DIMACS Technical Report 2003-19
June 2003

Checks and Balances: Monitoring Data Quality Problems in Network Traffic Databases

by

Flip Korn
AT&T Labs-Research
Florham Park, NJ 02932
flip@research.att.com

S. Muthukrishnan [1,2]
Rutgers University & AT&T Labs
New Brunswick, New Jersey 08903
muthu@cs.rutgers.edu

Yunyue Zhu[3]
New York University
New York, NY 10012
yunyue@cs.nyu.edu

[1]Permanent Member
[2]This work is supported by NSF CCR 00-87022 and NSF ITR 0220280.
[3]This work was started while the author was a DIMACS visitor at AT&T Labs.

ABSTRACT

Internet Service Providers (ISPs) use real-time data feeds of aggregated traffic in their network to support technical as well as business decisions. A fundamental difficulty with building decision support tools based on aggregated traffic data feeds is one of data quality. Data quality problems stem from network-specific issues (e.g., irregular polling caused by UDP packet drops and delays, topological mislabelings, etc.), and make it difficult to distinguish between artifacts and actual phenomena, rendering data analysis based on such data feeds ineffective.

In principle, traditional integrity constraints and triggers may be used to enforce data quality. In practice, data cleaning is done outside the database and is ad-hoc. Unfortunately, these approaches are too rigid or limited for the subtle data quality problems arising from network data where existing problems morph with network dynamics, new problems emerge over time, and poor quality data in a local region may itself indicate an important phenomenon in the underlying network. We need a new approach – both in principle and in practice – to face data quality problems in network traffic databases.

We propose a continuous data quality monitoring approach based on probabilistic, approximate constraints (PACs). These are simple, user-specified rule templates with open parameters for tolerance and likelihood. We rely on statistical techniques to derive effective parameter values from the data, and show how to apply them for monitoring data quality. In principle, our PAC-based approach can be applied to data quality problems in any data feed. We present PAC-Man, which is the system that manages PACs for the entire aggregate network traffic database in a large ISP, and show that it is very effective in monitoring data quality problems.

1 Introduction

In the internet world, much has been discussed about various databases: database of customers and their purchases, web clicks, DNS lookups, etc. Our focus here is on network databases that are in the bowels of the internet, namely, those that capture the IP traffic in the routing elements of the internet.

Unfortunately, there is a fundamental data management problem that arises in network databases comprising (aggregated) IP traffic logs. That is one of *data quality.* The mechanism of automatically gathering data from management stations and setting up data feeds over the internet causes *unique* data quality problems; these are not problems that arise, for example, from manual entry of customer care records. (We describe this in more detail in Section 2.) These data quality problems that stem from network-specific issues make it difficult to distinguish between artifacts and actual phenomena. In turn, this renders data analysis based on such data feeds ineffective. This is of serious concern to ISPs.

There are ways to deal with data quality problems in practice and in principle. In practice, data quality problems are cleaned outside the database. For example, missing values may be interpolated, or the database may get populated with default values, etc. This is often ad hoc, executed in scripting environments. This approach do not scale as new data quality problems emerge, and old problems morph over time and due to network dynamics. In addition, developing scripts is time-consuming. Furthermore, synthetically cleaning the database may not be desirable because data quality problems may by themselves indicate important network phenomenon. For example, missing values may indicate loss of UDP packets which may in turn indicate high congestion levels in the network. In principle, data quality problems can be monitored using traditional integrity constraints (ICs) and triggers to enforce database integrity. Unfortunately, this approach is too rigid and limited for the subtle data quality problems arising from network data; it is not always possible to make a binary decision regarding the correctness of network data at insertion time and these tools do not offer the capability required to detect them. We need a new approach – both in principle and in practice – to face data quality problems in network traffic databases.

We propose a novel approach to detecting and monitoring data quality problems in network traffic databases. Our approach is inspired by database integrity constraints but is quite different since ICs are inadequate for network data feeds. We propose using *probabilistic and approximate constraints* (PACs). One imagines the exact constraints one would like to impose on the logical view of the network database. Then, one defines various templates that are probabilistic, approximate versions of these constraints with parametrized tolerance levels and likelihoods of violation. PACs are therefore user-driven rule templates which are specified by the DBA or analyst; the parameters are instantiated based on training data. We use statistical techniques to derive these parameters. Our data monitoring system now manages various PACs that are learned from the database and adapted over time, and which are used to monitor the quality of the data feed.

Our PAC-based approach is quite general, and can be applied in principle to any database. However, in this paper, we use the aggregate IP traffic database example to demonstrate

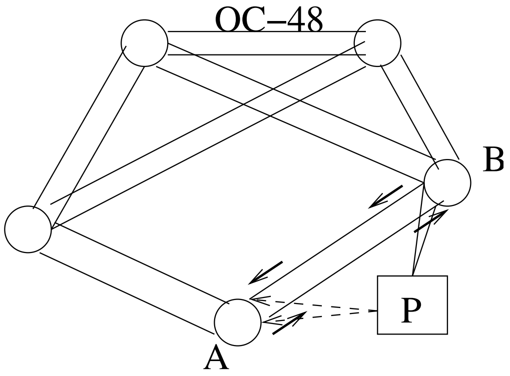

Figure 1: Network and poller

the effectiveness of PACs. *We have built a data quality detection and monitoring system for a large ISP on the entire aggregated IP traffic in their network, based on our PAC-based approach. Our system is fully operational over more than* 6 *months of this data at this moment.* All our experiments here are based on this real data. They show PACs are highly effective in both detecting data quality problems and in monitoring them over time. Our approach can learn stable and effective PAC rules from these real data.

The rest of the paper is organized as follows. In Section 2, we present a detailed view of network traffic databases and data quality problems. In Section 3, we provide detailed description of various types of PACs — domain, unique key and functional dependency-based PACs — all instantiated in the context of the aggregated IP traffic database. In Section 4, we discribe our system, PACMAN, that manages PACs and monitors data quality problems in network traffic databases. We also discuss the issues in integrating PACMAN into a database in section 5. We present our experimental study in Section 6, related work in Section 7 and concluding remarks in Section 8.

