Keeping Track of 70,000+ Servers: The Akamai Query System

Jeff Cohen
Akamai Technologies
Cambridge, MA 02142
jecohen@akamai.com

Thomas Repantis
Akamai Technologies
Cambridge, MA 02142
trepanti@akamai.com

Sean McDermott
Akamai Technologies
Cambridge, MA 02142
sean@akamai.com

Scott Smith
Formerly of Akamai Technologies
Cambridge, MA 02142
scott@clustrix.com

Joel Wein
Akamai Technologies
Cambridge, MA 02142
jwein@akamai.com

Abstract

The Akamai platform is a network of over 73,000 servers supporting numerous web infrastructure services including the distribution of static and dynamic HTTP content, delivery of live and on-demand streaming media, high-availability storage, accelerated web applications, and intelligent routing. The maintenance of such a network requires significant monitoring infrastructure to enable detailed understanding of its state at all times. For that purpose, Akamai has developed and uses *Query*, a distributed monitoring system in which all Akamai machines participate. Query collects data at the edges of the Internet and aggregates it at several hundred places to be used to answer SQL queries about the state of the Akamai network. We explain the design of Query, outline some of its critical features, discuss who some of its users are and what Query allows them to do, and explain how Query scales to meet demand as the Akamai network grows.

1 Introduction

Akamai's edge network is a distributed computing platform with over 73,000 servers in 70 countries in about 1,000 autonomous systems, which on any given day may handle upwards of 20% of Internet traffic. Akamai provides multiple services including the delivery of static and dynamic HTTP content and live and on-demand media streams, reliable storage, Web and IP application acceleration, and DNS services; see [15] for a recent overview. Each Akamai server runs multiple applications, constructed out of multiple components, and potentially participates in providing more than one of these services. Thus, Akamai's edge platform consists of over 1 million distributed software components.

The Akamai network supports customer businesses that run twenty-four hours a day, seven days a week. In many cases outages of even a short period of time can cause substantial business impact. The need for reliable real-time monitoring of the state of our network, therefore, is critical.

Query is a near real-time monitoring system, developed in-house, that monitors the Akamai network to provide up-to-date information about its state. It is used by automated applications to detect problems and measure performance over time, by software engineers to ensure their systems are behaving properly in the field, by operations staff to troubleshoot problems and ensure that the network is properly configured, and by services that provide data to customers. Information from Query is provided through a SQL interface, allowing users a familiar, precise way of specifying the information they need.

The Akamai network is divided into several thousand *clusters* all over the world at the edges of the Internet. It is in those clusters that Query begins collecting data. Every Akamai machine runs Query, and any software component on any machine can send data to the local Query instance to be published into database tables. Some subset of the machines in each cluster are designated as *Cluster Proxies* who also have the job of collecting all the data from their respective clusters. Each Cluster Proxy takes all the tables it receives from machines in its cluster and combines them into larger tables.

Query is partly distributed and partly centralized. The collection of data in thousands of clusters all over the world is fully distributed, but that data need to be aggregated to allow the issuing of SQL queries about the entire Akamai network. A set of a few hundred machines, called *Top-Level Aggregators (TLAs)* collects data from the cluster proxies and combines data from all the clusters into larger tables. Because it takes all the resources available to most TLAs just to talk to all those clusters and combine their data, TLAs don't have enough processing time left to also answer queries. Therefore they send their aggregated tables to *SQL parsers* that actually receive queries and compute their answers.

Several types of users make requests to Query. Hu-

man users, including software engineers and operations staff, issue queries to understand the state of the Akamai network. This is particularly important for detecting and responding to problems quickly. This monitoring and diagnosing is facilitated by the fact that Query provides aggregated data in the form of tables that can be accessed using a familiar SQL interface. This interface enables users to easily combine data from multiple real-time data sources, as well as statically generated configuration data, without the need to log in to individual machines. For example, by issuing a query such as the one below, a user can see processes on machines with role "dns" that are using more than 75% of system memory for their RSS:

```
SELECT sys.ip ip, procname, rss, pid
FROM   sys, processes
WHERE  sys.ip = processes.ip
  AND  (rss*100)/sys.memtotal > 75
  AND  sys.ip in
       (SELECT ip
        FROM machinerole
        WHERE role='dns');
```

Numerous automated applications issue queries as well. For example, Akamai's alert system is an important tool for detecting problems and fixing them before they affect customers. It issues queries to detect each of several thousand conditions that indicate problems, then alerts staff in the Network Operations Control Center whenever those conditions are present.

A third group of users is customer-facing applications. For example, EdgeControl [3], the Akamai customer portal, provides graphs of usage to each customer. The data presented fall under two categories. The most reliable usage data are collected from detailed logs on the machines and displayed precisely. Query, however, can report results faster than the logs can be processed, but with less than perfect reliability. We display to customers the most recent data based on results from Query, and the most accurate data based on log analysis. Similarly, graphs such as the ones that are available to the general public on the Akamai website [22] depend on data collected from Query. We will discuss how each of the groups mentioned above uses Query and the benefits each gains from it.

The rest of this paper is structured as follows: Section 2 talks about the goals of Query's design. Section 3 talks about the architecture of Query that achieves these goals. Section 4 explains several of Query's features that are most important to users, and Section 5 details who some of those users are and how they use Query. In Section 6, we present techniques that have allowed Query to scale as the company has grown far beyond its size when Query was first written. Along with those techniques we use a number of other techniques to manage Query and make sure that it is provisioned and configured as needed, detailed in Section 7. In any large deployed network, failures are bound to occur, so we explain how Query handles them in Section 8. Finally, we compare query against related systems in Section 9, before concluding in Section 10.

2 Design Goals

Query is designed with a number of goals in mind. Occasionally, those goals conflict, providing us with difficult tradeoffs. We describe those goals and some resulting tradeoffs.

2.1 Goals

- **Reliability:** Query should always be available to answer requests.
- **Scalability:** Query should continue to stand as the load doubles several times over.
- **Data latency:** When data are published at the edge, they should appear in the answers to queries promptly.
- **Query latency:** When a user issues a query, an answer should come back quickly.
- **Completeness:** All published data should be available. Query results should be based only on complete tables.
- **Consistency:** When data are published, they should eventually be available everywhere. Requests served by distinct machines should have similar answers.
- **Synchronization:** All data available on a machine should be up-to-date as of about the same time, so that all tables from that machine reflect the state at one moment as closely as possible.
- **Fault tolerance:** When a machine fails or a connection goes down, the system should still be available to serve requests.
- **Fault quarantining:** A fault in one place should stay in that place instead of spreading.

2.2 Tradeoffs

Some of the aforementioned goals sometimes conflict. Here we describe some of the more interesting tradeoffs we face in the design of Query.

2.2.1 Data Latency and Completeness

To have complete data, Query must wait for every machine to send its contributions to every table before putting each table together. To have low data latency, Query must put its tables together quickly, waiting for as few things as possible. We achieve a balance between the two by providing a relaxed notion of best effort completeness, which will be discussed in Section 4.2.

2.2.2 Fault Tolerance and Completeness

Fault tolerance requires Query to move on and work around machines that fail. Completeness requires it to find a way to obtain their data. We strike a balance between the two with the same relaxed notion of completeness we describe in Section 4.2.

