

Supporting System-Wide Similarity Queries for Networked System Management

Songyun Duan[†]
IBM T.J.Watson Research Center
Hawthorne, NY 10532
Email: sduan@us.ibm.com

Hui Zhang[‡] Guofei Jiang
NEC Labs America
4 Independence Way, Princeton, NJ 08540
Email: {huizhang,gfj}@nec-labs.com

Xiaoqiao Meng
IBM T.J.Watson Research Center
Hawthorne, NY 10532
Email: xmeng@us.ibm.com

Abstract—Today’s networked systems are extensively instrumented for collecting a wealth of monitoring data. In this paper, we propose a framework called System-wide Similarity Query (S^2Q) to support a new type of similarity queries on monitoring data for managing complex networked systems. The similarity queries are defined on a novel data model that captures system states, and the implementation includes a streaming algorithm for online state-modeling computation and a companion graph-based indexing technique for fast retrieval of historical system states. S^2Q simplifies many systems management tasks through a simple and intuitive query interface available to operators, and two applications are evaluated in the paper: (i) fast diagnosis of repeated failures in enterprise IT systems, and (ii) automated application traffic profiling on computer networks. For the first application, the diagnosis accuracy can reach 95% on a multi-tier web service testbed. For the second application, major network applications were automatically identified in the traffic logs from a large campus wireless network.

Index Terms—Management, Communication system, Data management, Data models, Statistical Machine learning

I. INTRODUCTION

In recent years, we have witnessed the proliferation of large-scale computer systems. With the success of Internet technology, such systems often contain a large number of computers and are capable of processing massive service requests simultaneously. For example, Google data centers employ hundreds of thousands of servers to process hundreds of millions of search queries in each day. Each of these systems has become extremely complex distributed systems with inter-dependent software and hardware components, including operating systems, application softwares, storage, networking devices and etc.

For the purpose of managing such complex systems, a common practice is to monitor the system status via extensive instrumentation. The obtained monitoring data are usually across various components and span a long time period. Due to the system size and complexity, the data are often massive and heterogeneous, which raises a grand challenge of how to correlate such data to the system status so that the system operators can have a holistic view on the healthness and performance of the entire system. Relevant to this challenge, there exist a few exploratory data analysis based system management solutions that support data navigation/visualization or SQL-like

queries. Representative work include AT&T *SWIFT-3D* data visualization and exploration system [18], UC Berkley TelegraphCQ system which supports continuous SQL queries [5], Yahoo Pig project on web-scale log processing [19], Facebook Hive system which supports a SQL-based query language in data warehouses [10], Microsoft DryadLINQ which supports large-scale data parallel applications with a SQL-like query language [25].

The solutions mentioned above allow the operators to conveniently query the raw data by monitored objects and/or timestamps. Nevertheless, there exists another missing yet important piece of information that can be derived from the raw data, that is, the (dis)similarity between the behavior of multiple monitored objects in the time domain. Queries regarding such (dis)similarity are common and can be quite useful for the operators. Example queries include:

- In performance management, a query could be “given the performance problem in time period T, whether and when the system ever experienced a similar problem in the past and had been reported a successful problem diagnosis result?”
- In network traffic management, automated traffic monitoring tools benefit from the answer of “what are the top- k protocols that exhibit the most similar traffic patterns at an hourly time scale?”
- In network security management, anomaly detectors could ask “what set of traffic streams, identified by $\langle ip, port, protocol \rangle$, suddenly change the inter-similarity among their traffic patterns?”
- In workload management for a virtualized data center, Virtual Machine (VM) placement decision makers could benefit from queries such as “among three VM instances, which two have the most similar workload and which two have the most different workload?”

The (dis)similarity described above is clearly a general concept and applicable to many systems management tasks. Compared with examining the raw data at individual objects, knowing the (dis)similarity between objects gives more hints to the operators for understanding the entire system status. In this work, we propose a methodology to support queries specifically for such (dis)similarity. We name it *system-wide similarity query* (S^2Q). The main results of this paper are

[†]Work done while at NEC Labs America.

[‡]Correspondence author.

summarized as follows:

- 1) A framework for S^2Q is described. The framework is general for various target systems and systems management tasks.
- 2) A novel system modeling technique called *covariance matrix structures* is proposed to characterize dependency between multiple time-series. Such a dependency metric can be considered as one type of similarity. In accordance, a streaming algorithm is developed for an online computation of the dependency scores. A graph-based indexing algorithm is also developed for fast queries on the dependency graphs produced by the streaming algorithm.
- 3) Two systems management applications are evaluated by applying S^2Q : fast diagnosis of repeated failures in enterprise IT systems, and network traffic application profiling in computer networks.

The rest of this paper is organized as following: Section II presents a framework for supporting system-wide similarity queries. Section III describes the candidate data models and our choice for system modeling, and Section IV presents the streaming algorithms for online state modeling computation and their preliminary performance evaluation results. Section V discuss the similarity metric and various indexing choices. Section VI presents the evaluation results of two S^2Q applications; Section VII describes the related work and Section VIII concludes the paper with future work.