2 Data Quality in Network Databases

In this section we describe an example network database and the associated data generation and data quality issues.

2.1 The Network

We focus on a large IP network such as an Internet Service Provider (ISP) where there are typically thousands of network elements configured in a topological hierarchy consisting of routing devices, each containing multiple interfaces which are endpoints for incoming and outgoing links connecting the devices. These links can have high-speed capacities, up to

OC-48 (2.4 Gbps) or even OC-192 (9.6 Gbps). There can be multiple links between device pairs; traffic on any link is unidirectional. Thus, the network can be abstracted as a directed multigraph $G(V, E)$ containing a multiset of nodes (devices) V and a multiset of weighted edges (links) E connecting these devices; weights correspond to link speed capacities. Figure 1 depicts a toy example with links in both directions between devices A and B. The topology data also contains device and interface names, geographic locations, etc. We assume that snapshots of the topology data are obtained from router configuration files and provided on a regular basis (e.g., daily).

2.2 Network Databases: An Example

IP traffic consists of packets of data sent on links between routers, and can be captured in multiple levels. For example, there is a *physical level* traffic log which may comprise of the packet headers on the packets on each link: source, destination IP addresses, number of bytes in the packets, type of traffic, etc. Then, there is a *logical level* obtained by grouping packets on a link into logical "flows" and assembling packet level information into flow level information containing for example, source, destination IP addresses, number of packets and number of bytes in each flow. Finally, there is the *aggregated level* in which we only capture the amount of traffic — number of bytes — in each link per time interval, without regard to the source and/or destination IP addresses, type of traffic etc. As we zoom in from the aggregated level to the physical level, the amount of information in the logs increases because we get a refined look of the traffic data; but, the data sizes increase too, rather dramatically. Thus, while it is clear that finer traffic logs will reveal more about the network dynamics, few vendors, if any, provide data feeds at the physical level of packet traffic.[1] On the contrary, almost every router vendor supports aggregated level data logs. In fact, such aggregated trafic information is collected network-wide and used crucially in everyday network operations in today's ISPs.

We focus on network databases that store aggregated traffic logs of ISPs. These logs are collected by network management stations using the Simple Network Management Protocol (SNMP) that is ubiquitous. SNMP data is collected by polling IP network elements. Figure 1 illustrates a single *poller* P, which periodically requests measurements from interfaces at routers A and B; inbound as well as outbound traffic gets polled at both endpoints of each link. These requests are sent and received using SNMP, which is implemented over UDP. UDP has less overhead than TCP because it lacks a protocol for verifying packet receipts; this can result in packet drops and delays. For simplicity, we consider only the (cyclic) count of bytes for each interface along with a timestamp (e.g., in GMT) and an ID for the interface. We compute the difference between successive counts. Note that the counter can wrap between successive polls.

We say network traffic databases have a multigraph of topology and a data feed. The

[1] Special packet sniffers and monitoring tools are developed in research enviroments and deployed in test mode within ISPs, or special data processing systems are built in order to perform query processing in the router itself because ISP-wide datafeed of packet level traffic log is simply infeasible.

topology data is contained in two tables with the following schemas: `DEVICE (ifaceID, devID, lat, long)` specifies which device an interface is attached to and the geographic location of the device; `LINK (ifaceID1, name, speed, ifaceID2)` specifies the two endpoints(interfaces) of a link and the name and speed of the link. The data feed contains periodically-polled inbound and outbound traffic rates and is recorded with a timestamp as well as an interface ID. The schema is `TRAFFIC (date, time, ifaceID, inrate, outrate)`. We assume that this data is insert-only and arrives in time order, or is only slightly out of order so that it can be reordered within a window.

2.3 Data Quality Problems

The quality of SNMP-polled data is poor for a number of reasons. The data management systems that gather aggregated traffic statistics use the underlying IP network to collect the data. So if there are problems in the network — minor problems with unreachable elements are common — then the data feed is delayed or disrupted. Similarly, these stations use the UDP protocol to collect SNMP traffic, which is unreliable: congestion in links can lead to lost UDP packets which disrupts data feeds. Also, these stations poll the network elements periodically: there are variations in the polling times because polling proceeds in some sequence, different polling stations may not be synchronized, etc. This leads to some fundamental difficulties in logically modelling the network database. For example, trivially, one would like to believe that the traffic that leaves one end of the link is the same as the traffic that reaches the other end, for any given time interval. This is difficult to ensure if the two endpoints are polled at slightly different time intervals, as frequently happens due to the sequencing delays in polling network elements. Also, there are cases where elements get polled multiple times because different pollers end up having common polling responsibility due to errors. Furthermore, configuration tables at various pollers may be out of date or erroneous, which leads to further data quality problems in the polled data. Finally, there are various counters in routers for aggregating traffic data in the links. At backbone routers where the links are exceedingly fast, these counters can wrap-around very quickly, and the reading on the counters may have to be adjusted to the proper traffic count. To summarize, the mechanism of automatically gathering data from management stations and setting up data feeds over the internet causes *unique* data quality problems; these are not problems that arise, for example, from manual entry of customer care records.

We focus on SNMP data quality problems that arise from the architecture described in Section 2.2, such as the following:

- **Missing polls:** There are either continuous or sporadic chunks of missing data. This may be due to several factors such as faults in a network device (e.g., failures, memory over-utilization), dropped UDP packets from SNMP polls, or incorrect topology data.
- **Irregular polls:** The polling rates are irregular due to UDP delays or to high CPU utilization at the network elements.

- **Extraneous polls:** A mislabeling of interfaces in the topology metadata and multiple pollers can accidentally result in double-polling.
- **Repeated values:** This often results from preset defaults (typically zeros); however, sometimes there are suspicious-looking repeated values, perhaps truncations resulting from exceeded capacity or utilization.
- **Short, huge bursts:** These artifacts can be difficult to distinguish from actual phenomenon; unlike actual traffic bursts, these tend to be sporadic and abrupt.
- **Violation of network traffic laws:** Disparity in traffic received from and transmitted to a link or device (à la Kirchoff's Laws from circuits stating that the sum of the currents entering a node must equal the sum of the currents exiting a node).