2.2.3 Fault Tolerance and Quarantining

A desire for fault tolerance suggests that when a machine fails, we should move requests to it to another machine. A desire for limiting the scope of faults suggests that, because a request could consume a large number of resources and take down a machine, we should not move requests that fail to another machine. We achieve a balance by having sets of a few equivalent machines called *aggregator sets* among which requests can move. A bad request may take down two or three machines in one aggregator set, but it will not take down the hundreds of aggregators system-wide or any machines that serve customer data. We are also very careful with aggregator sets used for critical data so that they do not get requests that consume more resources than they can afford. We describe aggregator sets in more detail in Section 3.5.

3 Architecture

We explain the architecture of Query by tracing the path data take from the time they are published to the time users see them affect the answers to queries.

3.1 Query at the Edge

Every machine on the Akamai platform runs an instance of Query. That instance listens for communications from processes on the same machine. Any process may open a connection to Query, after which point it is required to send Query a list of tables it wants to publish. Query does not start collecting these tables immediately, however. Some of them are very rarely used, and collecting them all preemptively would be a waste of resources. Instead, each Query instance maintains a list of tables that have been requested from it and requests from each process on the machine only those tables it needs.

Every once in a while (once a minute or two, depending on the machine configuration), every process is obligated to send Query a copy of all tables that process has claimed to be publishing that Query has requested. At the same frequency, but offset by several seconds, Query combines the tables being published by all processes on that machine. We call the set of tables a machine combines together a *generation*. The reason for the offset is so that the other processes have time to publish the data before Query consumes them. When Query prepares its generation, all the data were collected within a relatively short time span (several seconds), so the data provided by any individual edge machine come close to reflecting its state at one moment.

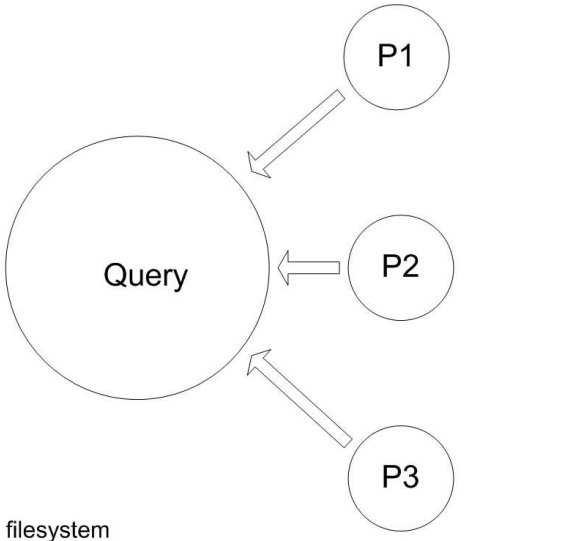

Figure 1: Query on an edge machine. Three processes, $P1$, $P2$, and $P3$, are shown publishing into Query, as is one example table.

There are several interfaces to Query used to publish on the edge machines. The most basic is a programmatic C interface that handles all the communication with Query. Wrappers around that interface exist in several other languages. Users also have the option of writing a file containing the values in their table in a text-based format. A daemon on the machines reads those files periodically and publishes their contents into Query. Finally, a separate software component enables Query to collect data published by SNMP-enabled devices, such as routers or filers.

A picture of an edge machine is shown in Figure 1. A single Query process and three publishing processes are shown, as is one row of an example table. That row describes information about one mount point on the ma-

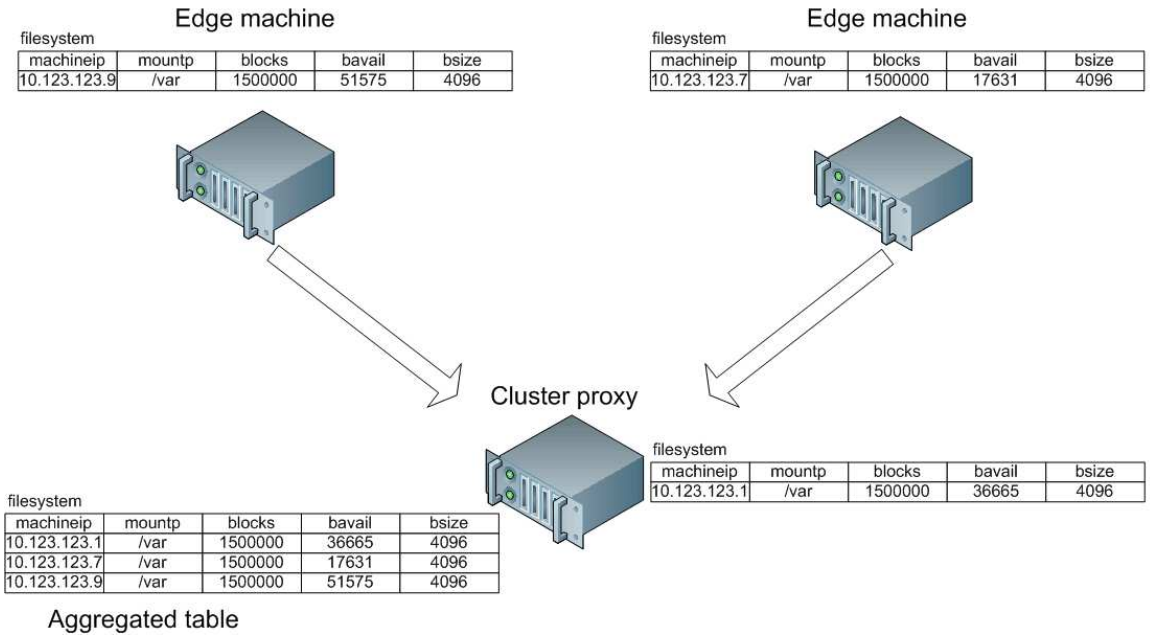

Figure 2: Query in a cluster. Two edge machines and a Cluster Proxy are publishing tables, which the Cluster Proxy aggregates. One example table is shown.

chine. In reality, that table has multiple rows per edge machine. Also, there are really several times as many publishing processes and hundreds of different tables available on each machine.

3.2 Cluster Proxies

The collection of data by Query is hierarchical. The Akamai network is divided into clusters all over the world, each located within a single data center. Within each cluster, some number of machines are designated *Cluster Proxies* and have the job of collecting data from all machines in the cluster. Each cluster is small, having at most a few dozen machines, so data collection does not incur a high overhead.

Each Cluster Proxy collects requests from the next level down the hierarchy and requests tables from each machine in its cluster. Every time any Query process collects a generation, it sends each Cluster Proxy a copy of all tables the Cluster Proxy is requesting. Any time Query sends a generation of tables from one machine to another, it sends it in an efficient encoded format to save bandwidth. Offset by several seconds from that process, the Cluster Proxies collect their own generations containing all the data from their entire respective Clusters.

The Cluster Proxies also serve as edge machines, so they also publish their own data, which they combine with the data from other machines in the cluster when making their generations.

A picture of Query in a cluster is shown in Figure 2. Only two edge machines, a cluster proxy, and one table are shown. In practice, a cluster would have up to dozens of machines, several cluster proxies, and hundreds of tables. The rows from all the edge machines are combined at the Cluster Proxy.

