# A Flexible Architecture for Statistical Learning and Data Mining from System Log Streams

Wei Xu Peter Bodík David Patterson
EECS Department, UC Berkeley
{xuw, bodikp, patterson}@cs.berkeley.edu

## Abstract

*Modern computer systems are instrumented to generate huge amounts of system log data. This data contains valuable information for managing the system, localizing failures, and recovery. However, the complexity of these systems greatly surpasses what can be understood by human operators and thus automated analysis systems are beginning to be used. Due to preprocessing required by the statistical algorithms, the extremely high volume of data cannot be processed using ad-hoc scripts. We present a flexible, modular and scalable architecture for statistical learning from large data streams that can easily process lots of data. We built a prototype that is evaluated using system log data from a commercial on-line service. Moreover, the results of the analysis were genuinely useful for the on-line service operators.*

## 1. Introduction

### Data analysis and collection in general

Most on-line services such as Amazon or eBay suffer from various user-visible failures. As reported by various authors (such as [14]), the most common causes of failure in computer systems are *software bugs*, *human operator errors* and *hardware failures*. These failures cause system downtime which is very expensive; for example, [15] estimates that 1 hour of downtime of a large on-line service can cost up to $1 million. Predicting, detecting or localizing these failures is a very difficult task – some systems consist of hundreds of software components running on up to 50,000 servers [13].

A standard approach of most companies is to instrument the system so that it reports various statistics such as performance of each machine, execution details for each request, or network statistics. Human operators monitoring the system use the log data and their experience to detect failures and identify possible root causes.

This approach has several limitations: computer systems are too complex and their behavior cannot be easily understood by humans. The systems today generate as much as 1 TB of log data every day [4]. Using more fine-grained instrumentation they could generate 100x more log data, but it is impossible to manually process such large data sets.

### Motivation

The recorded data contains more useful information than the operators can discern. The data might be used to predict that a particular machine is likely to crash in a few minutes or that a certain software component has a bug. This extra information can expedite detection and localization of failures. A fully automated analysis would be cheaper, more reliable and could potentially understand more complex behaviors. However, most of the companies today use very limited automated analysis of the log data they collect.

One of the reasons why it is difficult to build an automated analysis system is that it is hard to manage so much data in real time. A lot of preprocessing (such as sampling, adding/removing attributes, merging data from different sources, and so on) is required before the data reach the algorithms. Our experience shows that systems based on Python scripts are not flexible enough to analyze the scale of log data produced in on-line services. Using ad-hoc scripts for parallel preprocessing of data is tedious and does not allow easy modification of data streams and algorithms.

Instead, we need a better data model for the system log, and a scalable, modular architecture that can be distributed over a cluster of machines to process the data quickly. We need to be able to easily add new data streams of different types and data rates, create new features/attributes on-line, and try different types of algorithms. Since we need to test the system in production environments, it needs to be easily deployable for testing.

**Contribution**
In this paper we propose the use of stream-based processing and present a prototype of a system that allows us to preprocess data for any off-line or on-line statistical learning algorithm against massive quantities of system logs. The system can be easily distributed over a cluster of machines and new data streams and algorithms can be added on-the-fly. Although our architecture is designed for analyzing system log data, it can be used in other situations where data mining and statistical learning theory (SLT) algorithms need to be applied to huge amounts of data. To demonstrate its benefits, we implemented a decision tree algorithm for generating interesting rules from the log data. The algorithms and the architecture are evaluated on a data set from an on-line service with data rate of about 150GB per day.

The structure of the paper is as follows: in Section 2 we present a set of decision-tree-based algorithms for off-line detection of interesting rules/behavior in a typical computer system; Section 3 highlights practical problems we ran into when trying to implement them. Sections 4 and 5 describe the new system and our experience with it. The results of the algorithms in Section 6 are followed by the details of the design and implementation of the system in Section 7.

## 2. System log data analysis

In this section we describe a typical data set obtained from an on-line service that we use in our experiments. We also describe a few types of algorithms that would be useful for system operators and present some specific examples. These algorithms and problems with their implementation serve as motivation for our stream-based system.

### 2.1. Data set X

Our data set comes from a commercial on-line service (similar to AOL) running on 440 nodes. Since failure information is sensitive, the company sharing this data prefers to be anonymous (we refer to it as company X or system X). The data was recorded during a 20-day period and its size is about 2.5 TB.

The data set contains three types of data:

**Request data.** Every request executed in the system reports at least the following attributes: *time*, *machine*, *user id*, and *application*. In addition to that, a request reports a subset of 450 attributes including *request type*, *content length* or *queue duration*; the actual attributes that are reported depend on the application that is executed and the events that happen during the execution. Note that *requests* are not restricted to requests from *users*; requests from other parts of the system are also included. Due to privacy reasons, some of the attributes were *anonymized*; replaced by a hash of their string value. The peak rate of requests is about 11,000 requests per second. Each 1 minute of request data from all machines is stored in one compressed file of size $\approx$ 20 MB (uncompressed size is $\approx$ 80 MB).

**Performance data.** Every node reports its performance statistics every five minutes – 17 attributes such as: *memory utilization*, *swap utilization*, *load average*, *CPU idle time*, or *TCP segments received in error.* The size of the performance data for 20 days is $\approx$ 850 MB; the data rate is about 10,000 times slower than for the request data.

**Trouble ticket data.** The problems detected by operators were written to a log ($\approx$ 450 kB) that includes possible causes for the failures.

### 2.2. What is useful

Our experience with several companies suggests that the following types of automatic log data analysis would be useful in general:

1. **localizing the root cause of a failure**: Often the system operators know that *something* in the system failed, but localization of the root cause is a significant problem. As reported by [9], one large site estimates that about 93% of recovery time from application-level failures is spent in detecting and diagnosing them.
2. **predicting failures before they happen**: If we could predict failures of particular machines, we could redirect the traffic from the affected machine and avoid service disruptions.
3. **detecting (unexpected) patterns in the data**: During the *post-mortem* analysis of a failure it would be useful to have models of typical and unexpected system behavior. These models would allow the operators to better understand the system.