II. SYSTEM-WIDE SIMILARITY QUERY

A. Problem Statement

When a networked system is running, we can continuously collect monitoring data (about system metrics) from it, represented by data streams $D = \langle X^1, X^2, \dots, X^n \rangle$, where $X^i = (X_{t_0}^i, X_{t_1}^i, \dots)$ is a series of values of a system metric measured at some location i at time points t_0, t_1 and so on. For example, in a distributed multi-tier web service system, we may monitor its web servers, database servers, and application servers; on each server, we can collect system metrics (i.e., variables) such as network traffic volume, cpu and memory utilization. The series of values about each system metric is a data stream. We want to design a data management system supporting system-wide similarity queries, which ask about the similarity of one or multiple system objects on their time-based states. An object can be a physical item such as a blade server, a traffic flow between two IP addresses, or a logic item such as a 3-tier web service or an enterprise network. The states of an object characterize the property of the object that may change over time, which are derived from monitoring data D . The similarity of two system objects is calculated through a certain distance metric defined on their state space.

B. Framework

We propose a framework for supporting system-wide similarity queries, as shown in Figure 1. It is composed of the following steps:

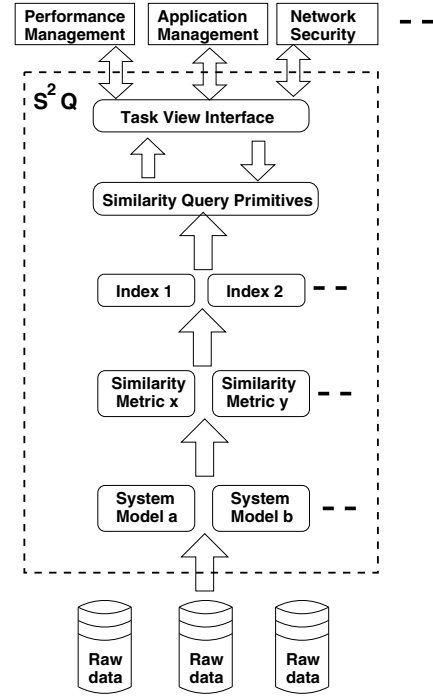


Fig. 1. S^2Q framework

- 1) System modeling. The first step in S^2Q is to transform the raw data from the monitoring process into useful information characterizing the state of the target system. Statistical analysis and data mining technologies like regression analysis have been applied in the modeling process as they can uncover rich relationships (e.g., probabilistic correlation, causal relationships, and statistical independence) from the measurement data, which are further utilized for management tasks. Clearly, different applications could require different data models, and a challenge in systems management is to design a data model framework to cover a large portfolio of diverse management tasks. We will present the detailed design of this step in Section III.
- 2) Similarity calculation. This step includes two parts: the definition of similarity metric and the computation approach. The similarity between two states needs to be quantified and defined through a distance metric on the state space, and the computation approach has to take the streaming property of the data sources into design consideration. We will present the detailed design of this step in Section IV.
- 3) Indexing: Monitoring data accumulate and are archived over time. To support efficient queries over the archived data, indexes need to be built for fast query processing. We will discuss this step in Section V-B.
- 4) Query plan formulation and execution. This step involves the translation of application queries into similarity query primitives, and the decision of query processing flows. A task view interface offers such functionality. We will present the preliminary results on this step in Section VI and discuss the future work in Section VIII.

Due to space constraint, we focus on system modeling and similarity calculation in this paper; indexing and query plan formulation are briefly discussed and left as future work.

C. An Example Management Query

System failures are very common in complex and large-scale networked systems, so diagnosis is a critical systems management task to ensure high system availability. We illustrate here how to use similarity query primitives to address diagnosis tasks. Suppose we have appropriate system modeling and indexing techniques to extract and index historical time-dependent system states, denoted as $S_H \cup S_U$, where S_H represents *healthy states* when the system performs well, and S_U represents *failure states* when the system does not meet performance requirements. When there is a failure state S_q to diagnose, which is captured during or right before a failure, the administrator can pose a diagnosis query in the form $Q = \text{most_similar}(S_q, N; S_U)$, which asks for the N failure states in S_U that are most similar to S_q . The administrator can refer to the past diagnosis results of the retrieved N failure states to find out the root cause of S_q . The administrator can also pose a diagnosis query in the form $Q = \text{least_similar}(S_q, N; S_H)$, which asks for the N healthy states in S_H that are least similar to S_q . The administrator can compare S_q with the retrieved N healthy states and use the difference between them to pinpoint the root cause of S_q .

Next, we present the details of the S^2Q design, starting with system modeling.

III. SYSTEM MODELING WITH TIME-SERIES DATA

In this section, we describe the design space of system state modeling and propose a new data model for systems management.

A. Raw Data

We can use observations of system metrics with the same time stamp t (i.e., $\langle X_t^1, X_t^2, \dots, X_t^n \rangle$) to represent the system state at time t . The drawback of this representation is it is sensitive to noise and only gets superficial property of system states.