3.3 Aggregators

The next level in the hierarchy is the *Top-Level Aggregators* (TLAs). Each TLA has a complete view of the network, because it talks to a Cluster Proxy in each cluster. The job of a TLA is to collect generations from the Cluster Proxies, aggregate together global generations of all the tables from everywhere, and provide those global generations to other machines that will use them to answer SQL queries. We don't have the TLAs answer queries because it takes all the resources they have just to aggregate the generations.

TLAs collect generations of data from all Cluster Proxies in much the same fashion that Cluster Proxies do from machines in their clusters. TLAs collect their generations once every one or two minutes. Because we are interested in data about all Akamai machines, including TLAs, each TLA also publishes into Query. It collects its own information and sends it to all other interested TLAs. Each generation a TLA makes can include, in addition to data from the Cluster Proxies, data from itself and other TLAs.

3.4 SQL Parsers

A *SQL parser* is a machine that receives generations of tables from a TLA, receives the text of SQL queries from clients, computes the answers to the queries, and sends back the results. If a SQL parser has all the tables it needs to answer a query, it does so immediately. Otherwise, it sends a request to the TLA and waits for the TLA to send back a generation that contains those tables. To provide results with data collected at about the same time, all the tables used to answer a query are required to be from a single generation.

Just as Cluster Proxies and TLAs publish into Query, so do SQL parsers. When TLAs collect their generations, they can also include data from SQL parsers.

A picture of the Query system is shown in Figure 3. The cloud represents all the thousands of clusters talking to the TLA. The TLA shown is currently providing tables to two SQL parsers. There is a user at a terminal issuing a query against the table published in Figure 1 and Figure 2 to figure out which machines on the network have less than 3% of space available on some mount point. There are actually hundreds of TLAs and SQL parsers, but only one TLA and two SQL parsers are shown.

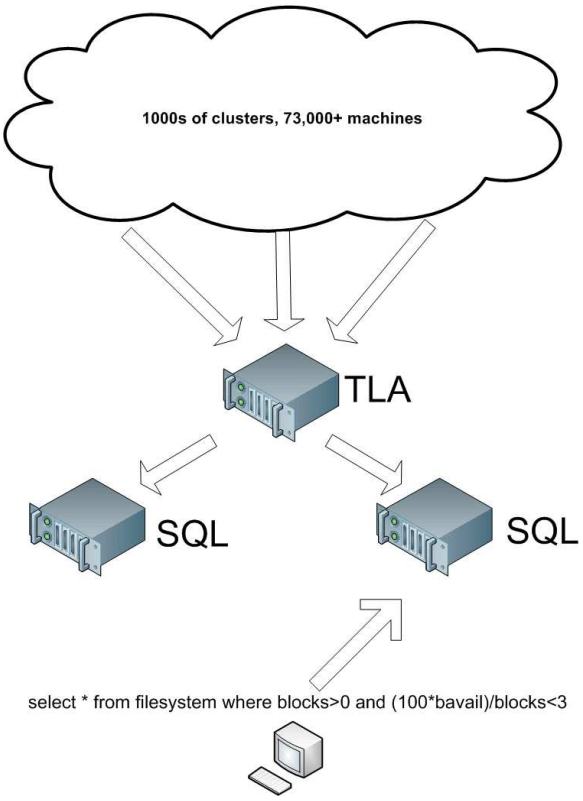

Figure 3: The Query system. The cloud is the whole Akamai network. Also shown are a TLA, the two SQL parsers getting tables from it, and a user at a terminal issuing a query.

3.5 Aggregator Sets

Not all queries are interested in data from the whole network, so not all TLAs talk to the whole network. For example, some queries' sole purpose is to monitor the health of the TLAs and SQL parsers. Those queries can get sent to machines that contain only data from the machines they are interested in. Each TLA can be configured to talk to only a subset of the network, and each SQL parser can be configured to talk to only a certain set of TLAs. We call the subset a TLA talks to its *span*. Because a SQL parser can get exactly the same data its TLAs can get, the span of a SQL parser is the same as the span of its TLAs.

Different users have different needs. For some users, latency is critical, and they need to issue queries to machines that are lightly loaded so that they never have to wait for a machine to collect a large generation before it can compute the answers. Other users need to join so much data that the issuing of their queries alone will make a machine heavily loaded.

We handle these disparate needs by dividing clients who issue queries into groups and giving each group some set of aggregators. We call the group to which a TLA or SQL parser is assigned that machine's *domain*.

Any time a set of TLAs or SQL parsers share a span and domain, the machines of each type in that set are performing the same job and are interchangeable. We call such a set of TLAs and SQL parsers sharing a span and domain an *aggregator set*.

3.6 Combined TLA-SQLs

Some aggregator sets are under light enough load that one machine can actually do all the work of a TLA and all the work of a SQL parser for that set. We configure those sets to do just that, to reduce our machine count and our costs. We call a machine doing the work of a TLA and a SQL parser a *TLA/SQL*.

3.7 Overall Network Distribution

Currently, Akamai has several hundred TLAs, SQL parsers, and TLA/SQLs divided into several dozen aggregator sets. Each aggregator set has at least three tuples of TLAs and SQLs for fault tolerance, and often many more, depending on its load.

4 Features

In this section we elucidate several of Query's features and explain how they empower its users.

4.1 Near Real-Time View

Query makes a new generation at each machine every minute or two, so the data at the edges of the network are at most two minutes old. Seconds go by between the collection of those generations and the aggregation of generations at the Cluster Proxies. Seconds more pass before aggregation begins at a TLA, a process which takes tens of seconds on the TLAs with the heaviest load. Encoding a generation to send to a SQL parser and decoding it at the SQL parser each take tens of seconds. Consequently, data at the SQL parsers can be a few minutes old.

Compared to the amount of time it may take for a human to diagnose and respond to a problem, a few minutes is not much. However, because Query's work is the first step in detecting and understanding a problem, a minute of time spent before data get into the results of queries represents a minute delay in the rest of the response process. Therefore, even though the latency of data is fairly low, continuing to lower it remains a priority.

Query's reliability is not perfect (see Section 8). Consequently, data in Query cannot be relied upon for certain things. Nevertheless, it is one of our fastest means of getting information, and sometimes we need fresh data, even if they are imperfect. In such situations, we use Query.

4.2 Synchronization

All data collected from a machine are collected within the span of several seconds. Although a TLA may have data collected potentially minutes apart from two different edge machines, its data from any one machine were published at about the same time. This condition is weak enough that we can achieve it without much overhead, but strong enough to provide some valuable abilities to the company.

The low variance in age of data from a given machine means we don't miss multiple related conditions, or the correlations among them. For example, suppose some rare event lead to the consumption of a large amount of memory. When one datum is present, the other will be as well, allowing us to detect such correlations.

4.3 Historical View

Query is used not just to understand the state of the network now, but also how it has changed. Query can be used to record prior data to get a view of past partial states of the Akamai network.

We have two means of doing this. The first is Akamai's historical reporting system, which will be described in Section 5. This system stores the results of queries for a long time and displays them in graphs, giving us a visual representation of how data in Query have changed. The second is Query History, a feature whereby aggregators can be configured to store old generations' copies of certain tables, load them, and answer queries based on them.

The ability to get a historical view has tremendous power. It allows us to see how usage patterns have changed over time, predicting future growth in usage based on past trends. It allows us to correlate changes in multiple parameters, so that we can know how much CPU is consumed by additional end user requests, how much memory, how much bandwidth, etc. If we detect a problem after it has existed for a while (say due to a software bug causing occasional spikes in the usage of some resource), we can figure out when that condition started to exist, helping us narrow down the cause.