The automated analysis systems should not replace the operators but help them understand the system better. Human operators are experienced with the system, so we need to exploit it in our analysis. The analysis systems should thus also incorporate feedback from the operators.

### 2.3. Analyzing the data

It is still not known what algorithms work best for system log analysis. However, as summarized in Section 8, current research mostly applies off-line algorithms such as decision trees, Bayes nets or association rules along with time series analysis and statistical tests. For analysis of our data set we decided to start with a simple decision tree algorithm and then gradually extend it.

**Decision trees for important attributes**

The basic algorithm uses decision trees to generate interesting rules about the behavior of the system. Some of the reported attributes indicate a possible error: *error-code*, *response-code*, *result*, or *db-error*. For the system operators it is important to understand what requests report different values of such attributes. Since we prefer output in human-readable format, it is natural to use decision trees to classify the values of such attributes using the values of the remaining attributes.

Thus, this simple algorithm would look at all requests on one machine during a specified time interval (say, an hour), treat one of the attributes as a *class attribute* and train a decision tree using this data set. An operator can then easily extract useful rules that define the classes.

It is often not enough to use only the original attributes from the raw data. An operator who understands the system might like to add *new attributes* that he would like to classify or use in classification of other attributes. For example, since most attributes are not reported most of the time (such as *error-code*), the presence of an attribute indicates an event of interest. Therefore, for every attribute $A$ in the original raw log data, we define new boolean attribute $R_$A: *"is attribute* A *reported?"*.

An anomalous value of a numeric attribute (such as *duration*) is yet another interesting event. For every numeric attribute $B$ in the original raw data, we define new boolean attribute $O_$B: *"is attribute* B *anomalous?"*. We compute the average $\mu_B$ and standard deviation $\sigma_B$ of the past values and define the new attribute to be true if its value is outside the following interval: $(\mu_B - 3\sigma_B, \mu_B + 3\sigma_B)$.

The algorithm described in this subsection (referred to as algorithm $A_1$) was implemented and the results are presented in Section 6.

**Alternate algorithms**

In the more advanced algorithm ($A_2$) we would like to correlate the performance and request data. We noticed many short peaks in the performance data that last only for one 5-minute interval; values in the previous and the following 5-minute intervals appear normal. It would be interesting to compare the requests during the *anomalous* interval with the requests during the previous or the following interval. The details of the algorithm are as follows: 1) look at the performance data and detect outliers/peaks, and 2) after detecting an outlier at time interval $T$, build a data set from requests during interval $T-1$ (class *normal*) and requests during interval $T$ (class *outliers*). The decision tree algorithm will then try to classify the requests into these two classes; the resulting tree would indicate the most important differences between the classes.

Other extensions include using data from all machines to detect types of requests of particular machines that cause problems and using a better outlier detection algorithm such as a one-class SVM (algorithm $A_3$).

We tried to perform the required preprocessing ad-hoc using Python scripts, but we found that it is very tedious and time consuming to do, as described in the next section.

## 3. Practical problems of log data processing

When presented with terabytes of data, before trying out any SLT algorithm, we need to think about how to handle it efficiently. In this section, we describe our experience with analyzing huge amounts of data. We first describe our early attempts which use ad-hoc Python scripts and traditional relational database systems for handling the data. We discuss the need for a flexible architecture both for processing raw log data and for providing input for SLT algorithms.

### 3.1. Preprocessing data for SLT algorithms

In practice, system log data are in arbitrary formats, and thus, far from ready to be fed into an SLT algorithm. The required preprocessing includes the following:

- **Sampling**: we need to sample the original log because we are not able – and it is not necessary – to look at all the observed data. Other reasons for sampling include temporally variable data rate, dealing with unbalanced data sets, removing duplicate entries or removing entries that do not report the class attribute. Thus, a reconfigurable sampling algorithm must be supported.

- **Cleaning the data**: we need to filter out some unnecessary attributes from each of the event log entries: attributes not reported by any sample and attributes with constant values.

- **Adding new attributes**: the original attributes in the raw data are often not enough or not suitable for analysis using SLT algorithms, as described in Section 2.3. Values of some new attributes are easily generated; *"is attribute A reported?"*. However, other attributes – such as *"is attribute B anomalous?"* – might require running a simple algorithm.

- **Integrating streams from multiple sources**: System logs are generated on separate machines describing different aspect of the system. For example, our data set contains performance statistics, request log and problem tickets, as described in Section 2.1. It is also often necessary to integrate data sources unanticipated at design time, since we might find more related information as our familiarity of the system increases.
- **Running multiple algorithms**: Applying several different algorithms to our data is an important part of our research. However, different types of algorithms (such as on-line and off-line) require different experiment setup and many publicly available implementations require different format of input data. Because of huge amounts of raw log data, accessing it is a very expensive operation. Thus, we need to produce a separate output file for each algorithm with one scan of the raw data.

We needed a flexible system that can be configured to generate input data for SLT algorithms in a more convenient way.

## 3.2. Early experience with Python scripts

At the early stages of this project, we did not realize all of the practical problems discussed in the previous section. In order to get started testing the algorithms as soon as possible, we began writing Python scripts to process the data.

Python is a scripting language, which is very efficient (in terms of code length) in processing text files. The simplest preprocessing (scan through the data, project certain attributes of each entry and output them as a text file) can be expressed in about 50 lines of code, and some of our original results were obtained with the data preprocessed in this manner.