B. Clustering-Based Techniques

The observations of system metrics at time t (i.e., $\langle X_t^1, X_t^2, \dots, X_t^n \rangle$) can be treated as a data point in high-dimensional space, as n is usually very large. Data points with different time stamps can be partitioned using a classical clustering technique *K-Means* [22], and cluster centroids can be used to represent system states. One advantage of this representation is it is relatively robust to observation noise. However, it is hard to determine the number of clusters, i.e., the number of distinct system states. In addition, recent work observes that *K-Means* suffers from the *curse of dimensionality* in high dimensional spaces [8]. In such spaces, for any pair of data points within the same cluster, it is highly likely that there are some dimensions on which the data points are distant from

each other. Thus, distance functions that use all dimensions equally can be ineffective. Furthermore, different clusters may exist in different subspaces, i.e., comprised of different subsets of system metrics [8].

C. Pairwise-Dependency Relationships

We can capture pairwise-dependency relationships of system metrics to characterize system states. The intuition is dependency relationships of system metrics change significantly only when systems transition from one state to another. As shown in [13], pairwise-dependency relationships can be made robust to errors and superficial changes in measurement readings. The key issue is how to capture dependencies between system metrics. There are two types of dependencies: (I) physical dependencies, which describe direct invocation relationship between a pair of components and can be constructed from domain knowledge and monitoring data, like described in [1], [15], [16]; and (II) statistical dependencies, which calculates statistical correlation of time-series from a pair of system metrics using some correlation metric. A simple metric of Type II is linear correlation [22], however, it is sensitive to observation errors, as shown in a preliminary evaluation later. We propose a new metric of Type II based on *covariance matrix* structure of a pair of time-series collected from two individual system metrics around a given time point. Let X and Y represent two such time-series. The dependency score (i.e., correlation score) of X and Y is computed in the following way:

- 1) Generate the auto-covariance matrix of X and Y around time t respectively. Specifically, for time-series X , we define a sliding window of size w starting from time t as $[X_t, X_{t+1}, \dots, X_{t+w-1}]$, denoted as $X_{t,w}$. Extract the past m sliding windows with timestamps $[t-m+1, t-m+w], \dots, [t, t+w-1]$. Denote the auto-covariance matrix of X as $ACov_X = \frac{1}{m} \sum_{i=t-m+1}^t X_{i,w} * X'_{i,w}$, where $X'_{i,w}$ is the transpose of $X_{i,w}$. The auto-covariance matrix of Y , denoted as $ACov_Y$, is defined in a similar way.
- 2) Compute the dependency score of X and Y based on their auto-covariance matrices. The first step is to decompose the covariance matrices using singular value decomposition (SVD). It is easy to show that the eigenvectors of $ACov_X$ are the same as the principle components in X , but the computation of the former is less expensive. The dependency score of X and Y is computed as the distance between two subspaces expanded by the top- k principle components of X and Y respectively. The subspace distance will be defined in Section IV-C.

In the above system modeling process, Step 1 is done on each time-series to identify their internal evolving patterns, which we call *local synopses*; Step 2 is done between two time-series to identify their behavior correlations, which we call *global synopses*. With *local synopses* and *global synopses*, our system model allows tracking both local status of each monitoring point and chain them together for a global view.

IV. PAIRWISE-DEPENDENCY RELATIONSHIP COMPUTATION

Given the streaming values of time series X and Y on two overlapping windows with timestamps $[t - m, t + w]$ and $[t - m + \delta_w, t - m + \delta_w + w]$ respectively, we want to update the dependency score of X and Y at time $t + \delta_w$, denoted as $d_{t+\delta_w}$, from the previous score d_t in an incremental way, instead of computing $d_{t+\delta_w}$ from scratch. The key step of a streaming algorithm for the incremental update of dependency scores is to reflect the change brought by new data with timestamps $[t + w + 1, t - m + \delta_w + w]$ and to remove the effects of old data with timestamps $[t - m, t - m + \delta_w - 1]$.

A. Auto-Covariance Matrix (Equal-Importance)

For time-series X , we can update its covariance matrix M incrementally as follows:

$$M_{t+\delta_w} = M_t - \sum_{i=t-m}^{t-m+\delta_w-1} (X_{i,w} * X'_{i,w}) + \sum_{i=t+w+1}^{t-m+\delta_w+w} (X_{i,w} * X'_{i,w}) \quad (1)$$

In the above formula, we assume that the past m sliding windows have the same effect on the computation of the current auto-covariance matrix, i.e., the weight of each sliding window is 1.