4.4 Static Tables

Some tables don't change often and have contents that should be dictated by the structure of the network or some sort of unchanging information. There is no reason to spend resources to aggregate such tables through the normal Query system at the cluster level. Instead, we store tables in text files on the disks of TLAs, and we store index files describing where those files reside. Each TLA reads its static data off of its disk, adds it to the data it has from the Cluster Proxies, and re-reads the data any time they change.

Below is an example of a query that joins normally published data with static data. It looks at three tables: (1) load_info, which has information about all requests Akamai is currently handling; (2) region_data, which describes data about the geographical regions our machines are in; and (3) continent_data, which describes information about the seven continents. The query computes how many hits we're serving on each continent per second.

```
SELECT   c.continent_name,
         SUM(l.hits) hits
FROM     load_info l,
         region_data r,
         continent_data c
WHERE    l.georegion=r.id AND
         r.continent=c.continent
GROUP BY c.continent_name
ORDER BY hits DESC;
```

```
c.continent_name       hits
---------------  ---------
   North America  4,620,551
          Europe  3,392,102
   South America    655,175
            Asia    552,258
          Africa    106,781
         Oceania     39,905
      Antarctica        135
```

A query similar to this one is used to generate one of the graphs Akamai displays on its web site [4]. The numbers of hits and even the ordering of continents change throughout a typical day. That data, for example, were collected at about 3:15 PM Eastern Standard Time, when one would expect most of the Americas and Europe to be awake, but most of Asia and Australia to be asleep.

5 Applications

We now explain several of the key uses of Query and how they empower operations staff at Akamai.

5.1 Alert System

Akamai's alert system is the primary tool for detecting problematic conditions on the Akamai network. Engineers and operations staff can easily develop and activate alerts by writing SQL statements which are submitted to the Query system at regular intervals. For example, consider this simplified SQL statement to detect disks with less than 3% of their disk space left free:

```
SELECT
    machineip     ip_key,
    mountp       mnt_key,
    bavail*bsize free_space,
    (100*bavail)/blocks pct
FROM
    filesystem a
WHERE
    blocks > 0 and
    (100*bavail)/blocks < 3;

   ip_key    mnt_key  free_space  pct
-----------  ----  -----------  --
10.123.123.1  /var  150,179,840   2
10.123.123.7  /var   72,216,576   1
```

The SQL statement along with many other configurable settings form an *alert definition*. Each row returned by the SQL statement constitutes a problematic condition, or an *alert instance*. Each time the alert query is run, the result is compared to the previous result. Any new rows are considered new instances of the alert. As soon as an alert instance is detected, the alert is said to *fire*. If any rows from the previous iteration are no longer present, the alert is said to *clear*.

Akamai has found it important to tune when alerts fire and clear. For example, when writing a "High CPU usage" alert for a critical server, we may want to fire an alert when CPU is over 98% usage. A single spike to 98% isn't interesting but if we check every 2 minutes and the CPU is still greater than 98% after 15 iterations, then there is clearly a more chronic condition worthy of investigation. On the other hand, when writing a "Disk showing SCSI errors" alert, we would want to ensure the alert stays active even if the underlying disk errors do not repeat. This gives time for the operations staff to react to the alert and investigate the condition further.

As a result, three commonly used alert definition settings deal with these timing parameters:

- Frequency of SQL execution (typically one minute).
- Number of iterations the data are present before an alert fires.
- Amount of time the data must be absent before an alert clears.

When an alert fires, the alert system can be configured to do one of two things. It can alert staff in the 24/7 Network Operations Control Center (NOCC), which is done for urgent matters, or it can send an e-mail to engineering or operations staff for later follow-up. In the former case, the NOCC staff can take appropriate action using a custom user interface shown in Figure 4. The user interface combines the alert details with corresponding procedure, network access and ticketing. In many cases, alert procedure steps include analysis using further Query data. The NOCC can routinely handle over 10,000 new alert instances in a single day with this approach, coming from over 73,000 machines. (That figure includes problems on partner networks, and problems that the Akamai mapping system can automatically route around.)

At present, there are several thousand queries that run to detect alert conditions, with multiple thousands running every minute. The alert SQL queries are typically much longer than the example queries above, sometimes with pages of complicated SQL logic. Using techniques such as the ones we describe in Section 6, we have allowed a few tens of TLAs and SQL parsers to handle all of this load.

The alert system and Query provide several advantages for incident detection and response. If an operations staff member begins to suspect a problem and wants to create a query to detect it, that person can create a query in a matter of minutes, start testing it immediately to make sure it produces the desired results and doesn't

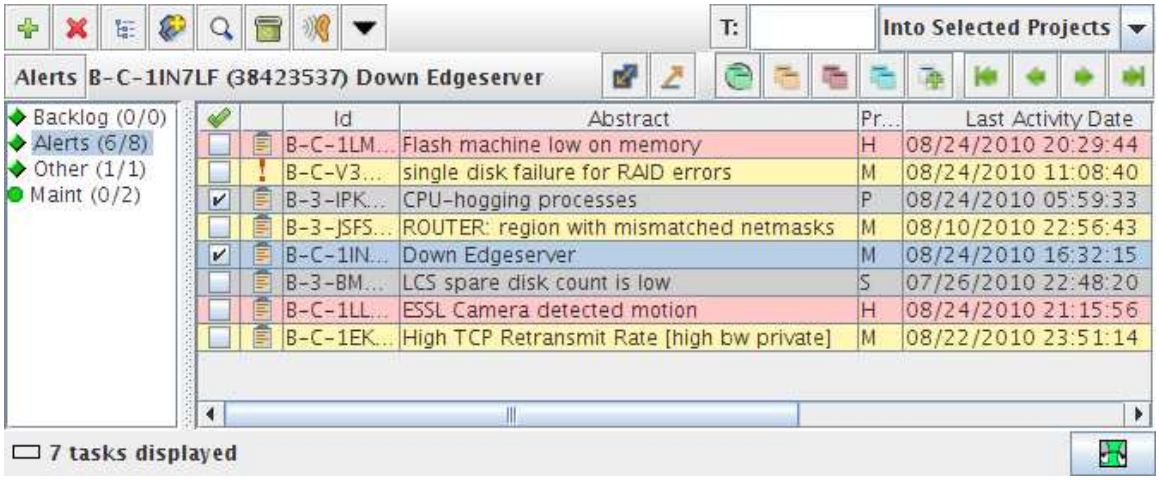

Figure 4: Alert-handling interface.

consume too many resources, and begin collecting results very quickly. Query also allows us to detect a problematic condition, then examine a large amount of information about it, all using one tool. Even if we did not anticipate needing some specific information before a problem is detected, we can write new queries and issue them at any time to help diagnose a problem.

The need to run the alert system using Query imposes several key constraints on Query's design that relate to the tradeoffs described in Section 2. Reliability, completeness, and low data latency are critical for the alert system. When the alert system issues a query, the answer needs to come back reliably, quickly enough, and be computed with data that are fresh and comprehensive enough to detect the problem promptly and respond to it before it affects our service to customers. The alert system also needs Query to be scalable. The ability to issue alerts to detect a wide variety of problems is quite useful. When the network grows in size and is handling more traffic, we need to continue to be able to answer all of the existing alert queries, as well as new ones that become necessary.

