DBStream: A Holistic Approach to Large-Scale Network Traffic Monitoring and Analysis

Arian Baer[a,*], Pedro Casas[a], Alessandro D'Alconzo[a], Pierdomenico Fiadino[a], Lukasz Golab[b], Marco Mellia[c], Erich Schikuta[d]

[a] *FTW Forschungszentrum Telekommunikation Wien, Donau-City-St. 1, 1220 Vienna, Austria*
[b] *University of Waterloo, 200 University Avenue West Waterloo, ON, Canada*
[c] *Politecnico di Torino, Corso Duca degli Abruzzi, 24, 10129 Torino, Italy*
[d] *Vienna University, Waehringerstrasse 29, 1090 Vienna, Austria*

Abstract

In the last decade, many systems for the extraction of operational statistics from computer network interconnects have been designed and implemented. Those systems generate huge amounts of data of various formats and in various granularities, from packet level to statistics about whole flows. In addition, the complexity of Internet services has increased drastically with the introduction of cloud infrastructures, Content Delivery Networks (CDNs) and mobile Internet usage, and complexity will continue to increase in the future with the rise of Machine-to-Machine communication and ubiquitous wearable devices. Therefore, current and future network monitoring frameworks cannot rely only on information gathered at a single network interconnect, but must consolidate information from various vantage points distributed across the network.

In this paper, we present DBStream, a holistic approach to large-scale network monitoring and analysis applications. After a precise system introduction, we show how its Continuous Execution Language (CEL) can be used to automate several data processing and analysis tasks typical for monitoring operational ISP networks. We discuss the performance of DBStream as compared to MapReduce processing engines and show how intelligent job scheduling can increase its performance even further. Furthermore, we show the versatility of DBStream by explaining how it has been integrated to import and process data from two passive network monitoring systems, namely METAWIN and Tstat. Finally, multiple examples of network monitoring applications are given, ranging from simple statistical analysis to more complex traffic classification tasks applying machine learning techniques using the Weka toolkit.

Keywords:
Network Monitoring, Data Stream Warehouse, Machine-to-Machine Traffic, On-line Traffic Classification, Machine Learning, Cellular Networks

1. Introduction

Since the introduction of computer networks in general and the Internet more specifically, networked computer systems have become more and more important to modern society. Todays Internet is a highly complex, distributed system, spanning the globe and reaching even into outer space to the International Space Station. Human communication relies to a large extent on emails, (mobile) phone calls and social media. It has become normal to buy electronics, clothes or even cars, book flights and make bank transfers over the Internet. The financial market exchanges large amounts of stocks via interconnected high frequency trading systems. This shows that computer networks have become a corner stone of today's modern society.

Network operators are responsible for the proper functioning of those highly complex networks. They face the challenge of detecting and reacting very quickly to network anomalies, security breaches and, at the same time, plan ahead to adopt their networks to novel usage patterns. Network monitoring and analysis systems play a central role in supporting operators in these tasks. However, the above challenges put a wide range of requirements to the system in charge to collect, store, and process the gathered monitoring data. Such a system should be: (i) able to store data over extended time periods, (ii) make analysis results available quickly, on the order of minutes or even seconds, and (iii) network experts should be able to easily specify and extend typical analysis tasks. Whereas, many isolated systems and approaches have been proposed to capture and analyze network data [1, 2, 3, 4], there is still

*Corresponding author
Email addresses: arian.baer@gmail.com (Arian Baer), casas@ftw.at (Pedro Casas), dalconzo@ftw.at (Alessandro D'Alconzo), fiadino@ftw.at (Pierdomenico Fiadino), lgolab@uwaterloo.ca (Lukasz Golab), mellia@polito.it (Marco Mellia), erich.schikuta@univie.ac.at (Erich Schikuta)

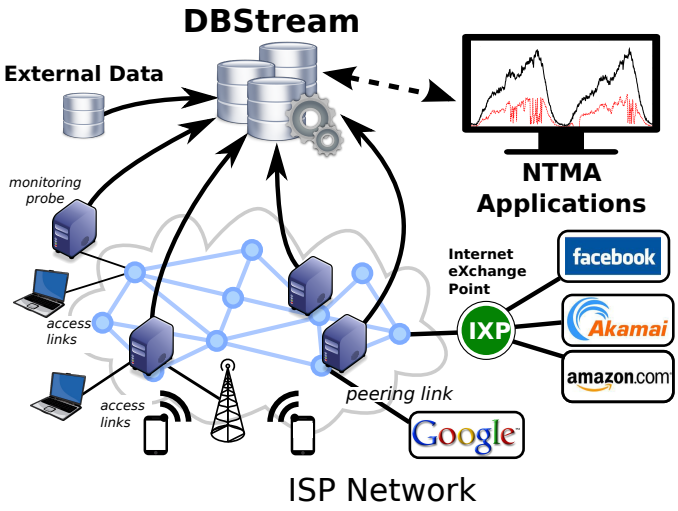


Figure 1: A standard deployment of DBStream in an ISP network. DBStream is a data repository capable of processing data streams coming from a wide variety of sources.

a clear lack of open, comprehensive approaches for integrating, combining and post processing data from multiple sources.

In this paper, we propose the open source system DBStream[1], a holistic approach to large-scale network data analysis. DBStream is a Data Stream Warehouse (DSW) based on traditional database techniques, designed with comprehensive network monitoring in mind. We show that DBStream is performance-wise at least on par with the most recent large-scale data processing frameworks such as Hadoop and Spark. We report the use of DBStream for several network monitoring and analysis applications, and the experience from its deployment in a production mobile network. Finally we show a DBStream integration with the well-known Weka Machine Learning (ML) toolkit can be used for on-line detection of Machine-to-Machine (M2M) devices in mobile networks, using only high level statistical information.

The specific contributions of the paper are:

- We propose the open source DSW DBStream.
- We present the high level, micro service architecture of DBStream.
- We show the high performance of DBStream by comparing it to state-of-the-art large-scale data processing frameworks.
- We demonstrate how the Continuous Execution Language (CEL) language empowers users to solve analytic challenges effectively.

The remainder of the paper is organized as follows. Section 2 presents the related work. In Section 3 and 4, we describe the system architecture and the processing language of DBStream, respectively. In Section 5, the performance of DBStream is compared to the in-memory MapReduce framework Spark. We discuss the impact of jobs scheduling on DBStream performance in Section 6. We provide in Section 7 an extensive report of the DBStream usage in several network traffic monitoring and analysis projects, as well as in a nation-wide mobile network. A prototypical integration of DBStream with a ML library is presented in Section 8, along with its application to M2M traffic detection. Finally, Section 9 provides the overall conclusions and an outlook on the future work.

[1] https://www.github.com/arbaer/dbstream

2. Related Work

The introduction of the term Big Data lead to a new era in which many scientific and commercial organizations started designing and developing novel large-scale data processing systems. Most of them achieve increased performance by re-implementing the whole or parts of the data processing engine. They often relax Atomicity, Consistency, Isolation, Durability (ACID) constraints [5] and/or apply novel data processing paradigms. Still, a limitation of such systems is the inability to cope with continuous analytics, where data arrive as high-volume, possibly delayed, data streams. Data Stream Management Systems (DSMSs), such as Gigascope [6], Borealis [7], Esper [8] or the more recent Streambase system [9], support continuous processing, but they cannot support analytics over historical data, as required in Network Traffic Monitoring and Analysis (NTMA) applications, and are not available as open source.

DSW systems extend traditional databases and Data Warehouse (DWH) with the ability to ingest and process new data in near real-time. DataCell [10] and DataDepot [11] are two examples, as well as the DBStream system presented in this paper. Another important development are Not only SQL (NoSQL) systems based on the MapReduce framework made popular by Google in [12]. Those systems use a key-value interface rather than a high level declarative language, like Structured Query Language (SQL), typically supported by Database Management Systems (DBMSs). Hadoop [13] and Hive [14] are two popular open source implementations of the MapReduce framework. Dremel [15] is a Google proprietary technology that exploits the MapReduce paradigm and uses a column oriented database to optimize web search. Spark [16] is another interesting system, promising an approximate 100x scale up factor with respect to the MapReduce implementation of Hadoop, by using an in-memory processing architecture.