These observed symptoms can serve as the basis for 'sanity' checks for data quality control in that they indicate underlying problems. Indeed, we will employ these observations in Section 3 for designing example rules. These are many other data quality problems which can arise that aren't listed here. This calls for a flexible approach to defining such quality checks.

3 Specific Constraints of Our Interest

In principle, traditional database mechanisms such as integrity constraints (ICs) and triggers can be used to enforce data quality inside the database. Unfortunately, this approach is too limited and rigid for the subtle problems arising from network data where existing problems morph with network dynamics, new problems emerge over time, and poor quality data in a local region may itself indicate important phenomena in the underlying network.

We propose a novel solution based on *probabilistic, approximate constraints* (PACs). Whereas traditional ICs exactly define a set of legal values, in contrast PACs are flexible, indicating the likelihood of correctness of a given value. These constraints are expressed as a quantity over a single or multiple attributes to be satisfied within a tolerance of ϵ. If ϵ is zero, then the constraint is said to be exact and the quantity is expressed as an equality; otherwise, if ϵ is greater than zero, then the constraint is said to be approximate. A PAC summarizes the likelihood δ of a constraint being satisfied within tolerance ϵ as a *cumulative distribution function* (CDF). Unlike traditional ICs, which are time-invariant, PACs are adaptive and govern temporal localities. We can specify associated time windows over which to measure and apply each constraint. This could be based on either the landmark or sliding windows models; the window ω is specified along with the constraint expression.

For example, a constraint could enforce a regular rate that a network device is polled of every 5 minutes. However, if this constraint were to be enforced using traditional ICs then, in practice, a very small portion of the data would be allowable and would thus result in false dismissals. After all, interfaces are polled via UDP, which is prone to small and unpredictable delays. One can attempt to get around this by relaxing this IC, for example, by permitting polling intervals of 4-6 minutes. However, this may not be deemed acceptable as it would allow consistent 4-minute polls as well as alternating polling intervals of 4 and 6 minutes. Hence, we need a way to express *degrees* of constraint satisfaction, both in terms of magnitude and rate. We will give the PAC for this constraint in the following section.

We now describe three general classes of PACs (namely, domain constraints, functional dependencies, and unique key constraints) and give examples of each. These classes are inspired by (generalizations of) standard integrity constraints from a DBMS.

3.1 Domain PACs

A *domain constraint* requires that an attribute value must be drawn from a specific set or range. The PAC analogue is a straightforward generalization of this.

Definition 1 (Domain PAC) *Given a legal domain D on an attribute, a Domain PAC specifies that all attribute values x fall within ϵ of D with at least probability δ, that is,*

$$Pr(|D - x| \leq \epsilon) \geq \delta.$$

Since, in our model, the data arrives in temporal order, domain constraints can be defined over differentials in contiguous values via a trivial data transformation. We will give a few examples of the constraints in network database and show how they are expressed in traditional ICs and PACs. The advantages of PACs will become obvious.

A simple domain constraint of interest is that traffic should be bounded by the capacity of an individual link. Thus, if V_i is the traffic rate between $[t_{i-1}, t_i)$ and C is the total capacity of bytes that can be transferred during the time interval $[t_{i-1}, t_i)$, [2] the constraint is $\forall i, V_i \leq C$. This is an exact constraint. However, in network operations, one often overengineers the link and, hence, there is a designed-for capacity of the link $DC < C$. One typically wants the traffic level to be less than DC. This is an approximate constraint. The rule can be expressed as follows.

Rule 1 (Traffic Capacity) *The traffic rate is mostly less than a designed-for capacity DC.*

Here is an effort to express this in traditional IC:

Traditional IC 1

$$V_i \leq DC \quad \forall i$$

The problem of the traditional IC for expressing the constraint is that "mostly" in the rule is replaced by "always". To overcome the problem, we will propose PAC as follows. Two ways to formalize this are based on *instantaneous* or *windowed average* terms. For example, the instantaneous PACs is as follows:

PAC 1

$$Pr(\frac{V_i}{DC} \leq \epsilon) \geq \delta \tag{1}$$

[2] C is usually constant except when there is a change in the network topology (e.g., the link speed is upgraded).

Another way to formalize this is in terms of the sliding window average. The PAC for this is as follows:

PAC 1'

$$Pr(\frac{avg(V_j, w)}{DC} \leq \epsilon) \geq \delta$$

where $avg(V_j, w)$ is the moving average traffic rates.

Next we explore the *smoothness* property of the data. SNMP data aggregates flows from numerous sources, so it tends to have smooth properties in general.Data quality problems often lead to violations of such smoothness. So we explore this using ICs that are designed to capture the posited smoothness of SNMP traffic. One approach might be to bound the change in traffic between consecutive polls:

Rule 2 (Traffic Jumps) *The difference between consecutive traffic rates should be small most of the time.*

Let $\overline{V}$ be the current daily average traffic volume, this rule in IC and its PAC analogue are expressed as follows:

Traditional IC 2

$$\frac{|V_{i+1} - V_i|}{\overline{V}} \leq \mu \quad \forall i$$

where μ is a pre-defined number.

PAC 2

$$Pr(\frac{|V_{i+1} - V_i|}{\overline{V}} \leq \epsilon) \geq \delta \tag{2}$$

Note that the IC version is very difficult to express in SQL due to the sequential nature of it. Also, the IC version can't capture the probabilistic nature of the constrains as does the PAC version.

We can also measure the regularity or "smoothness" of the polling rate as we had discussed. This involves setting up a constraint over the time axis.

Rule 3 (Regular Polls) *The polling rate should be around a constant Δ.*

The ICs and PAC of this rule could be expressed as follows.

Traditional IC 3

$$|t_i - t_{i-1}| = \Delta$$

PAC 3

$$Pr(|(t_i - t_{i-1}) - \Delta| \leq \epsilon) \geq \delta \tag{3}$$

3.2 Functional Dependency PACs

A *functional dependency* $X \rightarrow Y$ enforces that two tuples must agree on the values in attributes Y if they agree in attributes X.