5.2 Historical Reporting System

Another tool for analyzing and diagnosing the Akamai network is the historical reporting system, which collects and stores data from Query over time and graphs the results. The reporting system is Akamai's primary tool for observing how the network has changed over time. While we use the alert system to detect issues that need immediate attention, we use the reporting system to proactively analyze network behavior with the intention of preventing issues before they occur.

Much like the alert system, the reporting system stores several thousand queries written by developers and operations staff. Each query is issued every few minutes and the results are shown on graphs. The system provides various ways of displaying data to assist in visualizing and understanding the parameters of the network.

The resolution of the reporting system, which issues each query once every several minutes, is insufficient to detect problems and respond to them in real-time. It is sufficient, however, to help understand problematic conditions over the span of several hours. For example, a bug in Query itself once caused it to consume too much CPU. Due to the difference in scale between the Akamai network and the test network, this bug was not realizable in the test environment. After deploying the new software with the bug to a small number of machines, the alert system detected the increase in CPU load on some machines. Before deploying the software to more machines, we investigated the problem. The reporting system showed spikes in the CPU utilization of Query on certain machines, and seeing the frequency of CPU spikes helped in diagnosing the bug.

5.3 Customer Access

Several Akamai services that provide data to customers use Query to collect that data. Most customer interaction with those systems is through a web-based interface, EdgeControl [3], which is the Akamai customer portal.

The alert system can issue alert queries on customers' behalf, notifying a customer if one of that customer's alerts fires. That notification may be done via e-mail, a web service call, or through an SNMP MIB that runs on the customer's site (which allows customer alerts to be integrated with local monitoring clients like Openview [10], and Tivoli [11]).

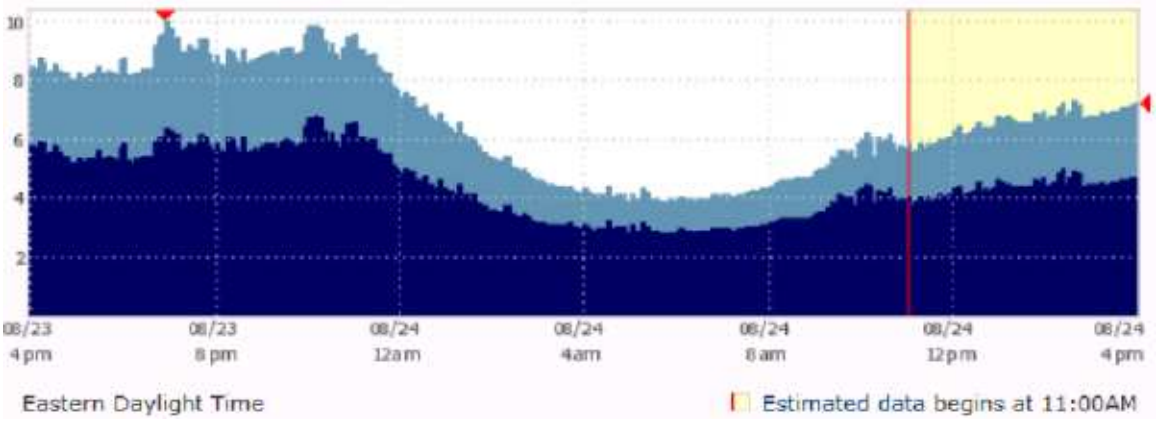

Figure 5: Customer access to traffic data via EdgeControl. The highlighted estimated data come from Query.

In addition to providing certain alerts to customers, we provide each customer with graphs of various data, such as how much traffic we have served for that customer over time. Figure 5 shows an example of such a graph. Query does not achieve perfect completeness. The first several hours of data are based on processing logs of all traffic we have served. However, we want to display usage graphs to our customers more quickly than we can process all the logs. Therefore we show customers log data until the latest time they are available, then show what we call estimated data for the most recent time period. That estimated data come from Query.

Several requirements arise from the fact that the data are customer-visible. The machines collecting the data have to be reliable, to have uninterrupted data display. In order for our displayed estimates to be as accurate as possible, the data need to be as complete as possible. Data must also be consistent across Query to avoid graph discrepancies. This is because a query may be issued to one SQL parser at a particular time and then issued to another SQL parser several minutes later. Additionally, data latency and query latency must be low, because we want to display near real-time data to customers quickly, providing them with current estimates. Finally, several of the queries whose results are displayed to customers are expensive and grow rapidly, joining multiple tables. Several of these tables grow as the number of machines in the Akamai network and the number of Akamai customers grow. Thus, Query must be scalable, to continue to handle the load from collecting the data for those graphs. Failure to provide features such as the ones outlined above would be unacceptable to customers.

5.4 Incident Response

An *incident* is an urgent occurrence that adversely affects customers, or may adversely affect them if left unchecked. Query is a vital tool for incident response at Akamai. As previously explained, it underlies the alert system and the reporting system, two important tools for incident response. Often, incidents begin when the alert system detects a high severity problem. If the problem is related to any of the thousands of graphs collected in the reporting system, that is another tool for understanding the problem.

In addition to being used by these tools that help with incident response, responders often issue SQL statements directly to Query. Much of the time, some information published into Query can help illuminate the problem and possible solutions. This use of Query, again, emphasizes certain goals for Query's design. Data latency and query latency are both vitally important: in an incident, we need to end problems before they impact customers, or minimize their impact, and every minute counts. Scalability is also important, because we don't know what tables will be needed until the incident takes place. The system may need to get any data from any or all of Akamai's machines, and the amount of data collected is far larger than any one machine can hold. The ability to divide data up among many machines, providing scalability, is vital to handling incidents. Our techniques for achieving such scalability are addressed in Section 6.

6 Scalability

In this section, we discuss the reasons Query has high needs for scalability and how we deal with those needs. We will address three ways of achieving scalability:

caching at each machine only the tables that machine needs; partitioning the network so that each machine needs only a subset of the data; and adding SQL parsers.

6.1 Causes of Growth

The volume of data and queries a TLA or SQL parser must be able to handle depends on several factors: the number of machines on the network publishing into Query, the number of customers about whom data are published, the volume of traffic on the network to be monitored, the number of services on each machine publishing data, and the number of ideas for things to monitor that people have come up with, among others. All of these factors grow monotonically with time.

As time goes by, Akamai signs more customers. As the Internet grows, our customers have more customers, so Akamai must serve more end users, leading to more traffic. To handle this additional load, we must deploy more servers. Managing rapid growth is one of the major challenges in the design and operation of Query.

6.2 Caching Policies

We try to cache tables around the SQL parsers and TLAs such that (1) each machine always has many of the tables it will need soon, and (2) machines have few tables they won't need soon. This is necessary because the volume of data available in Query is far larger than any single Query machine can hold: tens of gigabytes, a generation of which would take minutes to decode. That's why SQL parsers request tables from TLAs, which in turn request them from Cluster Proxies, which request them from the machines in their clusters.

Each machine *prewarms* tables. This means that it fetches those tables whether it needs them or not. That dramatically reduces query latency, because if the tables a query needs are already resident on the SQL parser, it doesn't need to spend minutes fetching them from the clusters through the TLAs. We can configure each aggregator set to prewarm its own distinct set of tables. That set can be thought of as our guess for which tables the aggregator will need. For example, we know what queries the alert system needs to issue for alerts. That means aggregators devoted to the alert system will need a specific known set of tables, so we prewarm that set.