MapReduce systems focus on processing data in large batches, rather than streams, as required for NTMA applications. There has been some recent work on enabling real-time analytics in NoSQL systems, such as Muppet [17], SCALLA [18] and Spark Streaming [19]. At the moment, the main focus of Spark Streaming lies on the processing

of real-time data, e.g., a stream of twitter feeds. Unfortunately, it is not possible out-of-the-box to perform non real-time processing, where data may arrive with delays of several seconds or even minutes. Nevertheless, Spark Streaming seems to be an interesting candidate for future network monitoring solutions.

However, with the exception of the proprietary, closed-source DataDepot system, none of these systems were designed to address continuous data processing, required for NTMA applications. Furthermore, to the best of our knowledge, DBStream is the only open source system that supports incremental queries defined through a declarative language. As we show in this paper, the continuous analytical capabilities, and the incremental query processing, make DBStream particularly suited for tracking the status of large-scale mobile networks and for traffic classification.

The field of automatic network traffic classification has been extensively studied during the last decade [20, 21]. The specific application of ML techniques to the traffic classification problem has also attracted large attention from the research community. A non-exhaustive list of standard supervised ML-based approaches includes the use of Bayesian classifiers, linear discriminant analysis and k-nearest-neighbors, decision trees and feature selection techniques, and support vector machines. In addition, many unsupervised and semi-supervised learning techniques have been used for network traffic classification, including the use of k-means, DBSCAN, and AutoClass clustering. Also the GRIDCLUST algorithm presented in [22] and its later extension to the BANG-clustering system [23] are good candidates for being used in NTMA applications, due to their computational efficiency on large datasets. We point the interested reader to [24] for a detailed survey on the different ML techniques applied to network traffic classification.

More recent approaches for traffic classification focus on the specific analysis of the applications running on top of HTTP/HTTPS [25, 26], including the analysis of modern Internet services such as YouTube, Facebook, WhatsApp, etc.

The particular classification and analysis of M2M traffic and M2M devices has very recently emerged as a need to understand the novel traffic patterns such devices introduce. So far only a few papers have explictily addressed this problem. The most relevant work on M2M traffic characterization is provided in [27]. There, the authors present an extensive analysis of the traffic generated by M2M devices in the AT&T US mobile network. They apply a Type Allocation Code (TAC)-based approach to separate M2M from other devices.

The M2M TRAffic Classification (MTRAC) approach presented in Section 8 aims at the classification of M2M devices in mobile network traffic. The whole system relies on DBStream for the data collection and processing, is operated online, and assumes the availability of only coarse-grained traffic descriptors at the user session level.

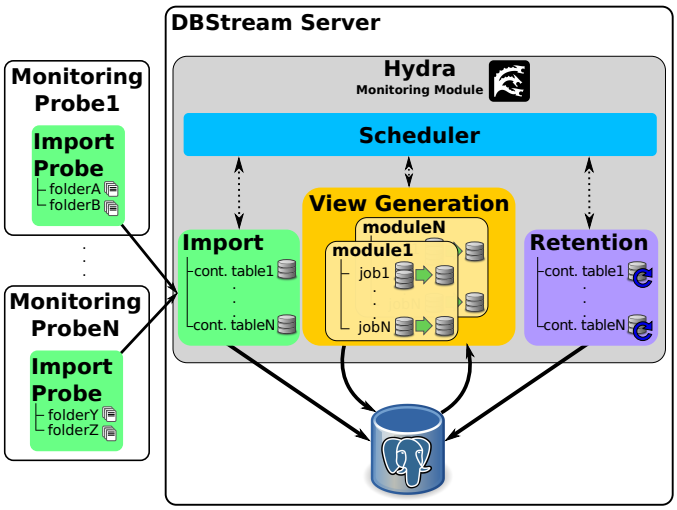


Figure 2: General overview of the DBStream architecture.

3. DBStream System Design

The main purpose of DBStream is to store and analyze large amounts of network monitoring data. But, it might also be applied to data from other application domains like e.g. smart grids, smart cities, intelligent transportation systems, or any other use case that requires continuous processing of large amounts of heterogeneous data data over time. DBStream is implemented as a middleware layer on top of PostgreSQL. Whereas all data processing is done in PostgreSQL, DBStream offers the ability to receive, store and process multiple data streams in parallel. In addition, DBStream offers a declarative, SQL-based CEL which is highly precise but yet very flexible and easy to use. Using this novel stream processing language, advanced analytics can be programmed to run in parallel and continuously over time, using just a few lines of code.

In Figure 2, a high-level overview of the architecture of DBStream is shown. The design of DBStream follows a micro service architecture and is composed of a set of decoupled modules, each executed as a separate operating system process. As opposed to a monolithic software architecture, modules can be stopped, updated and restarted without the need to stop and restart the whole DBStream system. The most important module is the `Scheduler`, which dictates the ordering in which jobs are executed. The `Import Probe` module, running on one or more monitoring probes, sends locally stored data to the `Import` module running on the DBStream side. The DBStream `Import` module writes data into time partitioned Continuous Tables (CTs) and also signals the availability of new data to the `Scheduler` module. Data from one or more CTs is read by jobs registered in the configuration of the `View Generation` modules and the results are written into new CTs, which are created automatically or appended to if the CTs do not already exists. In each of those jobs, data projections, transformations or aggregations are expressed in a batched data stream processing

language called CEL, which is explained in full detail in Section 4. The `Retention` module monitors the size of CTs and deletes old data if a certain pre-configured size limit is exceeded.

All modules register tasks for execution at the `Scheduler` module. As soon as the `Scheduler` detects that a certain task can now be executed, it messages the corresponding `Import`, `View Generation` or `Retention` module to start the task. The decision of when a certain task is ready for execution is based on two conditions. i) a full new window of data has to be available for all input time windows, meaning that all *precedence constraints* of a task have to be met. ii) the scheduling policy. As shown in detail in Section 6, it might not always be optimal to execute each job right away, but, in certain cases it is more efficient to wait for other jobs, sharing the same input partition. Therefore, in specific situations, e.g., when the maximum number of parallel jobs is exceeded, the `Scheduler` blocks the execution of certain jobs.

Each job advances one single CT. Meta data of each CT, along with the information until which point in time until a job has finished processing is persisted in the data dictionary of DBStream, whenever the internal state of the job is advanced by the execution of a task. In case the system crashes during the execution of a task and later on is restarted, the data dictionary is checked by the `View Generation` module for the latest point in time until which the job was finished. All intermediate tables, which might have been created before the crash, but are not complete, are deleted and recreated. This guaranties the *Atomicity* property of the ACID constraints.

The design decision to decouple job scheduling from job execution makes the system more flexible. Jobs can be executed by different `View Generation` modules and if one job has to be changed the `Scheduler` and all other `View Generation` modules can continue processing. Therefore, users can change or add new jobs to the system without impacting already running jobs.

As shown in Figure 2, all DBStream modules are started and monitored by an application server process called `hydra`. It reads the DBStream configuration file and starts all modules listed there. Each module has a standardized interface to provide status information. `Hydra` periodically fetches this status information and makes it available in a centralized location. Another crucial function of the `hydra` module is restarting other crashed modules. Since modules might depend on external processes potentially running on remote machines, they might crash at unpredictable moments. Therefore, all DBStream modules are designed and implemented such that they can crash at any point in time and leave the whole system in a recoverable state. This provides the guaranties of the *Durability* property of the ACID constraints to DBStream.