However, as we were trying out more algorithms, we found ourselves frustrated by the following problems:

1. **It takes a huge amount of time to scan through the data.** In our case, a single scan through one minute of log data takes more than 10-12 minutes. Most of the time is spent on reading the data from disk, uncompressing it, and parsing it.
2. **It is hard to handle multiple queries in a single script.** Producing data for multiple algorithms in a single script makes the Python scripts significantly more complex. A couple of aggregation queries take about 150 lines of Python code. It became even more complex when we wanted to save some of the intermediate results to disk for future use.
3. **It is hard to add/modify existing queries.** Adding one query to the code may require changing the code for existing queries, because we share the buffer and intermediate results among the queries.
4. **Fine grained parallelism is much harder to achieve.** We observed that preprocessing is CPU bounded (instead of I/O bounded) on our cluster, so speedup can only be obtained by using multiple processors. Parallel processing of the data is very hard to implement in fine granularities (such as at single-request level).

Our experience shows that even though ad-hoc scripts are enough for small data sets, they can be very difficult to maintain if re-running the script is slow and the set of SLT algorithms we need to preprocess the data for is not known in advance.

## 3.3. Problems with relational databases

We also considered using traditional relational databases for our data, since the preprocessing can be specified as SQL queries, reducing the complexity of preprocessing scripts. We rejected this approach for the following reasons:

1. **System logs do not have a fixed schema.** System logs usually have no fixed schema. Efficient relational database operations require a well designed schema; both logical (tables) and physical (file organization, indexing etc.). Changing schema is assumed rare and expensive. However, the format of system logs changes as the system evolves.
2. **One-time-queries are not suitable for generating multiple data output.** The queries in a relational database are *one-time queries*; generating another result usually requires a separate query. If a scan is required on a terabyte data base, each of the queries will take days to run.
3. **It is hard to support queries involving temporal properties of data.** However, this is essential in temporal data analysis.

4. **The cost of using a relational database is high.** Importing multi-terabytes of data into a relational database would have brought to us very high initial cost (both I/O and CPU) and storage cost.

## 4. A flexible architecture

In this section, we introduce a better data model – *data streams* and Telegraph Continuous Query processor to solve the problem described in previous section.

We focus on the functionality and flexibility of our software architecture and experience we get from running a prototype implementation. For the interested readers, we describe the details of the design and implementation of the system in Section 7.

### 4.1. Stream model of system log data

As our early experience suggests, a good data model is necessary for processing huge amounts of system log data. A *data model* is a collection of high-level data description constructs that hides the underlying low-level storage details [16]. It helps people to understand the data better and they can build proper data processing systems to manipulate the data.

We found that system logs fit the stream data models well, because of the following characteristics [2]:

1. Log data entries arrive on-line. The rate is determined by the data source and the temporal rate variation can be large. Also, the stream processor does not have any control over the order in which data elements arrive. In large scale distributed computer systems, such as the system X, logs are generated continuously on each of the machines with different data rate.

2. A data stream is an infinitely long sequence of data elements, but the memory on the stream processor is limited. Once a data element is processed, it has to be either discarded or archived, which makes it hard to locate the element and process it again. This situation fits our application well, since old system events are much less valuable, for failure detection, and even for long term client behavior analysis. Once the desired statistics have been obtained from the raw data, it becomes less important, and people usually do not have a chance to look into the old data again. Typical companies archive raw system log data for a few weeks before discarding them, but the statistics and data representing interesting system events are preserved. Stream processing and SLT can help us find and preserve the interesting data more efficiently.

3. A time stamp is attached to every entry in the data stream explicitly or implicitly (i.e., the arrival time at the stream processing system). Therefore, the system preserves the temporal properties of the data. It also allows most recent data to be accessed more efficiently so that temporal algorithms can be used easily.

The benefits of using data stream over ad-hoc scripts and relational databases are the following:

**Continuous queries**
The queries on data streams are usually *continuous queries*, in contrast to one-time-queries. One-time-queries run on a snapshot of the data, and return a single result to the user, while continuous queries are evaluated as data elements in the stream arrive.

Here is an example of a continuous query: *Which machine has handled 5 times more requests than any other machine over the last 10 minutes?* This query has to be re-evaluated each time a new system log entry arrives, and thus produces an output data stream containing the name of machines. The output stream can be used directly as an indicator of system failures or can be used as a regular data stream, for example, as an input to an SLT algorithm. Continuous queries can either be pre-defined or ad-hoc, and multiple queries can run on a single data stream concurrently. The ability to perform continuous queries has great advantage for preparing data for SLT algorithms.

**Off-line SLT algorithms can be used too**
As described in Section 8, many commonly used SLT algorithms are off-line algorithms (e.g., our decision tree algorithm), which work on *chunks* of data instead of streams. It is trivial to accumulate preprocessed stream in a buffer to get a data chunk large enough for the off-line algorithm. It is better than traditional methods in that preprocessing the data *before* buffering it allows us to save only the data we want, thus making the buffering much more efficient.

**Easy-to-change schemas**
It is easy to change stream schema, since there is no data stored in database. This makes it easy to add new streams and modify the output desired. For example, to generate the data for algorithm $A_2$ (Section 2.3) we needed to integrate performance and request data.

The idea of stream processing has been commonly used in the past. For example, *grep* in UNIX is a program that processes stream queries specified as regular expressions. The Python scripts we wrote are also