Definition 2 (Functional Dependency PAC) *A Functional Dependency PAC $X \rightarrow Y$ specifies that, if*

$$|T_i.A_\ell - T_j.A_\ell| \leq \Delta_\ell \quad \forall A_\ell \in X,$$

then

$$Pr\left(|T_i.B_\ell - T_j.B_\ell| \leq \epsilon_\ell\right) \geq \delta \quad \forall B_\ell \in Y.$$

We generalize the concept of functional dependency further by allowing for an aggregate f over a *set* of values in attributes X to functionally determine an aggregate g over a set of values in attributes Y. Notationally, we say $f(X) \rightarrow g(Y)$. Below we give two examples of Functional Dependency PACs.

Let e_1 and e_2 be the endpoints of a link. Let $V_{out}^{e_1}$ be the outbound traffic polled at interface e_1 and $V_{in}^{e_2}$ be the inbound traffic polled at the other endpoint e_2. If the link is functioning properly, we should have the following rule.

Rule 4 (Left-right Balances) *The outbound and inbound traffics at the two end points of a link should be roug hly the same.*

In this case, a traditional functional dependency could be set to ensure that the traffic volumes at both endpoints at the same timestamps are identical.

Traditional IC 4

$$V_{out}^{e_1}(t_i) = V_{in}^{e_2}(t_i)$$

However, the endpoints may be polled at different times and hence this constraint may be too strong. Let t_i be a timestamp at which e_1 is polled and let t_j be the closest timestamp to t_i at which e_2 is polled. We give the FD PAC as follows:

PAC 4

$$Pr(|V_{out}^{e_1}(t_i) - V_{in}^{e_2}(t_j)| \leq \epsilon) \geq \delta \tag{4}$$

For another example, consider a network traffic version of “Kirchoff’s Law”.

Rule 5 (In-out Balances) *The traffic flowing into a router should approximately equal that flowing out.*

Written as an FD, this would be $\sum_i T_i.V_{in} \rightarrow \sum_j T_j.V_{out}$.

Traditional IC 5

$$\sum_{in-links} V_{in}(t_i) = \sum_{out-links} V_{out}(t_i)$$

The above IC is too rigid for network database. To be more precise, this constraint should be expressed as a PAC:

PAC 5

$$Pr(|\sum_{in-links} V_{in} - \sum_{out-links} V_{out}| \leq \epsilon) \geq \delta \quad (5)$$

The sums are over traffic volumes close together in time from the same router. We can also specify a time window ω in which these sums are computed.

3.3 Unique Key PACs

A *unique key constraint* says that no two tuples may agree in their values for all of the attributes that constitute a key. The PAC analogue is a straightforward generalization of this.

Definition 3 (Unique Key PAC) *Let T be a table and A_ℓ be an attribute in T. A Unique Key PAC specifies it is unlikely that more than one tuple exists with approximately the same key:*

$$Pr(|T_i.A_\ell - T_j.A_\ell| \leq \epsilon_\ell) \leq \delta_\ell$$

for each attribute A_ℓ in the key.

Note that the ϵ_ℓ's and δ_ℓ's are independent of one another.

We can set up a unique key PAC to detect spurious traffic values whose values occur at suspiciously high rate.

Rule 6 (Spurious Values) *The same traffic value should not repeat too many times.*

Let t_i and t_j be consecutive polling times for an interface. A traditional unique key IC on `(time, V)` is as follows.

Traditional IC 6

$$V(t_i) \neq V(t_j)$$

This is infeasible because some instances of repeats may be valid (if unlikely). Hence, we use the following PAC to measure this likelihood:

PAC 6

$$Pr\left(\sum_{|V(t_i)=V(t_j)|} 1 \leq \epsilon\right) \geq \delta \quad (6)$$

4 Managing PACs: PACMAN

In this section, we describe the *PAC Manager* or the *PACMAN* that manages PACs in a SNMP database. It has a visual interface for the DBA or user to specify PACs. Each PAC has parameters ϵ and δ as we saw in the previous section with ϵ being the approximation to a constraint we allow and δ being the probability for each data item to satisfy the constraint with the given approximation.

PACMAN monitors the data in the database and performs two main tasks. First, it maintains a collection of PACs and picks thresholds δ^* and ϵ^* suitably. Second, it monitors new data in the database over time and *alarms* when data deviates from the PACs. It performs all the bookkeeping activities associated with the two tasks and manages all the PACs for the given database over time.

We will now discuss how PACMAN performs its two main tasks.

Initially, the list of PACs in PACMAN is empty. The user enters a set of rule-templates using a visual form whose parameters are to be instantiated as PACs over the existing training data. New PACs can be added at any time.

What PACs does PACMAN choose to maintain and how are thresholds chosen? One may rely on the user to specify both but with thousands of elements and PACs in the database, this is impractical. Hence, we rely on a variety of ways to automate this process.

PACMAN will fit the rule-template on different grouping partitions of the training data: on a per-interface basis, aggregated by interface class (eg, OC-48s), by day-of-week (eg, Mondays), by time-of-day (eg, evenings), etc. In total, we consider the following group-by lattices:

- topological: interface $\subset$ device pair $\subset$ device $\subset$ ALL;
- categorical: interface $\subset$ link speed class $\subset$ ALL;
- temporal: instant $\subset$ hour $\subset$ day $\subset$ week $\subset$ ALL; or
  instant $\subset$ time-of-day $\subset$ ALL; or
  instant $\subset$ day-of-week $\subset$ ALL.

Notice that the product of these hierarchies defines a lattice, and there is a well-define notion of going up and down this lattice to refine or coarsen the granularity in one or more of these attributes. Following factors are relevant in evaluating a PAC.

- **Usefulness:** The distribution must be skewed so that it is possible to tell the good from the bad. For example, the approximation distribution from a PAC may reveal that practically any ϵ-value will make δ small and always be violated, or that δ will be large and never be violated. (More formally, δ may be 0 or 1 except for trivial values of ϵ.) In these cases, the PAC is effectively useless.
- **Stability:** A distribution that is not stable cannot be used for a PAC rule. For example, the approximation distributions may differ substantially on each interface and on each day, which might render the PACs unstable for a given rule-template.