The communication with `hydra` as well as the communication between modules, e.g., between the `Scheduler` and the `View Generation`, is implemented as remote procedure calls over the HTTP protocol. Therefore, it is easily possible to distribute certain modules of DBStream over several machines.

Term	Description
window	A time slice of a stream, defined by **stream_name** (`window` **N** [`delay` **M**] [`primary`]) [`as` **window_name**] [, ...]
`primary`	Marks the window along which processing is advanced. `primary` can only be used once per job.
`__STARTTS`	Is replaced in a query with the start of the primary window.
`__ENDTS`	Is replaced in a query with the end of the primary window.
`delay`	Can be used to shift a window into the past.
job	Defines how inputs are transformed into the output stream. Its State tracks the application time until which the job has been finished.
task	Concrete unit of work which is executed to advance the state of a job.
application time	Time of the application, contained in the processed data.
system time	Time of the processing system, often referred to as *wall-clock time*.

Table 1: Definition of the most important terms of CEL.

3.1. System vs. application time

DBStream is a DSW and therefore similar to a stream processing system in many ways. In contrast to typical database applications, where time often is modeled as a column with a specific data type, time is an essential part of the architecture of DBStream. Therefore, the exact definition of time is crucial, determining how the system works and what kind of problems it can solve. The authors of [28] give an interesting overview of different methodologies for time handling in stream processing systems. Two of their definitions are very important to understand the time handling used by DBStream. First, the term **system time** is defined to be the wall-clock-time at the system processing the data. Second, the **application time** is used for timestamps which are part of the data processed by the system. Network monitoring systems typically assign a timestamp to observed events and use their wall-clock-time for this purpose. Since such systems are typically implemented as some sort of stream processing system, and one of the purposes of stream processing systems is to generalize the design of such systems, using the wall-clock-time directly is a reasonable choice.

The situation is very different for systems like DBStream. Here, the goal is to store and analyze the output

of other, typically remote systems which are stream processing systems themselves. Data arriving at DBStream already have a timestamp, assigned by another system in a higher layer of the processing chain. It would not be very helpful to add another wall-clock-time timestamp to the data. Instead, the time the event was created, already contained in the data, is of importance. Therefore, DBStream always uses the **application time** contained in the data for all time windows processing.

Performance is measured by the `View Generation` module on the level of tasks, not individual data rows. Each time a task is executed, the execution time of the task is measured and can be compared to the size of the primary window of that task. For example, the primary window is 10 minutes, meaning for each 10 minutes of time one task is executed. If the task can be executed in 1 minute, the whole job executes 10 times faster than real-time and thus the performance of this task is good.

4. Continuous Execution Language (CEL)

In this section, we describe the batched stream processing language CEL originally introduced in [29] in full detail. Table 1 gives an overview of the important terms of CEL. We start with a simple example explaining the main functions of CEL. In the following Section 4.1 we detail the Continuous Tables (CTs) used in CEL. Section 4.2 describes how time windows are handled in DBStream. Finally, in Section 4.3 we explain multiple complex examples showing the full expressive power of the presented language.

We start with a very simple example of CEL. Suppose we want to generate aggregate statistics from a router in the network under study. This router exports data on a per minute basis in the widely adopted NetFlow [30] format. Each row contains information on a per flow basis, where a flow is identified by the 5-tuple of <`source and destination IP, source and destination port and IP protocol number`>. In addition, each row contains information about the uploaded and downloaded bytes. Our first CEL query will compute the amount of uploaded and downloaded bytes passing through that router on a per hour basis.

Algorithm 1 Single window CEL job

```
<job inputs="A (window 60min)"
  output="W"
  schema="serial_time int4,
  total_download int8, total_upload int8" >
  <query>
select __STARTTS, sum(download),
  sum(upload) from A group by __STARTTS
  </query>
</job>
```

In CEL, such a job is expressed as presented in Algorithm 1. In this example, the `inputs` XML-attribute defines the input windows and the `output` XML-attribute defines the destination Continuous Table (CT) for the result. Here, only a single input window of 60 minutes of the CT A is defined, therefore a new task represented as a SQL query is executed in the underlying DBMS for each full hour of input data. The result of this SQL query is then stored in the CT with the name W. The SQL-query inside the `query` XML-element calculates the sum over all uploaded and downloaded bytes in one hour. In the `from` part of the query, the name of the CT A is used, which is a place holder and is replaced by a time slice of stream A of one hour for every executed task. Please note the special keyword `__STARTTS` which can be used in DBStream CEL jobs and is replaced with the start time of the primary window.

4.1. Continuous Tables

In DBStream all data are stored in Continuous Tables (CTs). First, raw data are imported into what would be called base tables in traditional DWHs. Jobs process data batches from those base tables and store the output into materialized views. Each job can have multiple inputs, which can either be base tables or materialized views. From each input, the job fetches a certain time slice, e.g., 5 minutes which is available to the SQL query inside the job like a regular table. The SQL queries are executed in PostgreSQL and use `INSERT INTO` to store the results in regular time partitioned tables. We refer to a time partitioned base table or materialized view in DBStream as a Continuous Table (CT), since both are handled in the same way. Please note, that in contrast to a DSMS, in DBStream all data are stored on disk and can be used in CEL jobs over extended time periods, only limited by the available disk space. For example, a job can compare the current hour of data with the same hour of the day one month ago, without keeping a full month of data in memory.

4.2. Time Window Definitions

In Figure 3, an overview of several possible time window definitions is shown. Those time windows define how jobs are continuously executed over time and which data is read from each input in each executed task. We give two more illustrative examples on how does time windows are used in real-world use cases in Section4.3.

CEL does not have an explicit definition for sliding windows, but instead, for each job one single window is marked to be the primary window by specifying the **primary** keyword in its definition. As soon as a task is successfully executed, the internal state of the corresponding job is moved into the future by the size of one primary window. When enough data in all input windows of the next task becomes available, the scheduler executes the next task if the maximum amount of parallel tasks is not yet

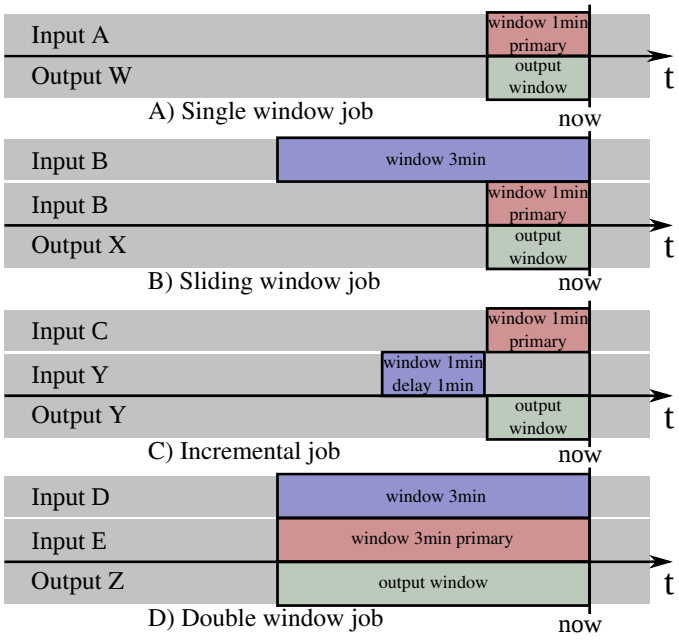


Figure 3: Multiple input window definitions possible in DBStreams Continuous Execution Language (CEL).

reached. The second important keyword for window definitions in CEL is **delay**. It can be used to shift a window into the past by a certain amount of time. For example, if the internal state of the job is "2014-11-21 12:20", a window of 1 minute would start at minute "12:19" and end at "12:20". If this window has a delay of 1 minute, given by the following definition `(window 1min delay 1 min)`, it would start instead at "12:18" and end at "12:19".

The window definitions visualized in Figure 3 have the following properties. Part A) shows the simplest possible window definition, similar to the window definition used in Algorithm 1. The single input window is also the primary window of the job. Such jobs are typically used for data projections, transformations and aggregations.