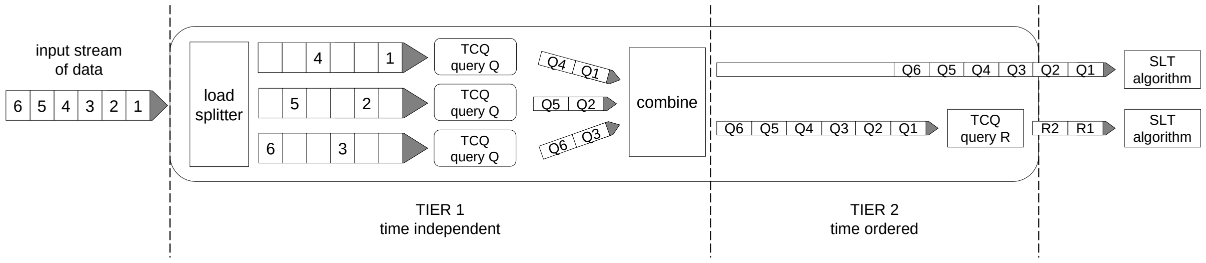

Figure 1. A general structure of the system. We used publicly available TCQ implementation as our major building block, and other components are currently written in Python. Our data are originally stored in a single data file. To make the data processing rate as fast as possible, we used 4 machines as stream sources, and load splitter is used to split the the stream round-robin by time to a set of (up to) 36 identical TCQ instances. Before splitting, a unique sequence number is assigned to all the entries of the log to allow a later reconstruction of the original order. The first tier of TCQ nodes performs queries that are independent of time (i.e. queries that do not have a time window specification). The output streams are directed to the stream combiner to reconstruct the time order. If many output streams are required, multiple combiners can be implemented. After the streams are combined, the second tier of TCQ instances performs time dependent queries. The final output streams are output to SLT algorithms. The output from SLT algorithms is also modelled as a data stream which can be displayed in a GUI and/or redirected to a centralized controller as feedback from the system.

stream processors. However, they resulted in ad-hoc and complex solutions.

## 4.2. Overview of the architecture

We wanted to build an infrastructure to support data analysis research of system log data. A major concern is *simplicity*. It should be simple enough that the initial configuration should either be automatically generated or be specified with a high-level description. The interface between our architecture and the system monitored should allow *easy deployment* in production environment. This architecture should be *flexible* enough to accommodate many algorithms, both on-line and off-line, without significant re-configuration. It should also be easy to add or remove data streams and components.

The purpose of the system is also to *make the algorithm implementation as easy as possible*, so that SLT researchers can focus on the algorithm rather than on tedious job of accommodating various input formats of raw data.

We tried to *make use of available software* from other research projects. The major component we use is Telegraph Continuous Query engine (TCQ) [12, 18]. TCQ is a continuous dataflow processing engine build on the code base of PostgreSQL, a popular open-source object-relational database system. It contains functionality of both relational database and stream processing. It supports continuous queries over streams, the cost of query evaluation is shared among all queries and the executor adapts to the characteristic change of the streams. It overcomes the problems with relational databases discussed in Section 3.3.

The use of TCQ helped us to easily specify and add/remove continuous queries, which solved the second and third problem discussed in Section 3.2.

*Turn-around time* is our next concern, or more specifically, the delay before one can start evaluating an SLT algorithm. We organized our architecture in multiple tiers and run each tier in parallel. Our software architecture is build on TCQ, which has all benefits of TCQ and it allows user to specify fine-grained parallel execution over a computer cluster to achieve short turn-around time and scalability. The main features include:

1. Data in the system are modelled as data streams, which are easy to understand and manipulate. Design of the system is driven by the flow of data. The output of one stream processor can be used as input of another and any stream can be buffered and used by an off-line algorithm. Another advantage of using streams, is that users unfamiliar with SQL can simply specify their queries in other languages such as Java.

2. It is easy to buffer a stream of data for a certain period of time to support off-line algorithms that require chunks of data. Result-saving policy can be specified separately for each stream in order to deal with temporal variation of stream data rate, importance of different streams and storage con-

```
select avg(f_delay) as f_delay_avf,
       stddev(f_delay) as f_delay_stddev
from rawlog
where f_app='http'
group by f_app window r ['30sec'];
```

Figure 2. A continuous query that computes the average latency for HTTP requests in the input stream over the last 30 seconds. This query can only run on tier 2 in Figure 1, since it involves time-dependent queries.

straints.

3. It is also simple to cache/store any intermediate stream to disk and reuse it later. This is especially important for research purposes, as we are constrained by the hardware resources available to us.

The architecture also supports the functionality described in Section 3.1, which we present with an example in the following section.

## 5. Experience with our prototype

We implemented a prototype of software architecture on a local cluster. The prototype is designed to be modular enough so each component can be easily replaced as long as it follows a simple stream interface. Major components include: data source, load splitter, stream combiner, TCQ processor, and SLT algorithm wrapper. The structure of the system is easily specified in an object oriented way (description of the components and the interconnections among them), and then automatically translated into a sequence of scripts which start all the components on multiple nodes of the cluster.

The general structure of the architecture is described in Figure 1. The functionality and interface of these components are described in more detail in Section 7. Here we focus on how to use the system and the benefits of using it, which we believe is of more interest to our readers.

This example uses two tiers; the first tier performs time-independent sampling and processing that reduces the size of the stream for time-dependent processing. The data are currently processed in the order they enter the system for the ease of implementation. We consider more advanced load balancing as an important part of our future work.

**A simple query**
We start with a simple query from a traditional monitoring system; query R: *the average latency for HTTP requests over last 30 seconds*. Note that this query cannot be run on TCQ instances of tier 1 (query Q on Figure 1), since not all the data for "last 30 seconds" is present in either one of the queries Q. Instead, query Q just performs sampling to decrease the size of the stream (time independent) and the actual query R is executed *after* the combiner in the second tier of TCQ instances. Adding a query (as specified in Figure 2) only takes one SQL statement and running a single-line shell command and can be done on-line without stopping the stream sources.

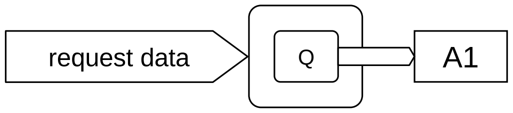

Figure 3. System architecture for algorithm $A_1$. The outer rounded rectangle corresponds to the whole architecture from Figure 1. For this algorithm the input stream consists of 450 attributes. If an attribute is not present in the entry, its value is defined as NULL.

```
select f_class, f_app, f_machine, f_bytes-served, f_ip,
       (f_error-code is not null) as f_error-code_reported,
       (f_duration < LOW OR HIGH < f_duration)
          as f_duration_outlier
from rawlog
where f_app='proxy'
      f_machine='machine54'
      f_class is not NULL
```

Figure 4. Query Q for algorithm $A_1$: the first line projects the specified attributes and the next two lines generate two new attributes (Section 2.3) from raw log. The `where` clause specifies that only accepted entries are from machine `machine54` and application `proxy` that also report a value for the class attribute. Note that we prefer to let this query run on tier 1 (Figure 1), since the output stream is much smaller than the input.