- **Granularity:** The distribution of a statistic dependents on the sampling space. The PACs can be derived on different grouping partitions and different granularity of the data. Suppose we have the current distribution of the data at certain granularity. If any distribution of a subpartition of the data is significantly different from the distribution, the conclusion drawn from the current distribution will not apply to that subgroup. One can have the grouping of the data in the finest granularity, but this will result in numerous distribution that are similar to each other. A better strategy is to merge these fine partitions of the data with similar distribution into coarser partitions recursively.

Currently, PACMAN works as follows. We use a month of data (so ALL in the hierarchy above in time attribute is a month) as training data, and learns thresholds and picks a set of PACs at suitable granularity in the lattice. This is used to monitor data henceforth. That is, learning and tuning may be thought of as being *offline* with sufficient resources to perform all the computations. In particular, we have the space to store a month of data and the computations on a month of data will take less than a month (this is trivial since we can now do this on a standard laptop for the entire SNMP database from a large ISP!). We will only describe algorithms for implementing this framework.

However a close look at the algorithms will quickly reveal that the algorithms we describe below can be implemented using recently discovered streaming algorithms in small space to a guaranteed approximation if needed, so this task can be performed by PACMAN in a dynamic, online manner. However, we do not explicitly explore the online learning of the PACs in this work. This is because of two reasons: first, online learning of thresholds often leads to overfitting of data to local effects and further research is needed to finely tune the learning process, and second, in our preliminary study, we have not yet found this to be very useful in our SNMP database. Henceforth, we will only discuss the offline learning framework.

Based on the above criteria, PACMAN will use some heuristic algorithms to learn and manage PACs. We will now describe how PACMAN chooses PACs and learns thresholds in some detail. The algorithms we use are simple; in fact, our focus has been on choosing straightforward algorithms that will work effectively in our setting, rather than design sophisticated algorithms that are more general than needed.

First consider how the thresholds for a specific PAC are learned. A specific PAC rule in a particular granularity is instantiated as an *Empirical Cumulative Distribution Function* (ECDF). We maintain the percentiles (or histograms) over the approximation errors e's to estimate their probabilities. That is, for a PAC rule, we maintain $\epsilon_1, \epsilon_2, ..., \epsilon_{99}$, where $Pr(e < \epsilon_i) \geq \frac{i}{100}$. In another word, the empirical probability that the approximation errors of the PAC rule are less than ϵ_i is more than $i\%$. Our first observation is that for the applications of PAC, only the tails in the ECDFs are useful. We care about only those points where the approximation errors are unacceptable. PACMAN can therefore maintain only the higher percentiles in the tails, say, $\epsilon_{90}, \epsilon_{91}, ..., \epsilon_{99}$. In general, the threshold (90 above) can be set based on the δ one considers the absolute lower bound on any reasonable threshold. PACMAN picks thresholds from ECDFs. Typical ECDFs have a natural "knee" or "sweet spot" where choosing (ϵ^*, δ^*) in various PACs is preferable. This is because increasing tolerance ϵ^* does not change δ^* significantly. The heuristic algorithm PACMAN uses to decide the sweet spot ϵ_s is based on finding the largest difference in the list $\epsilon_{90}, \epsilon_{91}, ..., \epsilon_{99}$. That is,

$$s = argmax_{90<i\leq 99}(\epsilon_i - \epsilon_{i-1}).$$

This simple rule turns out to be very effective in practice, as is confirmed in our experimental study in section 6. Notice that we could have actually taken the entire training data, sorted it and used the precise ECDF to make the threshold selection, but in all our experience, the simpler heuristic above of maintaining "equiwidth histogram" on the tail of the ECDF sufficed.

We also need algorithm to decide the stability of the PACs over time. This can be done in a number of ways dependent on comparing the ECDFs over different windows of a given granularity. Here, we will present a simple heuristic we use that prove adequate; more sophisticated comparison techniques can be used if necessary. PACMAN achieves this stability by monitoring the changes of the ECDFs. First we define the differences between two ECDFs $\xi_1 = (\epsilon_{90}, \epsilon_{91}, ..., \epsilon_{99})$ and $\xi_2 = (\hat{\epsilon}_{90}, \hat{\epsilon}_{91}, ..., \hat{\epsilon}_{99})$ to be their ∞-norm distance:

$$d(\xi_1, \xi_2) = max_{90\leq i\leq 99}|\epsilon_i - \hat{\epsilon}_i|.$$

The historical incremental differences between ECDFs for a particular PAC are stored as a vector $\vec{d}$. When a new ECDF is updated based on the new data, PACMAN checks the change in ECDFs, d_{new}. If d_{new} exceeds $avg(\vec{d}) + k \cdot std(\vec{d})$, then PACMAN knows that the ECDF is not stable, where $avg(\vec{d})$ and $std(\vec{d})$ are the average and standard deviation of $\vec{d}$ respectively, and k is a predefined parameter.

Given the definition of the distance between ECDFs above, we can also check if a group of ECDFs are close to each. That is, given a predefined parameter η, we say that a group of ECDFs $\xi_1, \xi_2, ..., \xi_k$ are close to each if the following holds:

$$\forall 1 \leq i, j \leq k, d(\xi_i, \xi_j) < \eta.$$

Combining the above algorithms, PACMAN can manage a large number of PAC rules automatically. PACMAN starts by constructing ECDF for each PAC rule in the finest granularity. Then, we perform a traversal up (and down) the granularity lattice. PACMAN will merge the ECDFs of the same PAC rule in the same granularity to a coarser granularity if they are close to each other. Also those ECDFs that are not stable will be separated into finer granularity to check whether stability can be achieved. Those PAC rules with unstable ECDFs will not be used. For PAC rules with stable ECDFs, the sweet spots will be computed and be set as their thresholds. In this manner, PACMAN automates the process of learning set of PACs and their thresholds for data quality monitoring.