What we need is to update the top- k eigen-vectors of the auto-covariance matrix incrementally. Let us define *lagged-matrix* as $D_t = [X'_{t-m+1,w}; \dots; X'_{t,w}]$. The following formulas can be used to update the top- k left and right eigenvectors of D_t . $Q_B(t)$ ($Q_A(t)$) tracks the right (left) eigen-vectors at time t and $R_B(t)$ ($R_A(t)$) tracks the corresponding singular values. $I(k)$ is an identity matrix of size k by k . $\vec{0}$ is a matrix of size k by $w - k$, composed of 0s. Due to space constraint, we do not include theoretical correctness proof in this paper. The correctness is verified with experiments in Section IV-D.

$$\begin{aligned} Q_A(0) &= [I(k); \vec{0}] \\ B(t) &= X_{t,w} * Q_A(t-1) \\ B(t) &= Q_B(t) * R_B(t) \\ A(t) &= X'_{t,w} * Q_B(t) \\ A(t) &= Q_A(t) * R_A(t) \end{aligned}$$

B. Auto-Covariance Matrix (Decaying-Importance)

Another choice to determine the effect of past sliding windows on the computation of the current auto-covariance matrix can be based on their lifetime. Specifically, when we compute the auto-covariance matrix at time t , the sliding window at time $t - \delta_t$ is assigned an impact factor β^{δ_t} , so the impact of a sliding window decays over time. Here β is a decaying factor. The auto-covariance matrix of time-series X at time t is $M_t = \sum_{i=1}^t \beta^{t-i} X_{i,w} * X'_{i,w}$, which can also be updated incrementally:

$$M_t = \beta * M_{t-1} + X_{t,w} * X'_{t,w} \quad (2)$$

Denote the subspace expanded by the top- k eigenvectors of M_t as $V(t)$ and the covariance of $V(t)$ as $C(t)$. $V(t)$ can be

updated incrementally. Please refer to [24] for details on the correctness proof.

$$\begin{aligned} f &= V(t-1)' * X_{t,w} \\ g &= C(t-1) * f \\ h &= \frac{g}{\beta + f' * g} \\ \epsilon &= X_{t,w} - V(t-1) * f \\ V(t) &= V(t-1) + \epsilon * h' \\ C(t) &= \frac{C(t-1) - h * g'}{\beta} \end{aligned}$$

C. Distance between Subspaces

Suppose V_1 and V_2 (column vectors) are the basis of two subspaces respectively. Do SVD on the product of V_1 and V_2 : $[U, S, V] = svd(V_1' * V_2)$. The singular values in S are the cosine values of principal angles between the two subspaces [2]. If the maximum principal angle is small (i.e., the maximum singular value is close to 1), the two subspaces are close to each other.

D. Preliminary Evaluation

We use two synthetic time-series to verify the correctness of our streaming algorithms. In the MATLAB simulation, a basic time-series X^1 is generated from $\sin(t)$, where $t = 1 : 1000$ are timestamps.

1) *Comparing Robustness to Observation Noise*: A second time-series is generated by adding noise to the basic time-series: $X^2 = X^1 + U(0, 1)$, where $U(0, 1)$ is white noise with values in the range of $[0, 1]$.

Figures 2(b) and 2(c) show that dependency scores based on auto-covariance matrix are more robust to observation noise than that based on linear correlation between X^1 and X^2 . In the existence of observation noise, the former still outputs dependency scores around 0.9, while the latter degrades to values around 0.6. The closer the dependency score to 1, the better the robustness to noise.

2) *Comparing Robustness to Timely Delay*: A second time-series is generated from $\sin(t + 1)$, in which there is one time unit delay in observations with regard to the original time-series $\sin(t)$. Figure 3(b) shows that dependency scores based on auto-covariance matrix are more robust to timely delay than that based on linear correlation. In the existence of timely delay, dependency scores of the former are not affected by such a delay, while dependency scores based on linear correlation are significantly impacted, as shown by the drop in dependency score to 0.54. The closer the dependency score to 1, the better the robustness to timely delay.

3) *Approximation Accuracy of Streaming Algorithms*: From Figure 2(d) and Figure 3(c), we can see that the streaming-version of eigen-space tracking algorithms can approximate the batch version algorithms when time-series values change gradually over time. But when there is no correlation between values with two consecutive time stamps (i.e., the values are randomly generated), the approximation accuracy degrades. We skip the details due to space constraint.

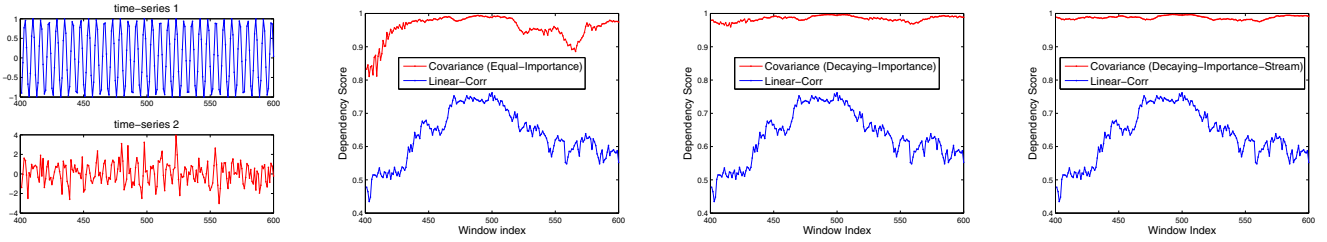


Fig. 2. Robustness to noise: (a) time-series, (b) equal-importance, (c) decaying-importance, (d) streaming version