**Supporting Algorithm $A_1$**
Figure 3 presents the simplified system configuration for algorithm $A_1$ (Section 2.3). In particular, we want to generate a decision tree for attribute *f_class* for requests from machine `machine54` and application `proxy`. Query Q on Figure 4 shows all the required preprocessing. The output stream generated by Q is formatted as comma-separated-values and can be directly used in a decision tree implementation. Note that this stream must be buffered to form a chunk before passed to the algorithm. The description of the system takes 48 lines of code.

The speedup when using multiple machines in parallel is shown in Figure 5. We used 4 parallel stream sources (i.e., we split the log file containing data for all

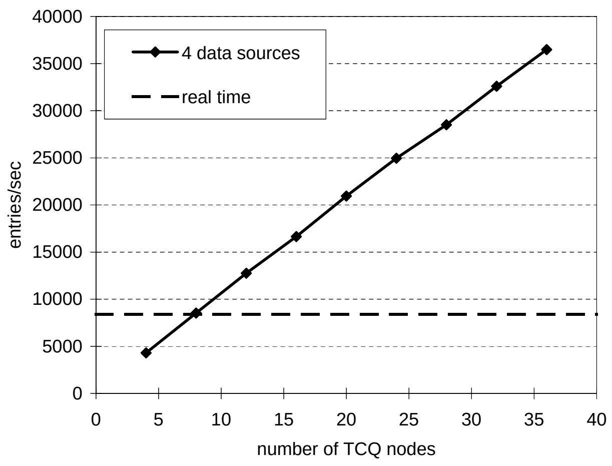

Figure 5. When increasing the number of TCQ instances running in parallel, average throughput increases linearly. System X generates about 8360 log data entries per second which can be handled on 8 TCQ nodes in real-time. 36 TCQ nodes can handle 36500 entries per second which is ≈ 600GB of log data per day. We used 4 parallel data sources; with fewer data sources, their throughput will eventually become bottleneck of the system.

of the 430 machines into four pieces and used 4 nodes to generate stream). Running up to 36 TCQ instances in parallel, we observed linear speedup and the system processed 1 minute of the system log (501572 log entries) in 13.7 seconds. We think that this turn-around time is good enough for us. Also, since the data are produced in time order, we can run algorithms on the output streams without waiting for the processing to complete. In a real world implementation, the data source itself is naturally parallel since the logs are generated on different machines.

### Supporting algorithms $A_2$ and $A_3$

To support algorithm $A_2$, we need to add the performance data stream to the system. This can be done by simply adding a data source and since the data rate is very low, a load splitter is not required. We connect this stream directly to one of the TCQ instances in the second tier, after the first tier processing is done. This TCQ instance looks at the performance metrics and uses their average and standard deviation to detect outliers. The output stream of this query (called a *control stream*) is non-zero when an outlier is detected. The control stream is then connected to another TCQ instance in the first tier that generates an appropriate stream for $A_2$.

The architecture for $A_3$ presented on Figure 6 is very similar to $A_2$. The outlier detection is now performed by an external algorithm – an off-line one-class SVM. The performance data stream is thus forwarded to two components: one that generates the model and another one that use a previously generated model for outlier detection. The rest of the architecture remains unchanged. These simple changes to the architecture for $A_1$ show the flexibility of the system.

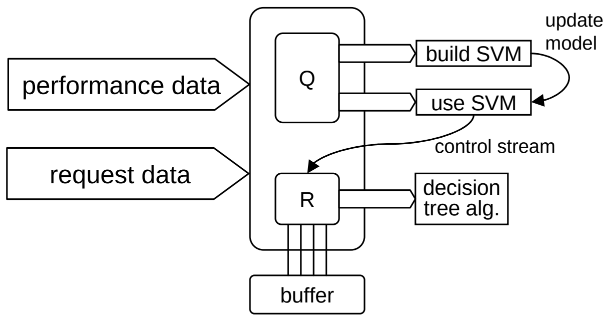

Figure 6. structure of the architecture that supports $A_3$. The data stream is used to build a Support Vector Machine (SVM), which is an off-line algorithm. the SVM built is then used on-line to detect outliers. Once an outlier is detected, a new query is initiated so that the data from last 10 minutes are preprocessed for decision tree algorithm C4.5 as described in Section 2.3.

### Supporting any algorithm

Since the data flows in streams that can be buffered we can use this architecture with any on-line or off-line data mining or SLT algorithm. All entries entering the system contain a time stamp and, in addition, we assign them a unique sequence number. All the processing (such as SQL queries or merging different data streams) preserves the original order of entries and thus temporal algorithms can be used too. Further, because the output of any algorithm is also modelled as a stream, we can arbitrarily combine multiple algorithms.

### Updating a component of the system

In our initial setup, we used a very simple stream source component which was implemented using a single-threaded Python script. After the system was up and running, we found its performance unsatisfactory, so we implemented a new stream source with multi-threading. Removing the old source and adding the new one took only two shell commands (one to stop the old one and the other to start the new one), while all other components were left intact without restarting. The whole process took only a couple of minutes, before the system continued to produce new output data.

All the stream source implementations are straightforward and even the multi-threaded one consists of only 90 lines of Python code.

## 6. Results of algorithm $A_1$

In this section we present results of the algorithm $A_1$ from Section 2.3. We used the standard implementation of C4.5 decision tree algorithm from the machine learning tools package Weka [23]. The depth of the trees was constrained to be no more than 7 and the minimal number of samples in a leaf was set to 10.