The second major task that PACMAN performs is "alarming". The rationale is that the thresholds associated with PACs were chosen to be indicative of the data distribution, so any significant deviation from it is an occasion for "alarming". Alarming is most naturally based on the thresholds learnt. In the most general scenario, users may choose a pair (α, β)

of *alarming thresholds* where α and β are fractions. A PAC with thresholds (ϵ^*, δ^*) leads to an alarm if the number of tuples in the window at specified granularity of the PAC that violate the constraint with approximation $\alpha\epsilon^*$ is at least $(1 - \delta)/\beta$. Intuitively α and β are fractions that are allowances for alarming over the thresholds that represent the data distribution. In practice, one does not want to choice these factions for each of the PACs monitored by the PACMAN. In all our experience, it suffices to set $\alpha = 1$ by default because the approximation thresholds are fairly robust. In our experiments we study the effect of gradually changing β, but our experience is that setting β to be close to 0.5 is a good choice since often when violations occur, they occur in significant fraction of tuples. The online monitoring algorithm is trivial, since when a new item arrives, we can determine in $O(1)$ time if it contributes to violation or not with simple arithmetic to check the approximation thresholds.

5 Integrating PACMAN into a Database

As described so far, PACMAN is a database application. It interacts with the database in a straightforward manner. The underlined database (in our case, Daytona) handles the transactions and manages the SNMP data. PACMAN manages the satellite data (configuration files, routing details etc.) and perform the analysis. So, as such, PACMAN works like a data quality browser and monitor. In this section, we discuss how to integrate PACMAN into a database. There are several issues to consider that we describe below.

An important issue concerns specification of PACs. PACMAN currently supports a few standard templates for specifying the checks and balances via a graphical form to be filled in. This will do as an application. However, if it is integrated into a database, one needs a systematically designed language to specify the checks. Integrity constraints which are our underlying inspiration are either incorporated into SQL or as special purpose TRIGGERS. To exploit the full power of PACs, we need to carefully design an extension of TRIGGERS. We leave this issue open.

Another issue concerns violations. In classical context of ICs, a violation automatically leads to an exception, so the data item is not inserted into the database. However, automatic datafeeds such as network traffic data come at a great rate, and contain many data quality problems both on a regular basis as well as in transitory basis. It is unrealistic to fjord such a stream so that all updates pass data quality checks. Instead, a practical approach is to populate the database with the logs and postprocess the database appropriately. There are multiple ways to postprocess the database for addressing data quality problems concerns.

One approach is *data cleaning*, which is reasonable in many traditional settings [3]. But this is not always appropriate in our setting. For example, one suggestion is to extrapolate the missing values and populate the database with these premised values. However missing polls may be an indication of network anomaly and cleaning this out of the database greatly affects the accuracy of the data analysis for network management.

A different approach is to abstract the burden away from the user and to rewrite queries automatically. The database uses the data quality monitor such as PACMAN to under-

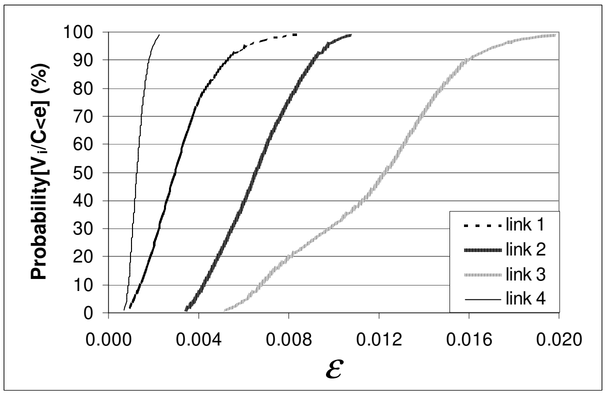

Figure 2: Distributions of traffic(PAC 1)

stand the data, and automatically rewrites users' queries into "equivalent" ones. This has an intuitive appeal. But it hides major conceptual and technical problems and is highly nontrivial.

The final approach is to involve the user in the loop. We use PACMAN to expose the data quality issues to the user through a visual interface. An informed user can pose database queries more carefully. This may seem like punting the problem, and putting the entire burden on the user. But we have routinely found that this works. Let us give an example. Say a user wants to determine the correlation between the traffic at two links on a given day. The user who is made aware of the fact that there are several missing polls may modify the query as follows: *compute the correlation only restricted to observations which were made in the system.* This approach is typically not a favorite in academic research, but it is successful in real systems. Engaging a knowledgeable and thoughtful user is facilitated by data quality monitors such as PACMAN. This is what inspired us to develop PACMAN as a database application, merely monitoring data quality and reporting alarms.

6 Experiments

We applied our PAC approach to a network traffic database of a large ISP. Our system is fully operational over more than 6 months of data. Here we just show a few proof-of-concept examples of PACs on real data.

In the first experiment, we examined the ECDFs of several PACs. Figure 2 shows the distributions of PAC 1 (traffic rate) over all days in June 2002 for four randomly selected links. We observed that different links have different distributions. That is because each link has its own designed-for capacity; hence, obviously, a PAC trained over the aggregate of *all* links would not be very meaningful. Same phenomena holds for PAC 2 (smoothness) as well as shown in Figure 3 for the same 4 links. On the other hand, we observed the ECDFs of different links to be roughly the same for PAC 3 (polling intervals), as shown in Figure 4.