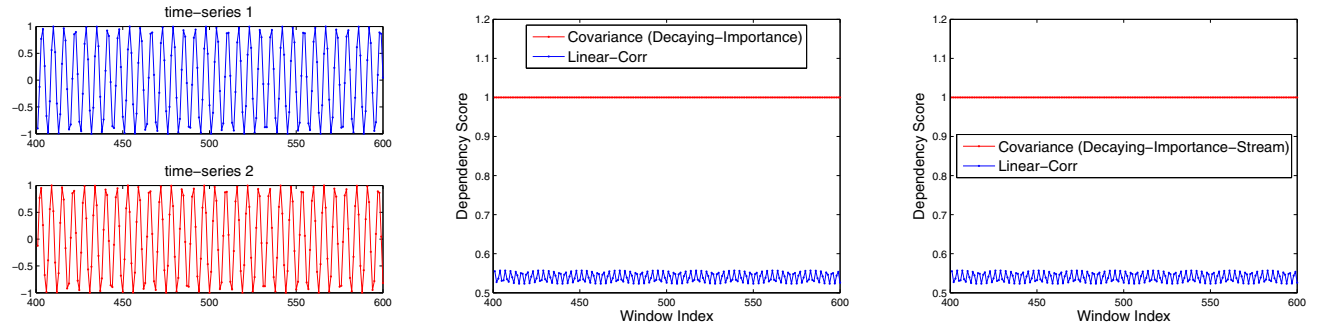


Fig. 3. Robustness to timely delay: (a) time-series, (b) decaying-importance, (c) streaming version

4) *Comparing Running Times*: Figures 4(a) and 4(b) compare the running time of different techniques for computing dependency scores when the timestamp starts from 200 and 400 respectively. It is clear that the streaming algorithm is the most efficient one; its running time is almost constant. The batch-version algorithm with decaying-importance factor increases almost linearly as historical sliding windows accumulate over time.

V. SIMILARITY METRICS AND INDEXING

A. Similarity Metrics

Given the representation of system states (based on raw data or transformed data such as pairwise dependency), our task is to retrieve the past states that are similar to the current state s . There are several options for such a retrieval.

1) *Instance-Based Retrieval*: A simple technique is to find the nearest neighbor of s among the past states. The distance metric can be Euclidean distance or Hamilton distance [22]. For multi-dimensional subspaces, the distance described in Section IV-C can be applied.

2) *Clustering*: Instance-based retrieval is highly impacted by noise in the data. To find robust neighbors, it may be good to cluster the past system states first, and retrieve the cluster centroid that is closest to s [7]. Clustering techniques such as *K-means* and locally-adaptive clustering (LAC) algorithms [8] can be applied here.

3) *Classification*: Classification can be used to group states that have a common pattern, assuming there are *annotations* associated with past system states. The annotation of a failure state could be the root cause of the failure. To learn the classifier, annotations of past system states are treated as

the class label, and the other system metrics are treated as predictor attributes. For instance, [26] identifies known failure types using a multi-class classifier, specifically, a Support Vector Machine (SVM) [22]. [6] learns a *decision tree* (i.e., CART) from the historic data $H \cup U$, where H is a set of data points collected when systems are in healthy states, and U is a set of data points collected when systems are in failure states. The tree's leaf nodes represent a partitioning of the raw data. A nice feature of this approach, which arises from its roots in classification as opposed to clustering, is that the tree minimizes the chances of putting data points from healthy states and data points from failure states into the same partition.

4) *Graph-Based Approach*: With the data model proposed in Section III-C, a dependency graph $G_t = (V, E_t)$ will be generated at time t using the covariance-matrix-based dependency metric, where V is the set of target system objects, and E_t is the set of dependency relationships between system objects at t . The distance between two graphs G_{t1} and G_{t2} can be measured as the sum of edge weight difference if dependency scores are used as the edge weights. In practice, we use a noise-robust approach which firstly prunes away the edges whose weights are below a threshold (e.g., 0.9), and then calculates the distance between G_{t1} and G_{t2} as:

$$D(G_{t1}, G_{t2}) = 1 - \frac{|E_{t1} \cap E_{t2}|}{|E_{t1} \cup E_{t2}|}$$

where $|E_{t1} \cap E_{t2}|$ is the number of common edges between the graphs of G_{t1} and G_{t2} , and $|E_{t1} \cup E_{t2}|$ is the number of unique edges in the two graphs.

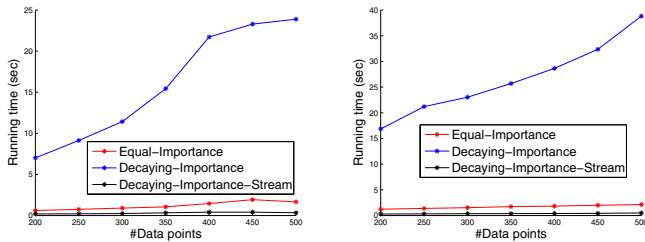


Fig. 4. Running time: (a) starting timestamp = 200; (b) starting timestamp = 400

B. Change-Detection-Based Indexing

Given one of the above system state representations (such as raw data and pairwise-dependency graph), we can compute the distance between system states at two consecutive time points. In this way we generate a one-dimensional plot of neighborhood distances. When system state does not change, the neighborhood-distances should be close to 0, while there should be a significant change in values when there is a state transition. An index can be built through the detection of change points in the neighborhood-distance plot. The index consists of representative states between each pair of consecutive change points; each representative state is associated with a beginning time stamp and an ending time stamp, between which there is no state change. The retrieval of a given state s among the past system states can be done by comparing s with the representative states in the index, thus requiring much less processing time, as the number of representative system states is usually much less than the number of all past states in enterprise IT systems.