The window definition shown in part B) is an example of a sliding window. Since the primary window has a length of one minute, a task is executed every minute. The second window has a size of 3 minutes. In consecutive task executions, the time slices of the second window will overlap. An example of such a job is shown in Algorithm 2.

The most complex window definition is shown in part C). Here, the primary window is used to fetch data from CT C, whereas the other window is used to make the last minute of the output CT available as an input to the job. Such jobs are very useful whenever state information has to be kept over time. A detailed example of such a job is given in Algorithm 3.

Part D) shows a double window job. Such jobs are typically used to merge information from multiple sources, which provide different parts of the same data, e.g., two monitoring probes, each monitoring a different part of the network. Another typical usage scenario is information enrichment, e.g., one source could contain information about contacted IP addresses and another source contains Domain Name System (DNS) information, i.e., a mapping of IP addresses to host names. In this situation, a double window job can be used to combine the two information sources and produce a new stream containing contacted host names.

The concept of window definitions in CEL by using primary windows is, to the best of our knowledge, a novel feature among stream processing languages. Other stream processing systems use different approaches to define windows and especially sliding windows. For instance, in the well known StreamBase [9] system windows are specified by a size and a slide definition in the following way: `[SIZE x ADVANCE y TIME]`. Here, `x` defines the length of the window and `y` the amount of time after which new output is generated. The window definitions of StreamBase need the definition of a separate join statement if more than one stream should be used.

Another important improvement of the DBStream system over typical stream processing systems is that data is stored on disk after each task execution. Therefore, the state of all running jobs is always persisted to disk and can be recovered directly after a restart of DBStream. This has three advantages. First, it makes DBStream robust against hardware failures since no state information is lost in case of a system crash or even a power outage. Second, streams can be replayed starting in the past, only limited by the amount of disk space available for a certain CT. Third, DBStream can efficiently process jobs accessing information from a long time in the past, e.g., a job can have the current day and the same week day one month ago as inputs.

4.3. Examples

In this section, we explain a rolling window job and a complex incremental job in full detail, by giving two exhaustive examples of incremental data processing jobs.

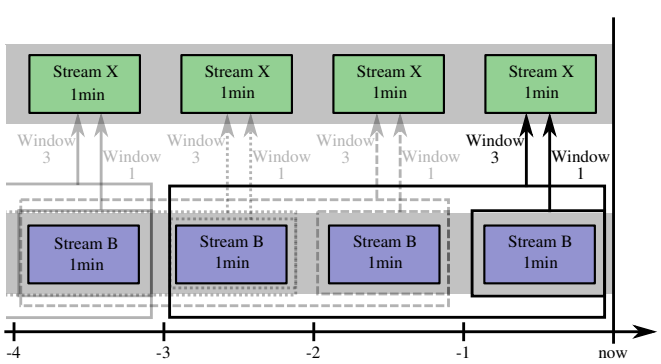
4.3.1. Rolling Window Average

In this example, we explain how a rolling average calculation can be implemented in CEL using a sliding window job.

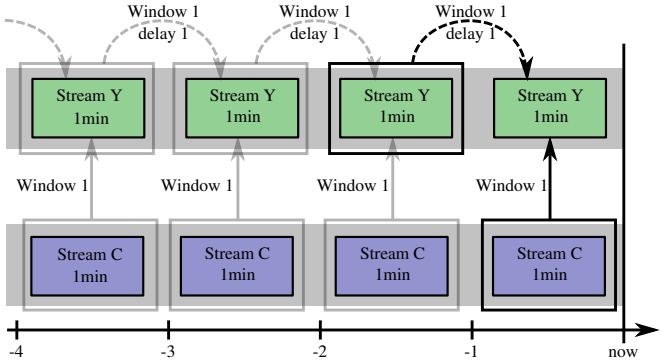
Algorithm 2 shows a job definition computing the average of uploaded and downloaded bytes over a sliding or rolling time window of three minutes. The job has two input windows, where the primary input window B1, along which processing is advanced, is one minute long. The sliding or rolling window B3, which is used for the average calculation in the SQL query, is three minutes long and uses the same CT as input. Figure 4a visualizes which parts of the input B are used over a period of four task executions.

4.3.2. Rolling Active Set

In the last example, we explain complex incremental job, incrementally computing the set of IP addresses active over the last hour, updated every minute.



(a) Rolling average over the last 3 minutes, updated every minute.



(b) Complex data processing flow for an incremental query.

Figure 4: Time windows for incremental data processing.

Algorithm 2 Sliding window CEL job

```
<job inputs="B (window 1min primary) as B1,
             B (window 3min) as B3"
  output="X"
  schema="serial_time int4,
          avg_download float8,
          avg_upload float8" >
  <query>
select __STARTTS, avg(download), avg(upload)
       from B3
  </query>
</job>
```

Algorithm 3 Incremental CEL job

```
<job inputs="C (window 1min primary),
             Y (window 1min delay 1min)"
  output="Y"
  schema="serial_time int4, last int4,
   ip inet" >
  <query>
select __STARTTS, max(last), ip
 from (
  select _STARTTS as last, ip from C
   group by 1,2 union all
  select last, ip from Y
   where last <= __STARTTS-60
   group by 1,2
 ) t group by 1,3
  </query>
</job>
```

Traditional large-scale batch, as well as stream processing systems, offer two different approaches to solve this problem. One approach is that for every minute, the last hour is queried and the active set of IP addresses is computed. This approach is similar to the previous example and can be useful in certain situations, especially if performance is not crucial e.g., on small amounts of data. Since the same minute of data is scanned over and over again, 60 times in the given example, this approach can become very resource intensive if data is large. Another approach is to keep all unique IP addresses encountered in the last hour along with a timestamp in memory, representing an intermediate state. This approach is very efficient regarding the processing time, but the active set has to stay in memory. In case the system is stopped or crashes due to a hardware failure or power outage, the in-memory state has to be rebuilt from past data, which might not be available anymore. In addition, in most Domain Specific Languages (DSLs), unlike in CEL, this type of jobs is not available out-of-the-box. Typically, User Defined Functions (UDFs) have to be used for implementing such behavior.

In the job implementation shown in Algorithm 3, we show how an incremental job can be used to calculate the set of IP addresses active over the last hour, updated every minute. This is achieved by using the past output of the job as an input, delayed by one minute. As we show in Section 5, this approach is much more efficient than the traditional ones. In addition, such a query stores all intermediate state on disk and can therefore be restarted at any time by just loading one minute of output data.

The input to this query is C which holds the IP addresses of active terminals. We now want to transform CT C into a new CT Y which contains for each minute, the distinct set of IP addresses active in the last hour. Therefore, we first add a timestamp called `last` to Y storing the time of the last activity of an IP address. Next, from the current minute of C, we generate a new tuple for each unique IP and set the last activity to the start of the window using the `__STARTTS` keyword. From the previous minute of the output stream Y, we select all IP addresses that where active less then 60 minutes ago. Finally, we merge both results using the SQL `UNION ALL` operator and we select from the result (for each distinct IP address) the current time, that is the maximum value of the last activity, and

the IP itself. This feedback loop allows us to efficiently compute the set of IP addresses active in the last hour every minute, without keeping any explicit state information. The windows used in this computation are shown in Figure 4b.

5. Performance Evaluation

In this section, we compare the performance of DBStream to the state-of-the-art Big Data processing framework Spark.

5.1. Spark Overview

Spark is an open-source MapReduce solution proposed by the UC Berkley AMPLab. It utilizes Resilient Distributed Datasets (RDDs), i.e., a distributed data abstraction which allows in-memory operations on large clusters in a fault-tolerant manner [16]. This approach has been demonstrated to be particularly efficient [31] enabling both iterative and interactive applications in Scala, Java and Python. Moreover, an application does not strictly require the presence of a Hadoop cluster to take advantage of Spark. In fact, the system offers a resource manager and supports different data access mechanisms. However, it is most commonly used in combination with Hadoop and the Hadoop Distributed File System (HDFS). Please refer to [29] for a detailed description of the implementation of the benchmark described in this section in Spark. Also the reasons for selecting Spark without the Spark Streaming extension are given there.