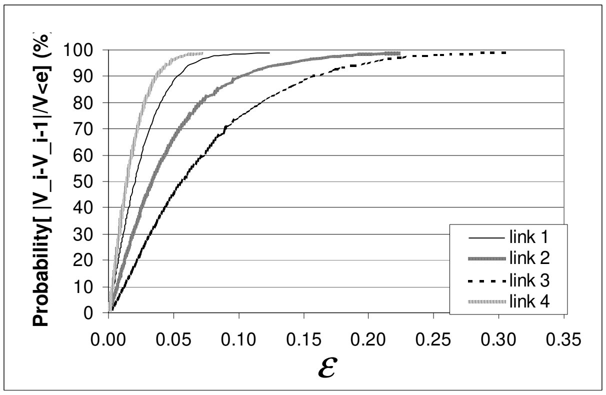

Figure 3: Distribution of the traffic jumps (PAC 2)

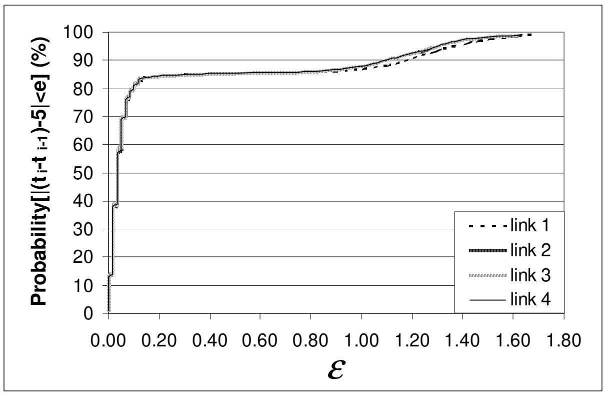

Figure 4: Distribution of the polling rates(PAC 3)

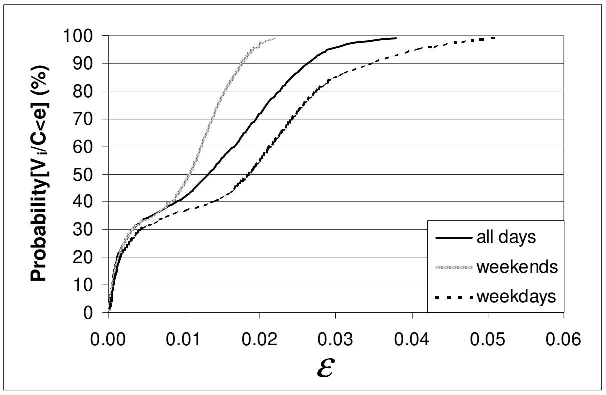

Figure 5: PAC 1 of a particular link based on different temporal granularities

This makes it unnecessary to maintain a separate distribution for each link for this PAC. These examples illustrate the importance of automatic granularity selection in PACMAN.

Figure 2 also shows a noticeable "sweet spot" in the ECDF, with a few outliers spread over a big range larger than ϵ^*. This characteristic, useful for choosing parameters automatically by PACMAN, is evident throughout the dataset.

While the above example shows that PAC 1 must be trained on a per-link basis, the temporal granularity can also affect PACs. Let us consider a link on which PACMAN raised alarms (with $\alpha = \beta = 1$) on a certain Sunday in July after training PAC 1 on the data from June. Figure 5 shows the ECDFs of PAC 1 with the training data from an individual link in June 2002. We consider two possible temporal granularities for training this PAC: grouping all days together or grouping the days based on weekdays and weekends. We can see that at the finer granularity level, the distribution on weekdays is very different from that on weekends. The distribution based on the coarser granularity level (all-days) averages the finer distributions together; as a result, when using it to detect data quality per day, it is not effective enough to discover data quality problems. Fortunately, PACMAN chose to maintain the ECDFs in the finer granularity based on heuristics discussed in Section 4. Given the value of ϵ^* at $\delta^* = 5\%$, the percentage of tuples that are above ϵ^* is 47% based on the weekend granularity, but only 2.8% based on the all-days granularity; hence, data quality problems may not be detected if the granularity of PAC is not partitioned into weekdays and weekends. (In this case, the data quality problem was that the link was upgraded in capacity without prompt change to the configuration tables.)

One way to examine the effectiveness of PAC is by the number of alarms generated in the testing data (with $\alpha = \beta = 1$). PACMAN trained PAC 2(Traffic Jumps) on all links using the data in June. Each link has its own ECDF. PACMAN then applied these ECDFs to monitor the traffic in a certain day of July. Figure 6 shows the percentages of tuples that raise alarms on PAC 2 for each link. The percentages are mostly below 10%. For those few ouliers with percentages above 20%, they deserved the attention of the DBA for further examining of data quality problems. In other words, β of 0.5 seemed like a case for further

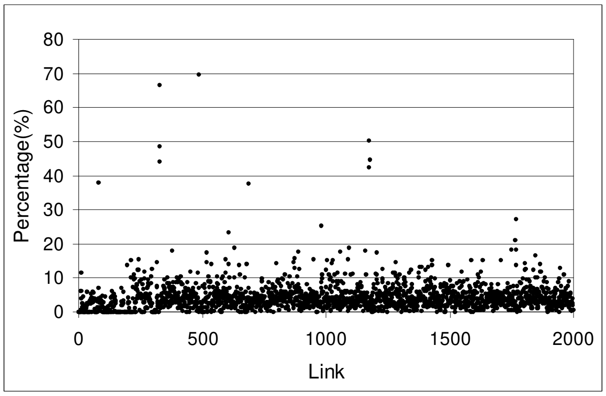

Figure 6: Percentages of violations of PAC 2 for all links in a certain day

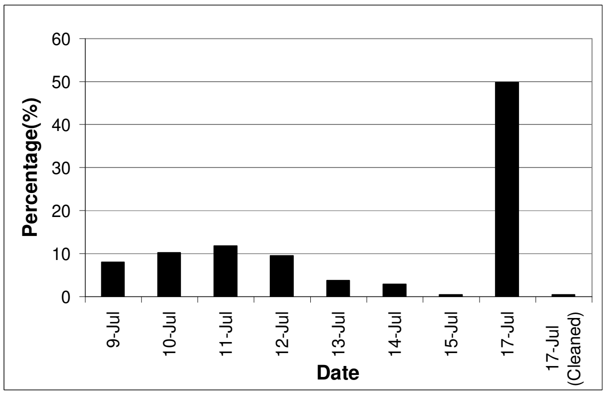

Figure 7: Percentages of violations of PAC 3 for a certain link in a certain week

examination. In general, the number of alarms generated by PACMAN is reasonable for human attention. Our close examination of those ouliers revealed that they actually had data quality problems. This example also demonstrates the stability of the distributions of PACs, which makes it possible to apply the ECDFs learned from the old data to check the PACs of the new data.