VI. EXPERIMENTAL EVALUATION

A. Fast Diagnosis of Repeated Failures in IT Systems

1) *Similarity Query Formulation*: [3] reports that typically 50%, sometimes as much as 90%, of all software problems reported by users today are recurrences of known problems whose cause has already been confirmed or is under investigation. If a tool can retrieve historic system states from the monitoring data that are similar to one or more given failure states, it helps system administrators leverage past diagnosis efforts for recurrent failures. Recall from Section II-C that an administrator or system-management software can diagnose the cause of a failure represented by S_q via a diagnosis query in the form $Q = \text{most_similar}(S_q, N; S_U)$, which asks about the top- N failure states in S_U that are most similar to the failure state to diagnose.

2) *Experimental Setting: Failures injected in a testbed*: We have implemented a testbed that runs *Rubis* [21]—a multi-tier auction service modeled after eBay—on a JBoss application server (with an embedded Web server) and a MySQL DBMS. It has been reported that software problems and operator errors are the common causes of failure in Web services [20]. We inject such failures into a running Rubis instance using a comprehensive failure-injection tool [4]. This

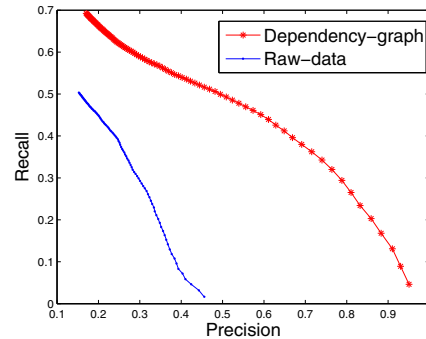


Fig. 5. Precision-recall: pairwise-dependency (decaying important - stream) vs raw data

setting makes it easy to study the accuracy of the diagnosis algorithms because we always know the true cause of each failure.

We collect data for each type of failure ft in the following way: (1) run the testbed under a stable workload (there is no performance bottleneck for this workload) for 20 minutes; (2) inject the specific failure ft into the testbed and let it run for 20 minutes; (3) micro-reboot [4] the EJBs where the failure is injected to fix the failure and let the testbed run for 20 minutes; and (4) go to Step 2. The whole process lasts for 200 minutes. While the system is running, we record the number of times that each of the 103 distinct EJB procedure calls is invoked per minute. Monitoring at this level is light-weight. The violation of *Service Level Objectives* (SLO) is defined on the average response time per request. For evaluation purposes, we assign an annotation to each instance (i.e., data points) being collected as: (i) healthy, if the instance has no SLO violation, or (ii) identifier of the injected failure, if the instance has SLO violations.

3) *Retrieval for Diagnosis*: We adopt the traditional metric of (precision, recall) for evaluating the quality of retrieval results: for a given state s , precision measures the percentage of the N returned instances that have the same annotation as s (100% is perfect); and recall measures the percentage of historic instances with the same annotation as s that are actually returned (100% is perfect). For a given number N , we compute the average precision value of all the test instances and the average recall value of all the test instances. The number N is varied from 1 to 50 to plot the precision-recall curves. Intuitively, as N increases, we expect recall improves, while precision drops. Thus, the bigger the area under the precision-recall curve, the better the quality of retrieval.

We compared the pairwise-dependency modeling approach with a few others including clustering and PCA-based multi-dimensional subspace approaches. Due to the space limit, we showed here the results of two:

- K-means clustering technique applied on the raw monitoring data, with Euclidean distance as the similarity metric. We call it *raw-data* approach in the rest of the paper.
- the pairwise-dependency state model chosen in Sec-

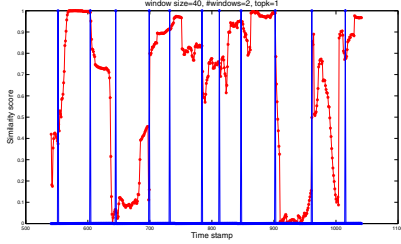


Fig. 6. Neighborhood distance plot: pairwise-dependency (decaying important - stream)

tion III-C. We call it *pairwise-dependency graph* approach.

For the pairwise-dependency-graph approach, the parameter settings are as follows: window size = 40; number of windows = 20; use the top-2 left eigenvectors to compute dependency; use the instances with index from 300 to 4000 for evaluation, among which 60% is used as training data and the rest as test data. Figure 5 shows the results for both approaches on the retrieval precision (X-axis) and recall (Y-axis) performance. For each approach, the rightmost point in its precision-recall curve corresponds to $N = 1$, and the left-most point corresponds to $N = 50$. Clearly, the retrieval performance based on pairwise dependency significantly outperforms that based on raw data. Actually, when $N = 1$ (the returned query result included the nearest historical state), our pairwise-dependency graph approach has the precision rate of 95%.