The results are based on 4 hours of request data from a single machine running mainly the proxy application. The following preprocessing was applied: a) add new attributes as described in Section 2.3, b) remove useless samples (ones that did not report the class attribute), c) remove useless attributes (ones that are constant), and d) resample the data to obtain smaller data set (about 20 - 30 MB) and to balance the classes of the class attribute. The resulting trees are presented in Figure 7. The decision trees `B'` and `C'` were generated from the original data set after filtering out attributes `R_cache-served` and `duration`, respectively.

### How useful this is for the operators

As this is work in progress, we didn't yet carry out any extensive evaluation of this algorithm. However, the early results are encouraging; we showed these decision trees to one of the operators in company X and he thinks they are very useful.

For the operators, there are two main characteristics that make this approach very valuable. First, the decision tree algorithm can automatically search through the large number of attributes and find a small subset that is correlated with the class attribute. For example, the first decision tree for attribute `R_error-code` says the following: *"the requests that report an `error-code` (almost certainly) do not report attributes `cache-served` and `server-duration`"*. The structure of the decision tree also allows the operator to quickly identify points of interest: for example URLs `6520...` and `2336...` (see the decision tree `B'`) almost always generate an `error-code`. The possibility of defining new attributes on-the-fly makes this very attractive.

We can hardly expect an automated analysis system to replace the operators; an ideal system will thus accept feedback from the operators and improve its analysis accordingly. Decision trees allow a simple version of this: if the decision tree generates rules that are trivial for the operator and he understands them, a few attributes can be filtered out from the data set and the algorithm can generate an alternative explanation.

```
A:  class attribute: error-code
    bytes-served <= 195: 145 (135/9)
    bytes-served > 195
    | R_content-len = yes: 32 (98)
    | R_content-len != yes
    | | R_not-cached-reason = yes: 32 (45/19)
    | | R_not-cached-reason != yes
    | | | duration <= 15.2
    | | | | bytes-received <= 2680: -13 (39)
    | | | | bytes-received > 2680
    | | | | | bytes-received <= 2805: 131 (30/7)
    | | | | | bytes-received > 2805: -13 (85/13)
    | | | duration > 15.2: 131 (69/6)
                    -------------

B:  class attribute: R_error-code
    R_cache-served = yes: no (10469)
    R_cache-served != yes
    | R_server-duration = yes: no (7686)
    | R_server-duration != yes: yes (18094/5)

B': class attribute: R_error-code
    attribute R_cache-served removed
    duration <= 2.25
    | client-write-duration <= 0.0: yes (200/4)
    | client-write-duration > 0.0
    | | bytes-served <= 210
    | | | R_server-duration = yes: no (873)
    | | | R_server-duration != yes: yes (1969/10)
    | | bytes-served > 210
    | | | visit-url = 6520...: yes (69)
    | | | visit-url != 6520...
    | | | | visit-url = 2336...: yes (72/1)
    | | | | visit-url != 2336...: no (18909/1934)
    duration > 2.25
    | R_server-duration = yes: no (291)
    | R_server-duration != yes: yes (13866/6)
                    -------------

C:  class attribute: O_client-write-duration
    duration <= 9.71: false (18018)
    duration > 9.71: true (18231/71)

C': class attribute: O_client-write-duration
    attribute duration removed
    bytes-served <= 67958
    | R_error-code = yes
    | | R_content-type = yes: true (253/6)
    | | R_content-type != yes: false (17)
    | R_error-code != yes
    | | gmt = 2003-06-24 00:01:07: true (54)
    | | gmt != 2003-06-24 00:01:07
    | | | user-id = 96848766314153157: true (99/6)
    | | | user-id != 96848766314153157
    | | | | gmt = 2003-06-24 02:23:28: true (45)
    | | | | gmt != 2003-06-24 02:23:28
    | | | | | visit-url = 8227...: true (43)
    | | | | | visit-url != 8227...: false (18005)
    bytes-served > 67958: true (17733/55)
```

Figure 7. Binary decision trees for the attributes *error-code*, *P_error-code*, and *O_client-write-duration*. Every line represents one conditional branch in the tree. The class attribute is in bold. The numbers in brackets represent the number of samples in that leaf (x) or number of samples and number of false positives for that leaf (x/y). The top of the tree `A` thus means: "If *bytes-served* $\leq$ *195*, the value of *error-code* is **145** (if the error code is reported). This rule is valid for 135 samples and invalid for 9 samples. If *bytes-served* $>$ *195* ..." (we move to the other branch). The attribute "R_*attr*" represents a newly generated attribute "was attribute *attr* reported?", "O_*attr*" represents "is attribute *attr* an outlier?".

An example can be seen in the decision tree for attribute `O_client-write-duration`; the first decision tree generated a simple decision tree that is probably clear for the operator (*"attribute `duration` is almost perfectly correlated with the outliers in attribute `client-write-duration`"*). After removing the attribute `duration` from the data set, the alternative decision tree offers more interesting insights.

Remember that these results were generated using data from a single machine and a single application running on that machine. After we add requests from all the machines and all the applications we get a much more powerful tool that allows us to correlate events across the whole cluster.

## 7 Design and Implementation details

In this section, we describe the components and their interfaces in our architecture in more detail.

### 7.1. Key component, Telegraph Continuous Query Engine

General stream processing techniques have been studied in database community in great depth [1, 6]. A number of general purpose stream processing systems have been built [2, 12]. We use Telegraph Continuous Query engine [12], which is designed to process data streams with adaptive, continuous queries.

The queries are specified in PostgreSQL SQL, with all data types and functions. One query is usually specified by a few lines of SQL and all the query plans are automatically optimized. This makes adding and modifying queries significantly easier than ad-hoc scripts.