5.2. System Setup and Datasets

We installed Spark on a set of eleven machines with the following identical hardware: a 6 core XEON E5 2640, 32 GB of RAM and 5 disks of 3 TB each. One of those eleven machines has been dedicated to DBStream, recombining 4 of the available disks in a RAID10. We use PostgreSQL version 9.2.4 as the underlying DBMS. The remaining 10 machines compose a Hadoop cluster. The cluster runs Cloudera Distribution Including Apache Hadoop (CDH) 4.6 with the MapReduce v1 Job Tracker enabled. On the cluster we also installed Spark v1.0.0 but we were only able to use the standalone resource manager.

All machines are located within the same rack connected through a 1Gb/s switch. The rack also contains a 40 TB Network-Attached Storage (NAS) used to collect historical data. In particular, in this work we use four, 5 day-long datasets, each collected at a different vantage point in a real ISP network from the 3rd to the 7th of February 2014. Each vantage point is instrumented with Tstat [3] to produce per-flow text log files from monitoring the traffic of more than 20,000 Asymmetric Digital Subscriber Line (ADSL) households. For each TCP connection, Tstat reports more than 100 network statistics and generates a new log file each hour. Overall, each of the four dataset corresponds to approximately 160 GB of raw data, about

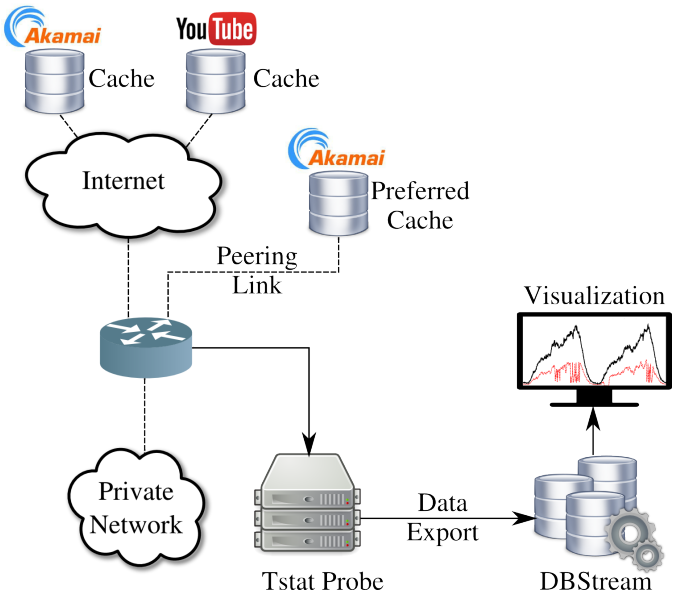


Figure 5: Tstat plus DBStream.

5 times the memory available on a single cluster node. In total, the four datasets sum up to approximately 650 GB, which is about twice as large as the total amount of memory available in the whole cluster. An overview of the setup along with the locations of some example widely adopted Internet services is given in Figure 5.

5.3. Job Definition

Based on our experience in the design of network monitoring applications and benchmarks for large-scale data processing systems, we define a set of 7 jobs that are representative of the daily operations we perform on our production Hadoop cluster.

Import imports the data into the system from the NAS, where raw data is stored in files of one hour each.

J1 for every 10 minutes i) map each destination IP address to its organization name through the Maxmind Orgname[2] database and ii) for each found organization, compute aggregated traffic statistics, i.e. min/max/avg Round Trip Time, number of distinct server IP addresses, total number of uploaded/downloaded bytes.

J2 for every hour, i) compute the organization name-IP mapping as in J1, ii) collect all data having organization names related to the Akamai CDN, and iii) compute some statistics, i.e. min/max/average Round-Trip Time (RTT), aggregated for the whole Akamai service.

J3 for every hour, i) compute the organization name-IP mapping as in J1, and ii) select the top 10 organization names having the highest number of distinct IP addresses connecting to them.

[2]The Maxmind Orgname database provides a mapping between IPs and Organization names; see `www.maxmind.com`.

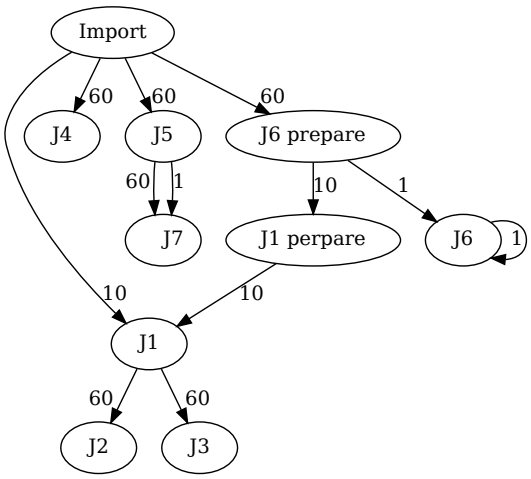


Figure 6: Job inter-dependencies for the DBStream job implementation.

J4 for every hour, i) transform the destination IP address into a /24 subnet, and ii) select the top 10 /24 subnets having the highest number of flows.

J5 for every minute, for each source IP address, compute the total number of uploaded/downloaded bytes and the number of flows.

J6 for every minute, i) find the set of distinct destination IP addresses, and ii) use it to update the set of IP addresses that were active over the past 60 minutes.

J7 for every minute, i) compute the total uploaded/-downloaded bytes for each source IP address, and ii) compute the rolling average over the past 60 minutes.

Overall, these jobs define network statistics related to Content Delivery Networks (CDNs) and other organizations (J1 to J4), statistics related to the monitored households (J5) and two incremental queries (J6 and J7) computing aggregated statistics over rolling sets of IP addresses.

5.4. DBStream Benchmark Implementation

All queries are implemented in the Continuous Execution Language (CEL) of DBStream, described in Section 4. The fact that the output of a job is stored on disk and can be used as input to another job is exploited to achieve increased processing performance. Figure 6 shows the resulting job dependencies, where the nodes represent the jobs and an arrow from e.g., job `J1` to `J2` means that the output of `J1` is used as input to `J2`. The number next to each arrow indicates the size of the input window in minutes. For example, in order to compute the results of `J6` we first gather the set of active IP addresses per minute in `J6` prepare. Then, `J6` uses `J6 prepare` and its own past output as input for the computation of the next output time window. This is indicated by the reflexive arrow starting from and going back into `J6`. A detailed example of the used CEL job underlying `J6` is given in Section 4.3.2.

5.5. System comparison

In Figure 7, we compare the performance of Spark and DBStream in terms of *makespan*[3]. In DBStream, the total execution time is measured from the start of the import of the first hour of data until all jobs finished processing the last hour of data. For Spark, all jobs were started at the same time in parallel. We report the total execution time of the job finishing last, which was `J6` in this experiment. Since for Spark, the import is done before the jobs start processing the data, we also report the job processing time plus the time it takes to import the data separately.

For the jobs `J1` to `J5` Spark offers great performance and the whole cluster is perfectly able to parallelize the processing, leading to very good results. Job `J6` and `J7` although, are not processed very fast. This comes from two factors: one the one hand, especially `J6` can not be parallelized very well, since data has to be synchronized and merged in one single node after each minute. On the other hand, distinct sets have to be computed for which huge amounts of data have to be moved around in the reduce phase. Please refer to [29] for the full details of the performance comparison. In the future, we plan to evaluate tools like Spotify Luigi[4] which are able to store intermediate results to speed up jobs like `J6` and `J7` in Spark.