The following case study is to do with the detection of double pollings. In July 17th, 2002, an alarm was raised for PAC 3 on a link B (with $\alpha = \beta = 1$). Figure 7 shows the percentage of tuples that were above the threshold ϵ^* for that day and the previous week. We can see that there is a clear spike in the percentage in that day. A possible reason for double polling is that two different links are polled at the same time due to mislabeling. As a naive way to clean the data, we treated odd and even polls as if they came from different pollers. In this figure we can see that the percentage returns to normal after we clean the data. Indeed when we looked at timestamps on these polls, we confirmed that there were two interleaved polls *each* roughly 5 minutes apart.

In summary, the distributions of statistics for the network traffic data seen as a ECDF, each has its own sweet spot. The granularity affects the distributions. These distributions are mostly stable over time. These characteristics of the ECDFs allow PACMAN to mine PAC rules for different complicated traffic behaviors from real world data. Unlike traditional integrity constraint, the PAC-base approach is very effective in detecting data quality problem for network traffic database.

7 Related Work

Several tools exist and are widely used for monitoring network traffic such as RRDTool [7]; these tools provide graphical plots of network traffic, but put the burden on the user to detect data quality problems. Recently there has been some work in the networking literature on automatically identifying anomalies in aggregated traffic measurements using wavelets [1]; but they address specific quality problems in the network database context, and there is no underlying mechanism for detecting data quality problems in general.

There is a large body of work in the statistics literature dedicated to defining and evaluating data quality (e.g., see [14]): outlier detection, imputation of missing values, etc. Mostly, these too tend to be focused on developing efficient, statistically grounded methods for detecting particular data quality problems.

A more principled approach for enforcing data quality in general in large scale data has been taken in the database literature. (There are a number of results in the database literature too that focus on detecting specific data quality problems; we do not survey them here because they are not directly relevant to our PAC approach here.) Traditionally, integrity constraints (ICs) and triggers are used to define deterministic constraints *a priori* as opposed to mining existing, non-deterministic rules from the data. Ad-hoc approaches to approximate functional dependencies have been defined in [6, 9, 8, 11]. Our work here is more general, abstracting from a variety of ICs, and employing a probabilistic, approximate framework in a systematic manner, to detect user-directed constraints and violations.

Recently, there has been work in providing various *browsers* for quality mining and cleaning. For example, Bellman [4] is a data quality browser that helps identify fields with similar values, estimate join sizes, tracing join paths, etc. This is useful in a database with many tables, but it does not address most of the crucial data quality problems that arise in network database context summarized in Section 2.3. Also, no principled approach akin to PACs is evident in Bellman.

The related problem of data cleaning has been studied extensively in the literature for several specific data quality problems (mapping schemas, finding approximate matches, etc.). For excellent surveys, see [12, 3]; for additional references, see [10, 2]. Potter's Wheel [13], like our work here, makes the observation that data discrepancies are best dealt with using extensible, domain-specific techniques, but addresses the problem of cleaning data rather than discovering data quality violations. Ajax [5] uses a language to express data cleaning specifications declaratively. The important problems of how to express general integrity constraints and how to automatically discover such constraints were missing in these works.

Our flexible framework of probabilistic ICs can provide a solid base for other data cleaning tools.

Finally, our underlying approach to use PACs (that is, to introduce tolerance and confidence parameters into ICs) is a generic approach in data mining and statistical data analysis. However, we are not aware of any application of this approach to address data quality problems.

8 Concluding Remarks

We have focused on network database of aggregated IP traffic data, gathered using the SNMP protocol, which is an integral part of daily operations of today's major ISPs. A fundamental problem in such databases is one of data quality which is poor due to many network-specific issues. We have proposed a novel, principled approach to detecting and monitoring such data quality problems. Our proposal is to use probabilistic, approximate rules inspired by integrity constraints (PACs). These are specified as user-defined templates with tolerances and probability thresholds. We use the statistical properties of the data to learn the parameters. PACMAN manages the PACs.

We have built the PACMAN-based system for a large ISP and it is operational on several months of real data. We use this system to perform extensive experiments and show that our PAC-based approach is highly effective for detecting and monitoring data quality problems. PACs are general, powerful, logical constructs to verbalize users' insight into structural properties of the data (much as ICs are intended to be, but PACs are more flexible and powerful); hence, we believe they will prove fundamental within the general, database context beyond their effectiveness in addressing data quality problems within network databases.

References

[1] P. Barford, P. Kline, D. Plnka, and A. Ron. A signal analysis of network traffic anomalies. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, 2002.

[2] Data Cleaning Bibliography. http://epoch.cs.berkeley.edu:8000 / rshankar/pwheel/bibliography.html.

[3] T. Dasu and T. Johnson. Problems, solutions and research in data quality. In *SIAM International Conference on Data Mining*, 2002.

[4] Tamraparni Dasu, Theodore Johnson, S. Muthukrishnan, and Vladislav Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *SIGMOD 2002, Proceedings ACM SIGMOD International Conference on Management of Data*, 2002.

[5] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.-A. Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB 2001*, 2001.

[6] R. Haux and U. Eckert. Nondeterministic dependencies in relations: An extention of the concept of functional dependency. *Information Systems*, 10(2):139–148, 1985.

[7] RRDtool homepage. http://www.caida.org/tools /utilities/rrdtool/.

[8] Wen-Chi Hou. Extraction and applications of statistical relationships in relational databases. *TKDE*, 8(6):939–945, 1996.

[9] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 392–401. IEEE Computer Society, 1998.

[10] Research in Data Quality Web Site. http://www.dataquality-research.com.

[11] H. Mannila and K. Raiha. Algorithms for inferring functional dependencies from relations. *Data and Knowledge Engineering*, 12(1):83–99, 1994.

[12] E. Rahm and H.H. Do. Data cleaning: Problems and current approaches. In *Data Engineering Bulletin 23(4)*, pages 3–13, 2000.

[13] Vijayshankar Raman and Joseph M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *The VLDB Journal*, pages 381–390, 2001.

[14] T Redman. *Data Quality: Managemetn and Technology*. Bantam Books, 1992.