As the characteristics of the data stream change, the query execution changes adaptively. For example, during a system failure the average delay may suddenly go very high and thus a selection condition "delay $>$ 10 sec", which normally throws away almost all tuples suddenly becomes not selective. Without adaptive query execution, the query evaluation may become very inefficient for other operators in the query plan.

TCQ supports running multiple queries on a single data stream and generating multiple outputs concurrently. This best fits our case of running different SLT algorithms requiring different input data on a single log file. The computation and storage are shared aggressively, so running more queries on the same stream does not increase workload significantly.

TCQ is still in its early research stage [18]. The released version is functional, but not optimized for performance. A single node running TCQ takes 7 to 14 minutes to process one minute of system log data in our data set, which is the reason why we used multiple instances of TCQ running in parallel. We are interacting with the TCQ research team to investigate higher performance and new features.

### 7.2. Other building blocks

Modularity is an important goal of our design. We designed the system so that it consists of simple building blocks (see Figure 1) that communicate with each other using sockets. They can be deployed on a single physical node or over multiple nodes in a cluster. These components were written in Python and comprise about 750 lines of code.

**Data source** is the interface for getting the various kinds of data, translating them into data streams and feed them into the stream processing system. It provides a small interface to the production system, and can be overridden to use multiple types of data, such as logs stored on disk, network monitoring readings, or live stream of system event reports.

**Load splitter** is a small component used for load balancing that takes a single input stream, divides it into multiple streams and redirects them on to multiple nodes. When the data rate cannot be handled by a single TCQ instance, we create multiple instances and use the load splitter to route the stream to all the instances. The data processing within the load splitter should be simple and fast, since it is on the critical path of the system and always sees a large data rate. Currently we think the best algorithm is splitting by time, i.e. sending data elements in round-robin manner to each of the stream processors. This provides best throughput since there is no cost for examining and parsing the data. Of course, more advanced load splitter can be built, especially considering the properties of the source stream. Load splitter can be changed in the system without affecting any other components.

**Stream combiner** is a component used to combine multiple streams generated by load splitter to re-create the original order of entries. It works on the original time-stamp attached to each entry in the stream. If the time-stamp is too coarse grained to order the entries (for example, there are 600 events in a single second and some of them have causal dependency), we attach a unique sequence number to each entry when it is pushed into the system.

**TCQ instances**, as described in Section 7.1, are the key components in the system. They take in multiple SQL queries, multiple data streams and output the results of the queries as data streams. The output data streams can be buffered for off-line algorithms or written to files for future use. The raw stream can be configured to be archived. The TCQ instances also output its own performance statistics as data streams (which is called introspective query) to centralized controllers.

**SLT algorithms** are defined as a components taking data streams or a file as input and output another data stream. The data stream can be interpreted by a GUI component for human administrators to review or achieved for future reference. Publicly available algorithms can be plugged into the system with only a minor wrapper for reading data streams (for example, through JDBC or simply through a UNIX pipe).

Note that the data streams between each component are not necessary in the same format. They can be implemented as text streams, but we can also use binary streams with type definition which saves parse time. Changing the format of a stream is simple; it only requires to change the output format of the sender and input format of the receiver or add a separate wrapper around the receiver.

### 7.3. Putting everything together

We provided a simple way to build the system from components with simple object oriented specifications. All the components are modeled as a class. To build the system, one only needs to specify the parameters of the components or override some of the functionalities and the interconnections among them. A program we provide automatically generates a shell script that starts the components on multiple machines in the cluster. There are separate scripts for adding and removing streams and queries from each.

There are two steps to add a new SLT algorithm. First, the algorithm "subscribes" a stream from the system, which is done by specifying a new query. Second, the user may need to write a simple wrapper to generate the correct format and/or add a header.

We feel that although now it takes only less than half an hour to write the script that generate the architecture in Section 5, it would be more convenient to use a GUI.

## 8. Related work

This research is a multi-disciplinary research involving SLT, data mining, dependable system design, data warehousing and processing.

**SLT for systems**

Recently, a few researchers started using SLT algorithms for detecting and localizing system failures and software bugs.

In [4], Chen uses decision trees for localization of failures on the eBay web site. Each executed request reports attributes such as name, type, machine, version and status of the request. A decision tree is trained to predict the status attribute and the generated rules are used to localize what machine, type of request, or version of software is causing problems.

In his work on Pinpoint [3, 9], Kiciman instrumented the JBoss application server so that a J2EE application reports *execution paths* of all requests. The path is a list of J2EE components that the particular request used. Pinpoint can detect anomalous paths and correlate them to identify the failed components.

Cohen *et al.* [5] uses Tree-Augmented Bayes Nets for automated performance analysis. 124 types of performance metrics are measured on a sample server and the induced model is used for prediction of Service Level Objective violation.

Researchers at IBM Research [21, 22, 17] apply temporal data mining and time series analysis to predict critical events in computer system such as *high CPU utilization* or *imminent router failure*.

Liblit [10] proposes a sampling infrastructure for gathering information about execution of C programs. He instruments the source code of the program at every branch, assignment and function call. The recorded information from runs of the program that crash are correlated to obtain the possible bugs.

Most of the work mentioned above has the same goal as our research – use automated analysis of computer systems. However, experiments conducted in the referenced papers use smaller data sets (10 – 100MB) compared to our data set of a few terabytes.

**System monitoring and management**

There have been a lot of efforts on monitoring systems in both academia and industry. Simple Network Management Protocol (SNMP) [8] allows user to instrument and monitor aggregated performance of heterogeneous component in a network environment. It provides a visualized and hierarchical infrastructure to support high volume data collection and separating management boundaries.

There are commercial tools that allow user to monitor and do simple analysis on the data collected. The major tools include HP OpenView (http://www.openview.hp.com/), IBM Tivoli (http://www-306.ibm.com/software/tivoli/), Microsoft Operations Manager (http://www.microsoft.com/mom/). These tools allow user to navigate through the collected and stored data, and run statistical analysis on them. However, they are not designed for preparing data for SLT algorithms.