For DBStream, the execution time increases nearly linearly with the number of Vantage Points (VPs) and therefore the amount of data to process. In contrast, for Spark the main factor is the execution time of `J6`. The total execution time does not increase much with more VPs since multiple instances of `J6` run in parallel. Therefore, Spark is able to utilize its parallel nature better the more jobs are running, whereas DBStream shows better performance for incremental jobs. For the one VP case, Spark, running on a 10 node cluster takes 2.6 times longer than DBStream running on a single node of the same hardware, to finish importing and processing the data.

6. Improving Performance with Intelligent Scheduling

In the setup considered for the performance comparison, the main bottleneck of the DBStream system is disk I/O. However, we will show that it is possible to minimize disk I/O overhead by intelligent tasks scheduling. In this section, we give an introduction to a more general scheduling problem found in disk-based continuous processing systems executing shared worflows. The automation of the scheduling presented here is part of future work and a first step towards this automation has already been published in [32].

[3] In operations research, makespan is the total time that has elapsed from the beginning of a certain project to its end. Here by makespan we mean the time until all jobs have been completed.

[4] `https://github.com/spotify/luigi`

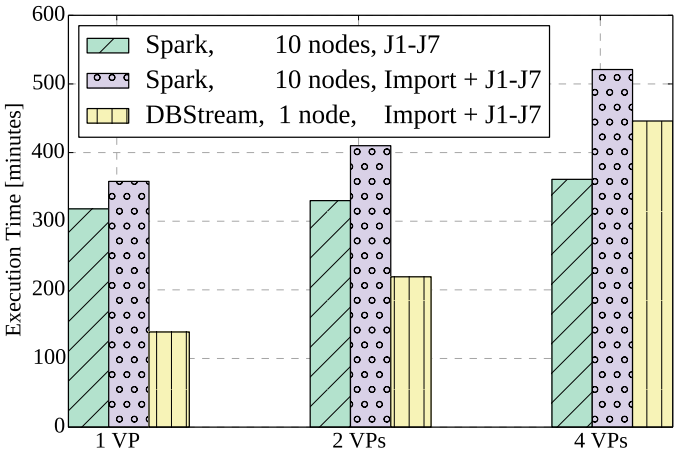


Figure 7: Comparison of DBStream and Spark.

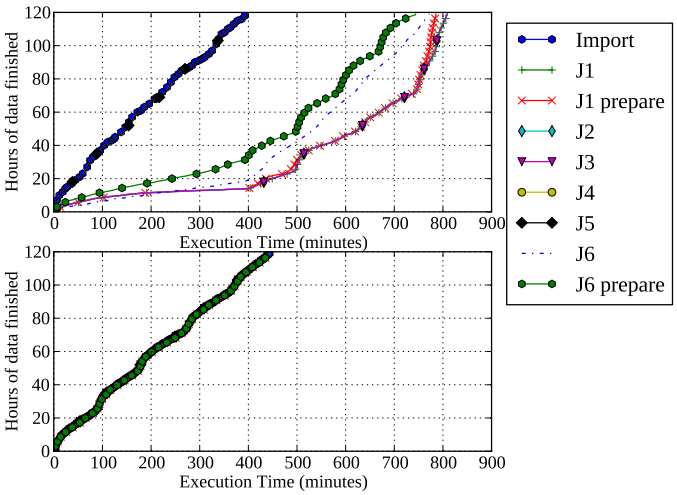


Figure 8: DBStream task execution over time for FIFO and Shared scheduling.

Typically, tasks are scheduled in first-in-first-out (FIFO) order in DBStream. Since we have set the number of parallel tasks to 64, FIFO effectively results in all tasks being executed as soon as the input data is ready. The effect of the FIFO scheduling is shown in Figure 8 (top), where each point in the plot corresponds to the execution of one window. The x-axis of this figure corresponds to the time after the start of the experiment at which a certain task finished execution. The y-axis corresponds to the time when the data item was created by the vantage point, normalized to the start of the whole dataset. Since some jobs process faster than others, in the FIFO case, the time distance between those jobs increases over the run of the experiment. The first step is the data import, which not only puts data onto the disk, but also into the disk cache of the Operating System (OS). As soon as the difference in progress of different jobs needing the same input gets too big, the data of the input drops out of the cache and has to be read from disk again. This increases the I/O overhead and, at the same time, decreases the overall system performance of DBStream.

Let us explain the underlying processes in more detail. For example, imagine the import has a fixed size of 8 GB per hour of data and takes 1 minute to finish processing. Let's assume as well, that there are only two jobs, A and B, defined on top of the import, and A needs 1 minute to process and B 2 minutes. Now we start the experiment and the first job which can execute is the import which needs one minute to finish. Then, both jobs A and B start to process hour 1 and the import starts processing hour 2. All those three tasks are executed in parallel in DBStream. After another minute has passed, the import has finished hour 2 and starts processing hour 3. Job A has finished hour 1 and now starts processing hour 2. Since the import and job A have the same processing time, their progress will only get, in the worst case, one hour apart form each other.

In the upper example, we defined hour of data to have the size of 8 GB. In a computer system with 16 GB of RAM available for disk cache, this will lead to the following situation. After the import has finished processing hour 1, this hour automatically is available in the cache from where it can be used by job A. Since the import and job A never get more then one hour apart from each other, no data has to be read from disk twice. In contrast, for job B, the situation is very different. Since the processing time of job B is 2 minutes and therefore twice as long as the import, for every imported hour, the time difference between job B and the import increases by one hour. Therefore as soon as this difference gets longer than 2 hours, the imported data does not fit in the disk cache any more and has to be fetched from disk again when it is needed by job B. This results in an increased I/O overhead since data need to be read from disk multiple times, thus decreasing the performance of the whole system.

Figure 8 (bottom) shows execution of the same set of jobs using a "shared" scheduling strategy. In the "shared" case, a new hour is only imported if the difference in time between the imported hour t_i and the hour t_j for which all jobs have finished processing is smaller than x.

$$t_i - t_j < x \tag{1}$$

In this case, data stays in the OS cache and fewer I/O operations are needed to complete the experiment. By setting $x = 1$, we are able to reduce the execution time of 4 VPs by a factor of 45% from 808 minutes to 446 minutes.

7. Experience from NTMA projects

DBStream has been adopted in several research projects for running a number of NTMA applications. To provide a concrete example, we report in Section 7.1 several statistics from running DBStream in the network monitoring project DARWIN4 [33], where it has been used as central analysis system. In addition, in Section 8 we present the M2M TRAffic Classification (MTRAC) approach [34]

as one prominent advanced analytics application of DBStream.

Besides these illustrative examples, there exist a number of NTMA projects, where DBStream has been fruitfully deployed, which we briefly summarize in the following for completeness. The authors of [35] describe DBStream as part of the general architecture for network monitoring envisioned in the European FP7 project mPlane [36]. In this project, DBStream has been integrated with the network monitoring system Tstat [3] to store the data it generates. Those data are then processed by analysis modules as described in detail in [37].

In [38] several performance impairments of the CDNs hosting Facebook and YouTube are analyzed using DBStream. An early study of performance degradations in YouTube is given in [39], which was later further detailed in [40]. Classification of HTTP traffic have been conducted in [41] and [42] using TicketDB [43], the predecessor system of DBStream. Most recently a characterization of the well-known Whatsapp chat service was studied in [44] using DBStream.

These examples show how easily DBStream can be deployed and instrumented to run diverse network traffic monitoring and analysis applications.

7.1. Operating DBStream at Scale

In this section, we present several statistics gathered from the DBStream installation operated in the Austrian nationally funded DARWIN4 [33] project. Those statistics give an overview of the scale at which DBStream can be operated although it is based on a classical database system and not the Hadoop stack. The focus of the project was on the development of innovative methods for i) detecting congestion in 2G/3G/4G mobile cellular network signalling capacity, ii) anomalies induced by macroscopic attacks, iii) malfunction of network equipment, as well as iv) the detection of highly synchronised M2M devices.