4) *Change Detection*: As discussed in Section V-B, we can generate the neighborhood-distance plots based on the pairwise-dependency state representations.

Figure 6 plots the neighborhood similarity (1 - distance) based on pairwise-dependency graph representation. The red line records the neighborhood distance, and the blue line shows the actual change points of system states. The parameter settings for pairwise dependency modeling are as follows: window size = 40, number of windows = 2, use the top-1 left eigenvectors to compute the distance between two states. The instances have index from 500 to 1000. The change points can be visually identified. For 500 states, the 11 change points divide them into 12 sets where in each set the states are similar and can be compressed with few representative states.

B. Automated Application Traffic Profiling in Enterprise Networks

1) *Similarity Query Formulation*: In this task, our approach is based on the observation that the traffic data from randomly used ports is similar to noise signals. Given network traffic data, we compute the similarity score at each pair of traffic objects <IP address, port number>, and group together the traffic objects with similar traffic patterns with the speculation that they likely belong to the same network applications, e.g., HTTP service, p2p application, instant messenger. The application query is formulated as: $Q = \text{within}(O_q, d; O)$. O_q is the state of the target traffic object, d is a similarity threshold, and O is the set of all traffic objects found in the monitoring

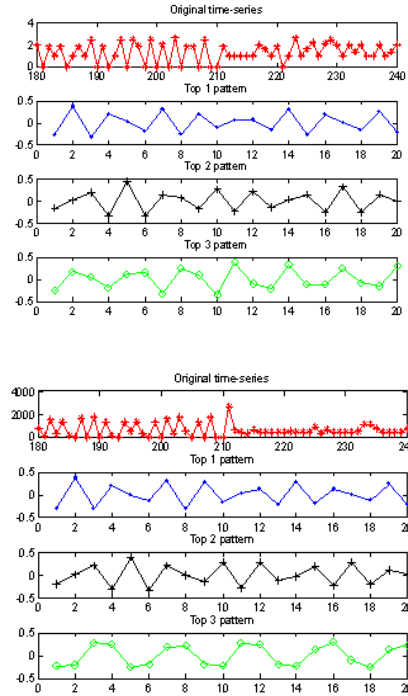


Fig. 7. Traffic data and patterns: (a) port 137, protocol = UDP, (b) port 139, protocol = TCP

data. The query Q asks for all traffic objects that have a similarity score within d to O_q . In the evaluation, we first compare O_q with itself at different times of daily windows. If the similarity satisfies the requirement, we conclude O_q is a part of some application traffic, and continue to compare it with other objects within the same time windows.

2) *Experimental Setting*: We analyze the tcpdump data from Dartmouth campus-wide wireless network traffic [17]; each packet is in the format of <SIP, DIP, SPort, DPort, protocol>, where SIP is the source IP address, DIP is the destination IP address, SPort is source port number, DPort is destination port, and protocol is the transport protocol which can be TCP or UDP. We aggregate the data based on <port, protocol> combinations with flow statistics in 5-minute interval. The statistics metrics include the number of packets, the number of bytes, and two entropy-related metrics. The entropy is calculated based on the distribution of IP addresses that appear with a specific port number. We do not assume any prior knowledge about application-port mappings. Our goal is to identify the main applications that account for the major part of the network traffic; each application is represented as a group of <port, protocol> combinations.

3) *Results*: Figure 7 plot the traffic data of port 137 and 139 respectively with their extracted patterns using our techniques described in Section IV. Although the original time-series data (at the top) look dissimilar, the patterns extracted from them are very similar. The underlying reason is both port 137 and port 139 are associated with the same application

of NetBIOS. The figure shows the power of our technique to identify applications automatically from network traffic data.

We also run experiments with one-day traffic data at a sniffing point, which contains around 8000 <port, protocol> combinations. By computing the similarity scores between these <port, protocol> combinations, we identified 15 applications, represented by the port number. The well-known applications include: 80 (HTTP), 53 (DNS), 137&139 (NetBIOS), 1214 (Kazaa), 5190 (AOL Messenger), 161 (SNMP), 0 (ICMP), 67&68 (DHCP), 1071 (BASQUARE-VOIP), and 6699 (WinMX). Out of these 15 applications, 6 were confirmed as the top applications in their data set ¹.

VII. RELATED WORK

Recent work [14][13][11][23][12] proposed applying various statistical and data mining techniques to uncover pairwise relationships in data collected from complex computer systems. These work differ in their defined pairwise relationships and their usage of the uncovered relationships. Kandula et al [14] extracted predicate rules from packet traces collected in edge networks. Their method is based on statistical rule mining techniques and leverages the packet timing information. The extracted rules are used to help network monitoring, fault diagnosis and intrusion detection. Jiang et al [13] used AutoRegressive models with eXogenous inputs (ARX) to model pairwise relationships and learned such relationships between timeseries measurement data. The learned ARX models are further used for fault detection and capacity planning. Hua et al [11] built system performance analysis tool based on linear regression model. Qu et al [23] improved the prediction and accuracy of their anomaly detection algorithm for network security by computing the pairwise correlations between a large amount of features subtracted from massive IP flow data. Huang et al [12] used the PCA method to diagnose network-wide disruptions. Compared to these work, Our method is based on a general and robust pair-wise dependency metric which covers both local and global views of the system states. Therefore, our method is flexible and can support diverse systems management tasks.