Traditionally, the collected data are sent to some centralized servers which may waste bandwidth. Both Astrolabe [20] and PIER [7] manage to collect and analyze the data on the node where they are generated. Astrolabe makes use of gossip protocol and the architecture is formed in a hierarchical structure of domains. PIER is implemented on a DHT [19]. Both allow user to run queries in SQL which are then evaluated in a distributed way in the system.

**Stream data processing and mining**

Our work is also related to the stream processing and data mining work in database community. Stream processing addressed the issue of dealing with data that arrive in multiple, continuous, rapid and time-varying data streams [2].

A number of stream processing systems have been proposed to handle continuous queries over a data stream [2, 1]. TelegraphCQ [12] addresses this problem with eddy query processing framework that adapts the temporal variation of data streams in data rate and statistical characteristic of the data stream. It also allows to share evaluation path among multiple queries.

Several new algorithms that are suitable for mining data streams were proposed. The characteristic of most of these algorithms is that they only look at every tuple in the stream once [6]. In contrast, for most of the SLT algorithms it is not enough to look at each data tuple just once. Buffering and caching of old data are supported in our work to solve this problem.

Stream processing is also used in sensor network data monitoring and analysis [11]. Though the data rate from sensor network can also be high, it is much less complex than logs generated by a large cluster of computers.

## 9. Future work

As a joint disciplinary research project, we have two tracks of future research that are interleaved.

**SLT algorithms**

There are a few interesting challenges for SLT researchers. To understand a system better, we can – in the extreme case – instrument every line of source code of the applications running on the servers. However, this instrumentation is highly redundant and so an interesting question to ask is *"how much instrumentation do we need to get the best results?"* Source code instrumentation at even a much coarser level will generate a few orders of magnitude more attributes; *"how can we select a subset that contains the most interesting attributes?"*

Another problem of current SLT algorithms is that they assume global knowledge of all the data. However, with tens of thousands of machines, the central storage will certainly become a bottleneck; we cannot even download all the data. The possible solutions include: a) store history of the raw log data in the machines, monitor only a few selected attributes and download the necessary data only after we detect a problem, b) do early processing of the log data in the machines, or c) sample all the attributes and over time decide which ones are less/more important and sample slower/faster.

**Software architecture**

We will turn the prototype discussed in the paper into a more general toolkit and make it publicly available to the SLT researchers. There are several other components that we plan to add into our system, which are not implemented in the prototype.

- We want to support the operator-automated system interaction. We want to implement this by a GUI for viewing the output of the algorithm, modify the parameters and data fed into the algorithm to generate better result. We also want to model human input as another stream into the system and let the system adjust queries automatically.

- Adding a centralized controller to the architecture, which collects statistics from different components and dynamically allocates physical nodes to each of the components. This gives us the ability to monitor and analyze the architecture itself. We can monitor the load of each component of the system and dynamically balance load when sudden load change happens or query behavior changes.

- A distributed optimizer for query evaluation. Currently adaptive query executor only works on single node, but we need a centralized/distributed query optimizer in order to remove the job of routing the stream manually among the nodes, and thus providing a single system image of the distributed architecture.

## 10. Conclusion

Our experience suggests that we cannot use ad-hoc systems for automated analysis of huge system log data set from on-line services. Instead, we propose a modular architecture that was shown to easily handle 600 GB of system log data a day. The architecture is flexible enough to be used for any type of on-line or off-line data analysis algorithm.

## References

[1] C. C. Aggarwal. A framework for diagnosing changes in evolving data streams. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 575–586. ACM Press, 2003.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM Press, 2002.

[3] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. DSN 2002.

[4] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer. A statistical learning approach to failure diagnosis. In *International Conference on Autonomic Computing (ICAC-04), New York, NY*, May 2004.

[5] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *OSDI*, 2004. To be published.

[6] M. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: you only get one look, a tutorial. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 635–635. ACM Press, 2002.

[7] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *Proceedings of the 29th VLDB Conference*, 2003.

[8] M. S. J. Case, M. Fedor and J. Davin. A simple network management protocol (SNMP). *RFC1157*, May 1990.

[9] E. Kiciman and A. Fox. Detecting and localizing anomalous behavior to discover failures in component-based internet services. Technical report, Stanford, 2004.

[10] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.

[11] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 491–502. ACM Press, 2003.

[12] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 49–60. ACM Press, 2002.

[13] J. Markoff and G. P. Zachary. In searching the web, Google finds riches. NY Times, April 13, 2003.

[14] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *4th USENIX Symposium on Internet Technologies and Systems*, March 2003.

[15] D. A. Patterson. A simple way to estimate the cost of downtime. Submission to 16th Systems Administration Conference (LISA '02), 2002.

[16] J. G. Raghu Ramakrishnan. *Database Management Systems.* McGraw-Hill Higher Education, 2003.

[17] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, and S. Ma. Critical event prediction for proactive management in large-scale computer clusters. In *KDD*, 2003.

[18] S. K. Sirish. Telegraphcq: An architectural status report.

[19] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In R. Guerin, editor, *Proceedings of SIGCOMM-01*, volume 31, 4 of *Computer Communication Review*, pages 149–160, New York, Aug. 27–31 2001. ACM Press.

[20] R. van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, 2003.

[21] R. Vilalta, C. V. Apte, J. L. Hellerstein, S. Ma, and S. M. Weiss. Predictive algorithms in the management of computer systems. *IBM Systems Journal*, 2002.

[22] R. Vilalta and S. Ma. Predicting rare events in temporal domains using associative classification rules. Technical report, IBM Research, T. J. Watson Research Center, Yorktown Heights, NY, 2002.

[23] I. H. Witten and E. Frank. *Data mining: practical machine learning tools and techniques with Java implementations.* Morgan Kaufmann Publishers Inc., 2000.