In the DARWIN4 project, many different data sources have been used, ranging from in-network passive probes to logs from the core network devices. Therefore, DBStream has been operated with several import modules. In particular, the project relied on the network monitoring system Measurement and Traffic Analysis in Wireless Networks (METAWIN) [45], tailored for passively monitoring operational mobile networks and capable of monitoring all links in the core of a Third Generation (3G) network. Figure 9 shows the setup when the monitoring system is connected to the *Gn* interface.

DBStream was successfully used to identify and analyze multiple network anomalies and to run many continuous analysis tasks in parallel. In fact, it has been used as the main analysis system in the project, and was the base for a near real-time alarming system relying on the availability of aggregate traffic statistics. Unfortunately, due to the non disclosure agreement constraints with the network operator, we are not allowed to present further details

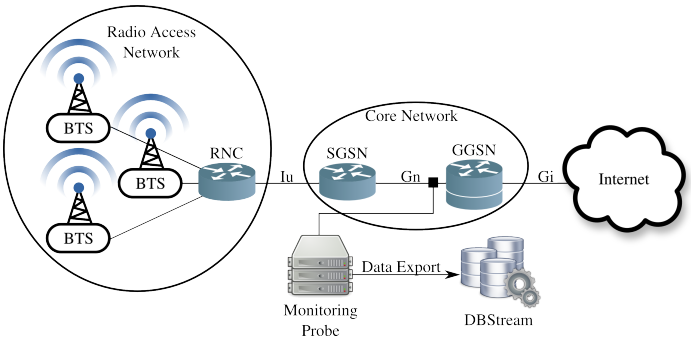


Figure 9: Simplified overview of a 3G network including a monitoring system, e.g., METAWIN, and the data export to DBStream.

about the applications running on top of DBStream in the DARWIN4 with the exception of the MTRAC approach presented in Section 8. Therefore, we report here several statistics about the general performance of DBStream.

DBStream was installed on a high performance server machine, hosting four AMD 6380 CPUs, running at 2.5 GHz. Each CPU houses 16 cores, resulting in a total of 64 cores. In total, we installed 256 GB of RAM. The disk subsystem in it's final state consists of four fiber-channel attached RAID arrays, each with 12x 2TB disks forming a RAID10. In addition, the 24 internal disks are split into two disks for the OS, running a RAID1, the other 22x 2TB disks form a large RAID6. All disks except those used for the OS are PostgreSQL tables spaces and are used by DBStream to store imported data and analysis results. To date, the DBStream installation operated in DARWIN4 is the largest one we are aware about.

In total, DBStream was operated for 385 days. On the final day of operation, the table partitioning resulted in 984,000 tables. Notice that this is a very high number of tables, considering that most databases use several hundreds and a typical database administrator knows most of them by name. Those tables stored a total of 67 TB of network monitoring data and processing results. Please note that these statistics, as well as the number of tables, are only a snapshot and include only those tables and amounts of data which were not already deleted by the DBStream retention module.

In total, over the whole run time, 9.482 million DBStream tasks (each corresponding to a PostgreSQL query) were executed. That is, on average, each 3.5 seconds a new task, updating a time window, was executed. Those tasks produced a total of 1.999 trillion result rows. Please note that in the current version of DBStream it is not possible to track the number of rows imported into DBStream, therefore this number should be considered as a lower bound and the actual number might be more than twice as high.

8. MTRAC - M2M Traffic Classification

In this Section we describe the MTRAC as one of the most important applications of DBStream not under Non-Disclosure Agreement (NDA) constraints.

Machine-to-Machine (M2M) traffic has become a major share of today's mobile networks and will grow even further in the near future. The quickly increasing number of M2M devices introduces unprecedented traffic patterns and fosters the interest of mobile operators, who whish to discover and track those devices in their networks. MTRAC enables the discovery of M2M devices relying on the analysis of coarse-grained network statistics by applying several different ML algorithms. Notice, that the use of very simple traffic descriptors makes MTRAC robust against traffic encryption techniques, and improve its portability to other types of networks or usage scenarios. We have designed and implemented MTRAC on top of DBStream using data from the network monitoring system METAWIN. Utilizing DBStream allowed us to classify M2M devices in near real time, using different time and session based network traffic aggregation methods. We report the performance of MTRAC for online classification of more than two months of traffic observed in an operational, nationwide mobile network.

8.1. Obtaining Network Descriptors from METAWIN

In the considered setup, the METAWIN monitoring system is connected to the *Gn* interface (see Figure 9). At this interface data from large parts of the network are concentrated, making it a suitable vantage point for network-wide analysis. In the METAWIN system, data is first captured at line-speed at the monitoring probe, equipped with one or more Endace capture cards [46]. Still on the monitoring probe, aggregations are generated for certain protocols, e.g., Transmission Control Protocol (TCP), Hypertext Transfer Protocol (HTTP) or DNS without applying any packet sampling. For MTRAC, network statistics aggregated at short time intervals in the minute range are used. Those network statistics are buffered locally on the monitoring probe using RAID arrays for optimized I/O. Finally, the statistics are fetched by the separate DBStream server and imported into the underlying PostgrSQL database.

8.2. DBStream Weka Integration

To enable online classification based on ML algorithms in DBStream, we added a new module able to interface DBStream with Weka [47]. Weka is a collection of machine learning algorithms for data mining tasks, and contains tools for data pre-processing, classification, regression, clustering, association rules, and visualization. It is also well-suited for developing new machine learning schemes.

The developed module enables users to write DBStream jobs which take a table of feature vectors as input and output a new table containing the classification results. This is achieved by the application of classification models, previously trained using Weka, to the table of feature vectors. This module can be used for any classification purpose. Generally speaking, Weka is instrumented to classify an exported time window of data use the pre-trained model. Then, the classification results are imported back into DBStream. As soon as the time window is imported, it becomes available to other DBStream jobs for further processing or visualization. Since the module is executed as a DBStream job, the DBStream scheduler automatically takes care of executing it for each new window of data.

8.3. TAC-based Ground Truth

Supervised classifiers need to be trained on a dataset containing the real class of the devices, i.e., the ground truth. Getting access to such labeled datasets is generally a very cumbersome process, especially in the case of an operational network. One standard approach followed by mobile network operators to identify a M2M device is by its hardware model [27], which can be obtained from its Type Allocation Code (TAC), using the TAC databases of the GSM Association. The hardware model information is generally complemented with cellular operator templates which provide a categorization of M2M devices, based on the device type (e.g., laptop, modem, POS, router, telemetry, etc.)[5]. This TAC-based approach imposes several limitations to the classification and discovery of M2M devices, such as: i) the need of manual gathering of TAC information whenever new devices appear in the network, and ii) incompleteness of the available TAC databases. We then train classifiers using only those devices for which the real class is known. Finally, this ground truth has been used to compute the accuracy of the different ML algorithms in terms of True Positive (TP) and the False Positive (FP) ratios.

8.3.1. Online M2M Classification

In this section, we show how the features described in detail in [34] are used by MTRAC to identify M2M devices in the operational mobile network of a European Internet Service Provider (ISP). The features extracted form the session data of each device are stored along with the corresponding ground truth of the device, obtained by the TAC-based approach of Section 8.3. Part of these data are used to build the ML-based classification models, training different classifiers using Weka. The trained models are finally installed into DBStream, and used in an online basis to assign a class to each of the monitored devices. Recall that our classification problem is a dichotomic one, in which a device is either classified as M2M or non M2M. As discussed in detail in Section 8.3.3, we use multiple ML-based approaches to improve the classification performance of MTRAC.

[5]For example, the AT&T specialty vertical devices template at `http://www.rfwel.com/support/hw-support/ATT_SpecialtyVerticalDevices.pdf`.