Several existing work [18][5][19][10][25] described in the introduction proposed various massive monitoring data exploration methods. While these work facilitates querying the raw data, S^2Q complements them by supporting querying pairwise (dis)similarity.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we describe S^2Q , a framework supporting a simple yet powerful data query primitive that system operators can utilize to discover pairwise relationships for the purpose of systems management. In S^2Q framework, we propose a novel system model which characterizes the temporal patterns of monitoring objects. We also propose an algorithm for computing similarity scores based on the patterns instead of

the raw values. S^2Q has been evaluated in two system management contexts: diagnosis of repeated failures in IT systems, and application traffic profiling in enterprise networks. The experimental results are promising.

One challenge for realizing S^2Q in a large system is the high computational overhead brought by massive amount of monitoring data. We plan to implement the S^2Q interface and query execution functionalities on top of Hadoop [9], a general platform supporting data-intensive distributed applications. Also, we plan to extend S^2Q to other systems management tasks such as security and workload management.

REFERENCES

- [1] P. Bahl et al. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proc. of SIGCOMM 2007*.
- [2] A. Björck and G. H. Golub. Numerical methods for computing angles between linear subspaces. Technical report, Stanford, CA, USA, 1971.
- [3] M. Brodie et al. Quickly finding known software problems via automated symptom matching. In *Proc. of ICAC 2005*.
- [4] G. Candea et al. Automatic failure-path inference: A generic introspection technique for internet applications. In *Proc. of IEEE Workshop on Internet Applications*, 2003.
- [5] S. Chandrasekaran et al. Telegraphcq: continuous dataflow processing. In *Proc. of SIGMOD 2003*.
- [6] M. Chen et al. Failure diagnosis using decision trees. In *Proc. of ICAC 2004*.
- [7] I. Cohen et al. Capturing, indexing, clustering, and retrieving system history. In *Proc. of SOSP 2005*.
- [8] C. Domeniconi et al. Locally adaptive metrics for clustering high dimensional data. *Data Mining and Knowledge Discovery*, 14(1), 2007.
- [9] Hadoop. <http://hadoop.apache.org/>.
- [10] F. Hive. <http://hadoop.apache.org/hive/>.
- [11] K. A. Hua et al. Admire: an algebraic approach to system performance analysis using data mining techniques. In *Proc. of SAC 2003*.
- [12] Y. Huang, N. Feamster, A. Lakhina, and J. Xu. Detecting network disruptions with network-wide analysis. In *Proc. of SIGMETRICS 2007*.
- [13] G. Jiang, H. Chen, and K. Yoshihira. Discovering likely invariants of distributed transaction systems for autonomic system management. *Cluster Computing*, 9(4):385–399, 2006.
- [14] S. Kandula, R. Chandra, and D. Katabi. What’s going on?: learning communication rules in edge networks. In *Proc. of SIGCOMM 2008*.
- [15] S. Kandula et al. Shrink: A tool for failure diagnosis in IP networks. In *Proc. of workshop on mining network data*, 2005.
- [16] R. Kompella, J. Yates, A. Greenberg, and A. Snoeren. Ip fault localization via risk modeling. In *Proc. of NSDI 2005*.
- [17] D. Kotz and K. Essien. Analysis of a campus-wide wireless network. In *Proc. of MobiCom 2002*.
- [18] E. E. Koutsofios, S. C. North, R. Truscott, and D. A. Keim. Visualizing large-scale telecommunication networks and services (case study). In *Proc. of the conference on Visualization*, 1999.
- [19] C. Olston et al. Pig latin: a not-so-foreign language for data processing. In *Proc. of SIGMOD 2008*.
- [20] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do internet services fail, and what can be done about it. In *USENIX Symposium on Internet Technologies and Systems*, 2003.
- [21] *Rice University Bidding System*. rubis.objectweb.org.
- [22] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, second edition, June 2005.
- [23] T. Xia, G. Qu, S. Hariri, and M. Yousif. Genetic algorithm and information theory: A hybrid approach on intrusion detection. in *Proceedings of 24th IEEE International Performance Computing and Communications Conference*, 2005.
- [24] B. Yang. Projection approximation subspace tracking. *Signal Processing, IEEE Transactions on*, 43(1):95–107, Jan 1995.
- [25] Y. Yu et al. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. of OSDI 2008*.
- [26] C. Yuan et al. Automated known problem diagnosis with event traces. In *Proc. of EuroSys 2006*.

¹The confirmation was based on a detailed data report obtained through the private communication with David Kotz, the data owner at Dartmouth.