Of course, tables exhibit temporal locality of reference: if a user issues a query using a table, that table is likely to be used again in the near future. If a table that isn't prewarmed is used, it continues to be requested in all generations for some period of time afterwards, and that timer is reset every time the table is used again.

A second type of caching is views. We cache the results of every view we compute for use in future queries, invalidating that cache every time we switch to using a new generation of tables. There are about 1000 view queries defined, many of which describe common subqueries. Storing their results reduces query latency and improves scaling in the number of queries by avoiding repeated computations. It also reduces the memory load on the SQL parsers, because the intermediate state for computing the answers to queries can be reduced by computing them fewer times.

Another technique for improving scalability is *diff updates*. Instead of sending a full copy of each encoded table to each TLA, the Cluster Proxies send only a diff – that is, a description of how the tables in that cluster have changed. The first time a Cluster Proxy sends data to a particular TLA, it sends a full generation, but subsequently, it sends only the diff. This makes TLAs decode tables more quickly, and saves about half the bandwidth Query would otherwise need to consume.

6.3 Partitioning

Partitioning the Query system can provide scalability. We have three ways of doing this: we can partition the network, we can partition the users, and we can partition the tables each individual user needs.

6.3.1 Network Partitioning

Talking to 73,000 machines takes a lot of resources from each TLA, but not all issuers of queries are interested in data from the whole network. Therefore we designate certain subsets of the Akamai network to be the span of each machine, as described in Section 3.5. For example, aggregators whose purpose is to monitor the Query system itself need span only the few hundred aggregators, not all 73,000 or more Akamai machines. Aggregators with small spans suffer far fewer demands on their memory, bandwidth, and CPU than machines that span all of Akamai.

6.3.2 User Partitioning

We don't want users going to randomly chosen aggregators to issue queries. Some applications, like the alert system, are critical, and must send queries to machines we know will have the resources to handle them. Some users run test queries to see how they perform, and while they are being written, they may mistakenly use excessive amounts of machine resources. This leads to assigning each aggregator set to a user or set of users, as described in 3.5.

After assigning a span and domain for each aggregator set, we can figure out what tables it is likely to need and make sure each machine in the set can decode, aggregate, and store all the tables it needs.

Because each machine in an aggregator set prewarms the same tables, no aggregator set can have more tables than one machine can handle. If a user is so demanding as to need more tables than a machine can handle, that user needs multiple aggregator sets. Users can issue queries to whichever of multiple aggregator sets they need, but operations staff responsible for Query need the ability to change where queries are sent without having to change the software of the components issuing the queries. We place each aggregator set behind a hostname and have users issue their queries to an arbitrary machine that hostname resolves to. Operations staff for Query can then change the machines a hostname resolves to, to add or remove aggregators from a set.

6.3.3 Table Partitioning

Partitioning tables among aggregators also helps with scalability. Suppose an aggregator set has four machines, $A, B, C,$ and D, and the tables it prewarms grow too large for one machine to handle. We can partition that aggregator set into two subsets, say A, B and C, D. On each subset, we prewarm half the tables the original set had. We point the same hostname at all four machines, but if A gets a query for which it doesn't have the tables and C, D do have them, A can send, in place of an answer, a message redirecting the query to C, D. The programmatic interface to Query then automatically goes to C or D to get its answer. In practice, the partitioning can't be perfectly even and some overlap between the tables on A, B and the tables on C, D must exist to still answer every query users want to issue. To date, in all cases where we have tried to partition an aggregator set in this fashion, no machine has needed more than 55% of the data of the original set.

6.3.4 Aggregator Sets and Fate-Sharing

Different users have different needs, but sometimes their needs are similar enough that they can be grouped together using a single aggregator set. There are a number of benefits, risks, and costs to doing so.

The main benefit is saving machines. Instead of many aggregator sets, we must deploy only one. The main risk is that two users on the same aggregator set share fate. If one user causes a failure, all of them will feel it.

The lesson here is that the most critical users should be isolated, and other users should be placed in groups with shared expectations about reliability and failure. If two non-critical applications that may potentially bring down an aggregator set have to share it, no problems will occur: both applications are non-critical and are designed with aggregator failures in mind.

6.4 Adding SQL Parsers

Akamai provides global traffic management and enhanced DNS services [2], mapping a hostname to several IP addresses and balancing the load among them. Because Query's users issue their requests to hostnames, rather than specific IP addresses, we can allocate the queries approximately evenly among all the IP addresses sharing a hostname. We create a hostname for each aggregator set's SQL parsers (or combined TLA/SQLs) and have our users issue queries to those. If we need to add more machines to handle more queries, we can do so transparently to the user. Twice as many machines can handle twice as many queries.

There are two caveats. First, we cannot simply add SQL parsers, because each TLA won't have the resources to send hundreds of megabytes of encoded tables (gigabytes of decoded tables) to that many other machines during an one or two-minute interval between generations. If we add too many SQL parsers, we must also add TLAs. Second, we must combine this approach with partitioning and wise caching policies. Each SQL parser must decode a generation of tables every time one arrives. Without partitioning and intelligent caching, as the network grows, eventually the SQL parsers will spend most or all of their time decoding generations.

7 Management Lessons

Managing a complex system like Query has taught us several lessons that may be of use to administrators of other systems. We need to be able to fix a variety of problems in Query that arise during operation. Some of these problems are due a user needing more tables than one aggregator set can handle. Some are due to a user needing to issue more expensive queries than their aggregators can handle, due to their requirements in either CPU or memory. Some are from the need for new features. Some are from software bugs. We now explain some of the lessons we have learned about these issues.

7.1 Management options

When a difficult use case arises, either due to new needs or due to organic growth, we have several options:

- Find a less expensive means of achieving the same goal.
- Reconfigure the network.
- Deploy additional hardware.
- Perform additional operational work to handle the use case.

- Develop the Query software to be more efficient about that use case.
- Go without, telling the user that the difficult use case cannot be accommodated.

The first option, finding a less expensive solution, is always the first thing we try, as it clearly saves the company the most money. Unfortunately, it isn't always possible, and the interesting tradeoffs are among the remaining options.

7.2 Configuration Options

For urgent problems, reconfiguring the network is usually the best option if it is a possibility. We can deploy configuration files to the entire network quickly to change what the machines are doing. For example, if a set of queries are taking up a lot of CPU on some set of SQL parsers, but they contain a common subquery, we can push a configuration file that creates a new view to reduce the number of times we need to compute it.

This example shows an important lesson: make rapid changes in behavior easy any time it is safe to do so. Some aspects of a machine's behavior, such as the software version, are difficult to change safely without restarting. Others, such as the views, are easy to change safely without restarting. In early versions of Query, all of these changes required a software install. Now, many just require our configuration management system [19] to copy new configuration files to the machines, which makes us much more reactive.

7.3 Adding Hardware

We can deploy new hardware to fix some problems. If a SQL statement is too expensive for the machines trying to run it, we can always put up additional SQL parsers to send it to. Deploying new machines takes less work than developing new software and can be done much more quickly. If we can't accommodate a request by configuration options or finding a more efficient way to achieve the user's goal, this is by far our most common solution.

7.4 Operational Intervention

Sometimes, a problem can be fixed by operations personnel manually. For example, we found a slow memory leak in Query that affected one set of aggregators such that their resources were essentially all consumed after about a month of continuous operation. We came up with a temporary solution to use until the next regularly scheduled release: manually restart the machines in the set every few weeks.

The lesson we've learned from trying this solution is that it's good for the short term only. It's expensive, because it requires a human in the loop. It's time consuming and stressful for operations personnel. It doesn't actually fix the problem; it's just a way of living with it. We try to use this approach as rarely as possible and to depend upon it only for short periods of time.

7.5 Software Development

Software development is a longer-term activity than pushing a configuration file or deploying new machines. To deploy new software, we must develop it, run it through Quality Assurance, and install it in several phases, allowing time between phases to make sure the part of the network that was installed initially is working properly.

The advantage to developing software to fix a problem or add a feature is that once it's done, the problem is fixed or the feature is available everywhere forever. No one has to do any work to maintain it, and there is no additional hardware cost.

One lesson we have learned about when to develop new software to solve a problem is that it's best to use it after the other solutions, because it's slower, cheaper, and more permanent. A few years ago, there was a bug that caused SQL parsers to be unable to get new data and to continue answering queries with old data. We initially solved this problem with operational work, adding an alert to the alert system to detect the condition and asking the Network Operations Control Center to restart machines when the alert fired. That was a temporary solution that lasted for a few months until we could fix the bug.

8 Handling Faults

With over 73,000 machines publishing into Query and several hundred running infrastructure for Query (TLAs, SQL parsers, and TLA/SQLs), some number of them are down at any time. Sometimes pairs of them can't reach one another. Sometimes TCP sockets between machines fail due to congestion. Sometimes machines have too many resources consumed and can't keep up with all the communication they're supposed to do. This section is about how we handle faults.

There are several goals regarding handling faults, including:

- **Easy detection:** Problems should be found quickly and easily.
- **Fault tolerance:** When a fault occurs, Query should work around it.

- **Quarantining faults:** The scope of a fault should be kept narrow, limiting the number of machines that go down.

8.1 Error Detection

Query is unusual among Akamai systems in that a lot of the other systems can count on Query to detect their faults. If something goes wrong with Query itself, we have an obvious bootstrapping problem.

If something goes wrong with a subset of the Query processes on the network, we can detect it because multiple aggregator sets span all the TLAs, SQL parsers, and TLA/SQLs. If any of those aggregator sets are functioning properly, we can detect problems. Additionally, we can detect problems that cause queries to fail rather quickly, because the absence of an answer coming back registers as an error in, for example, the alert system.

There remain two cases: incorrect results coming back that cause false positives for alerts, and incorrect results that cause false negatives. False positives are easy to deal with: when an error has been detected but the people looking into it can't figure out the root cause, they know to also bring in experts on Query to debug simultaneously. This shows another lesson we have learned: don't forget that your monitoring tools may be the problem when you've detected an error.

False negatives are trickier. Occasionally, an alert will not fire. Usually, we find this is due to a bug in the alert SQL, not a bug in Query. The only way to deal with that is for alert writers to test carefully before and after their alerts are deployed. If there were a mass-scale incident of Query failing to publish data, we would also detect that case, because of a number of alerts that check for the presence of data, not their absence. For example, if data from half the network were to disappear, the alert for Query having data from too few machines would fire almost immediately.

8.2 Fault Tolerance and Quarantining

When deciding how to achieve fault tolerance and quarantine faults, we must keep in mind the tradeoffs of Section 2.2. There is a tension between the two goals. Tolerating faults requires moving load away from a machine that fails, so that its outstanding requests may still be serviced. Quarantining faults requires that load *not* be moved away from a machine that fails, because the load may have caused the failure.

Aggregator sets help us limit the scope of failures while achieving fault tolerance. The SQL parsers of each set have a single hostname pointing to all of them. If a SQL parser fails, the programmatic interface to Query automatically redirects the query to another machine in the set. A Query that consumes enough resources to take down a machine could thus take down the whole set. This can happen occasionally due to an ad hoc query being written by a human, but only on non-critical aggregator sets used for development. If an aggregator set is used by humans writing ad hoc queries, we only send queries to it from applications that are allowed to fail to get answers sometimes.

If a TLA goes down, any SQL parsers talking to it continue providing answers to queries with old data until they can get tables from another TLA in the same aggregator set. This typically takes a few minutes (not much more than a normal interval between generations). This allows the same balance between tolerating faults and quarantining them as for SQL parsers failing.

If a TLA loses its connection to a cluster, similarly, SQL parsers switch away from it. Each TLA advertises how many clusters it can see and SQL parsers take that information into account when selecting TLAs. Initially, each SQL parser chooses a TLA arbitrarily. Suppose SQL parser S chose TLA $T1$. If $T1$ loses visibility to some clusters, some other TLA, $T2$, may gain the ability to see some percentage more clusters than $T1$ can. S will then switch to using $T2$ instead of $T1$. If there are multiple such $T2$, S will switch to an arbitrary one.

Usually, if there are connectivity problems, one TLA will fail to see some set of clusters, but the other TLAs will be able to see it. In other words, there will be multiple possible choices for a $T2$ to switch to. That prevents the switching algorithm from placing extreme load on any one TLA.

We want SQL parsers to prefer TLAs that are geographically close to them. Using configuration files we can tell each SQL parser to give a bonus to some set of TLAs when deciding which one to use. That helps make the mapping from SQL parsers to TLAs more static and prefer close by machines, while also helping each SQL parser have as complete a view of the network as possible.

TLAs are normally required to have a full view of the network before they can collect a generation. Each cluster must have reported tables within a certain time interval. If a cluster has failed to do so, the TLA drops the cluster's tables and advertises one fewer cluster, so that SQL parsers can switch away from it as needed.

9 Related Work

In this section we review related work in the area of large-scale network monitoring that has appeared throughout Query's lifetime of approximately 12 years.

Several system administration tools such as Nagios [14], Microsoft SCOM [21], Hewlett-Packard OpenView [10], IBM Tivoli [11], and Sun Management

Center [20] exist for monitoring network services and machine resources, often using SNMP. Akamai accomplishes network monitoring by feeding data collected and aggregated by Query into applications such as the ones discussed in Section 5. Query allows users to specify complex monitoring tasks using a SQL-like interface. In addition to providing a familiar interface, Query's focus is on scaling its monitoring capabilities to tens of thousands of machines, while still providing near real-time updates. Via a software component that acts as an SNMP gateway, Query is able to collect data published by SNMP-enabled devices, as was described in Section 3.1. Similarly, Query is also able to export data as an SNMP MIB, as was described in Section 5.3.

One common approach to network management for security purposes is Security Event Managers (SEMs). An SEM logs all events it expects will be interesting to system administrators. When a problem is detected, the SEM provides a means of reading the logs from each machine and presenting them to the administrators. By publishing such data into Query, Akamai has all of that information in one place that is easy to monitor constantly by human users and automated applications. Additionally, queries can be issued right away, minimizing the setup time for detecting conditions of interest.

Processing large volumes of continuously updated data in real-time has also been the focus of several academic and industrial research projects in the area of stream processing systems. Telegraph [6], STREAM [13], and Aurora/Medusa [7] were the first generation of such systems, with a focus on providing a SQL-like interface to query continuously updated data. The next generation of such systems focused on distributed implementations, to increase both the scalability and the fault-tolerance of low-latency, high-throughput stream processing applications. Borealis [1] has focused on challenges related to implementing a stream processing system in a distributed fashion, with particular emphasis in load shedding and fault-tolerance. Synergy [18] has focused on composing distributed stream processing applications, while paying attention to their end-to-end Quality of Service requirements. Among industrial research efforts in the area of distributed stream processing, IBM's System S [23] has focused on a variety of stream processing applications with highly variable rates, utilizing a large number of stream processing nodes. Additionally, AT&T's Gigascope [8] has focused on monitoring network traffic at extremely high-volumes. Similar to the systems above, one of Query's main challenges comes from the large data volumes that need to be processed near real-time. Query addresses this challenge via the clustered architecture outlined in Section 3 and the techniques described in Section 6.

Research efforts have also focused on the challenges faced by large-scale network monitoring systems, both due to data volume and network size, as well as due to network and machine failures. SDIMS [24] has attacked the scalability challenges by using Distributed Hash Tables to create scalable aggregation trees. It has also utilized lazy and on-demand reaggregation to adjust to network and node reconfigurations. PRISM [12] has proposed imprecision to provide consistency guarantees with reduced monitoring overhead and despite failures. Specifically, arithmetic imprecision was proposed to bound numeric inaccuracy, temporal imprecision to bound update delays, and network imprecision to bound uncertainty due to network and node failures. Query faces similar tradeoffs, as was described in sections 2 and 8.

Distributed event services can also be used for network monitoring. Research projects in this area, such as Siena [5] and ECho [9], have focused on maximizing the performance of event notification, while providing data models that are generic enough to express a variety of events. CORBA also provides support for event [16] and notification [17] services. Akamai uses Query to collect data from many different software components, implemented in a variety of programming languages. To achieve that, the publishers utilize various native language interfaces that Query provides, as was described in Section 3.1.

10 Conclusion

We have explained the goals and design of Query, Akamai's near real-time monitoring system. We have presented a number of the issues we face developing, managing, and operating it. We have stated some of the lessons we have learned from our experiences. Management of Query has been made much easier by the availability of rapid changes in configuration; isolating critical users and putting others into groups of similar reliability expectations; having multiple ways of addressing a problem in both the short term and the long term, and being explicit about which ones are good for each time scale; and having a strategy for debugging our monitoring tools. All of those strategies have allowed Query to scale with the Akamai network and handle the growth of load that it has been experiencing for more than a decade.

11 Acknowledgements

The authors would like to thank everyone who has developed and supported Query throughout the years, including Andy Berkheimer, Josh Buresh-Oppenheim, Timo Burkard, James Chalfant, Ron Chaney, Greg Fletcher, Stephen Gildea, Dan Katz, Sef Kloninger, Alan Krantz,

Phil Lisiecki, Andrew Logan, Brian Mancuso, Barry Margolin, Erik Nygren, Tim Olsen, James Park, Jan-Michael Santos, and Brad Wasson.

We would also like to thank Andy Ellis, our shepherd Chad Verbowski, and the anonymous reviewers, for valuable advice on the writing of this paper.

References

[1] ABADI, D., AHMAD, Y., BALAZINSKA, M., CETINTEMEL, U., CHERNIACK, M., HWANG, J., LINDNER, W., MASKEY, A., RASIN, A., RYVKINA, E., TATBUL, N., XING, Y., AND ZDONIK, S. The design of the Borealis stream processing engine. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA* (January 2005).

[2] AKAMAI EDGE PLATFORM PRODUCTS. `http://www.akamai.com/html/technology/products/index.html`, 2010.

[3] AKAMAI EDGECONTROL. `https://control.akamai.com`, 2010.

[4] AKAMAI EDGEPLATFORM. `http://www.akamai.com/html/technology/dataviz3.html`, 2010.

[5] CARZANIGA, A., ROSENBLUM, D., AND WOLF, A. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC), Portland, OR, USA* (July 2000).

[6] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., ANDJ.M. HELLERSTEIN, M. F., HONG, W., KRISHNAMURTHY, S., MADDEN, S., RAMAN, V., REISS, F., AND SHAH, M. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA* (January 2003).

[7] CHERNIACK, M., BALAKRISHNAN, H., BALAZINSKA, M., CARNEY, D., CETINTEMEL, U., XING, Y., AND ZDONIK, S. Scalable distributed stream processing. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA* (January 2003).

[8] CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, CA, USA* (June 2003).

[9] EISENHAUER, G., BUSTAMANTE, F., AND SCHWAN, K. Event services for high performance computing. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC), Pittsburgh, PA, USA* (August 2000).

[10] ENTERPRISE MANAGEMENT SOFTWARE: HP OPENVIEW. `http://www.managementsoftware.hp.com/`, 2010.

[11] IBM TIVOLI SOFTWARE. `http://www.ibm.com/software/tivoli/`, 2010.

[12] JAIN, N., MAHAJAN, P., KIT, D., YALAGANDULA, P., DAHLIN, M., AND ZHANG, Y. Network imprecision: A new consistency metric for scalable monitoring. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), San Diego, CA, USA* (December 2008).

[13] MOTWANI, R., WIDOM, J., ARASU, A., BABCOCK, B., BABU, S., DATAR, M., MANKU, G., OLSTON, C., ROSENSTEIN, J., AND VARMA, R. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA* (January 2003).

[14] NAGIOS. `http://www.nagios.org/`, 2010.

[15] NYGREN, E., SITARAMAN, R., AND SUN, J. The Akamai Network: A platform for high-performance Internet applications. *ACM SIGOPS Operating Systems Review 44*, 3 (July 2010).

[16] OMG. Event service specification v.1.2, 2004-10-02, 2004.

[17] OMG. Notification service specification v.1.1, 2004-10-11, 2004.

[18] REPANTIS, T., GU, X., AND KALOGERAKI, V. QoS-aware shared component composition for distributed stream processing systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 20*, 7 (July 2009), 968–982.

[19] SHERMAN, A., LISIECKI, P., BERKHEIMER, A., AND WEIN, J. ACMS: The Akamai configuration management system. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI), Boston, MA, USA* (May 2005).

[20] SUN MANAGEMENT CENTER. `http://www.sun.com/software/products/sunmanagementcenter/`, 2010.

[21] SYSTEM CENTER OPERATIONS MANAGER (SCOM). `http://www.microsoft.com/systemcenter/en/us/operations-manager.aspx`, 2010.

[22] VISUALIZING GLOBAL WEB PERFORMANCE WITH AKAMAI. `http://www.akamai.com/html/technology/visualizing_akamai.html`, 2010.

[23] WU, K., YU, P., GEDIK, B., HILDRUM, K., AGGARWAL, C., BOUILLET, E., FAN, W., GEORGE, D., GU, X., LUO, G., AND WANG, H. Challenges and experience in prototyping a multimodal stream analytic and monitoring application on System S. In *Proceedings of the 33rd Very Large Databases Conference (VLDB), Vienna, Austria* (September 2007).

[24] YALAGANDULA, P., AND DAHLIN, M. A scalable distributed information management system. In *Proceedings of the 2004 ACM SIGCOMM International Conference on Data Communication, Portland, OR, USA* (August 2004).