8.3.2. Aggregating Sessions per Device

The main challenge when aggregating sessions per device is to find a timing which still leads to results in an acceptable short time, but gathers enough data to achieve good classification results. In an offline setting, one can aggregate all sessions per device available in the whole dataset and select only those devices which meet certain criteria to perform the classification. For example, one might restrict the classification only to those devices for which a minimum amount of at least N sessions have been observed. In contrast, in an operational online setting, the dataset does not have a defined end. On the one hand, the amount of sessions used as input to the aggregation should be as high as possible. For example, features based on the session inter-arrival time can only be generated if more than one session is available and many statistical features benefit from more input data. On the other hand, the time it takes until the classification results are available should be as short as possible, thus reducing the number of sessions available in the aggregation period. As shown in Section 8.3.3, the classification performance increases with the number of aggregated sessions. Therefore, the user of such a system is facing an interesting discrepancy. She can either wait longer time until more sessions are available for aggregation and gain a higher classification performance, or receive the results earlier and accept the resulting lower classification performance.

We implemented two different session aggregation approaches to visualize this trade off. The first approach is called **Simple Daily Aggregation (SDA)** and is based on time, meaning that we execute the aggregation from sessions to feature vectors, e.g., after 1, 2, ..., N days. The second approach, called **Threshold Based Aggregation (TBA)**, is based on the number of sessions observed per device. The TBA approach is implemented using a rolling DBStream job (please refer to Section 4.3 for an example of rolling/incremental DBStream jobs). The job starts from a table `A` containing all sessions from all devices as they are produced. We now want to produce a new table `B` in which all sessions are kept, until at least S sessions for a single device have been gathered. For simplicity, let us assume we update `B` only once a day. A new time window of `B` is thus created by the union over all sessions of the current day stored in `A`, plus all sessions of all devices from the last time window of `B` which have not reached S sessions yet. The result is that a session is moved from the old to the new time window of `B`, until there are S sessions available for that device. The session aggregation can now be applied on those devices of `B`, having at least S sessions and is stored in a new DBStream CT.

8.3.3. Evaluation of Classification Accuracy

We have evaluated six different ML algorithms.

Decision Stump is a decision tree algorithm generating trees of only one level, therefore only a single feature is used to decide to which class a device belongs.

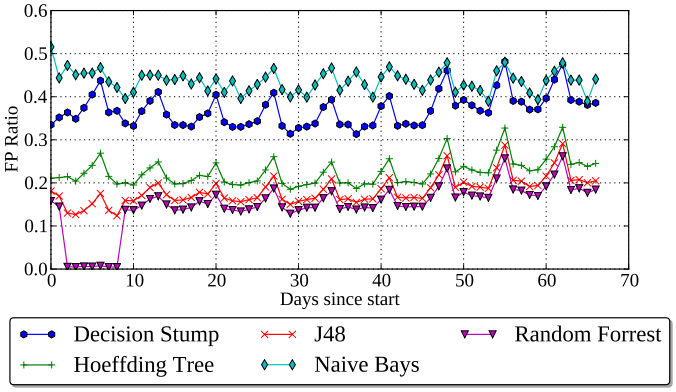


Figure 10: FPR per day for selected classifiers.

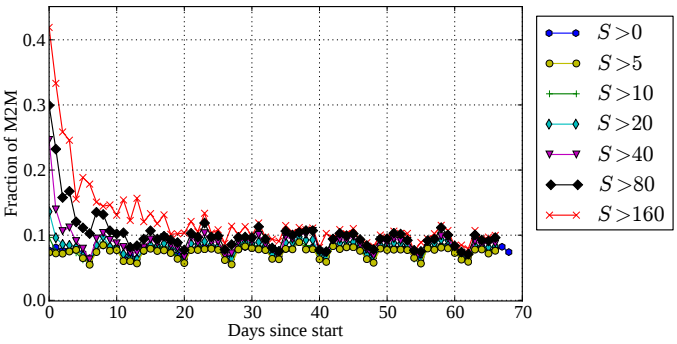


Figure 11: Fraction of M2M devices

J48 is an implementation of the well-known C4.5 decision tree learner provided by the Weka toolkit.

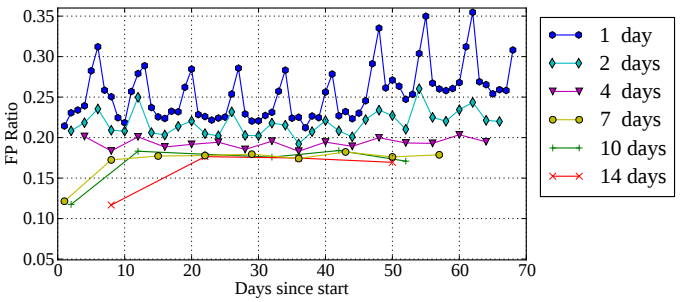
Random Forest trains an ensemble of decision tree learners, each on a randomly selected subset of the given features using bootstrapping to generate unique sample subsets for each tree.

Hoeffding Tree is a special decision tree learning algorithm. It produces classification models quickly, which can be updated dynamically as soon as new items arrive.

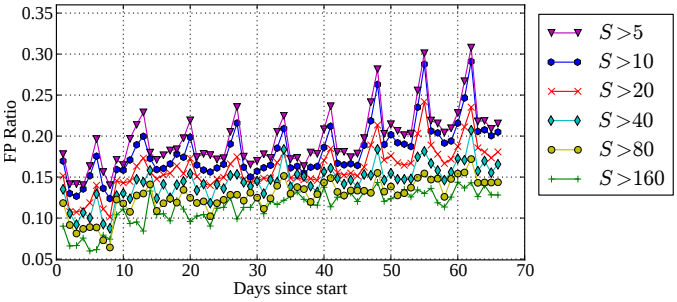
Naive Bayes is a statistical classifier based on Bayes' theorem with a strong (naive) assumption that each feature is independent from each other feature.

SVM is a non-probabilistic binary classifier. Support Vector Machines (SVMs) typically provide high classification performance at the cost of long training phases.

In Figure 10, we compare the FPR achieved by different classification algorithms, aggregating device sessions through the 10 session TBA approach. The days two to eight are used as a training set, therefore this period shows a decreased FPR, most prominent for the random forest algorithm. In this classification problem, complex tree algorithms like the Hoeffding, J48 and random forest achieve the lowest FPRs. The best performance is achieved by the random forest algorithm, at the cost of a very long training lasting several hours. The J48 algorithm provides the best balance between training time and classification performance. Therefore, we have used the J48 algorithm in the following. We also trained a SVM model, which re-



(a) SDA based on different number of days



(b) TBA based on different number sessions (S)

Figure 12: Comparison of our different session aggregation approaches SDA and TBA using a J48 classifier.

sulted in a very low performance classifier, where nearly all the devices where classified as non M2M.

As shown in Figure 11, the fraction of M2M devices is small but represents definitely an important share of the devices in the network, and it further decreased over weekends. This is likely the main cause for the decrease in classification performance during weekends, which results in the spikes of the FPR shown in Figure 10. In fact, the correct identification of M2M devices becomes harder as soon as the fraction of M2M devices becomes lower.

In Figure 12, we compare the SDA to the TBA session aggregation approach, using J48 models in both cases. In Figuret 12a we show the FPR for the SDA approach, aggregating sessions based on an increasing number of days. For each aggregation we export the first part, i.e., the first day, the first two days, etc., as training set for the J48 classifier. The FPR decreases with longer aggregation intervals, although aggregation intervals longer than 7 days do not seem to decrease the FPR any further. Figure 12b shows the classification performance for the TBA approach. In general, the FPR is lower than for the SDA. Also here, longer aggregation intervals result in a decreased FPR, which can get as low as 11.6% in average for $S > 160$. In total, the TBA approach performs considerably better for longer aggregation intervals as compared to the SDA approach.

Finally, it is interesting to analyse how long it takes until a device is classified, especially for the TBA approach. For this purpose, Figure 13 shows the normalized cumulative amount of devices reaching at least S sessions. The number of devices with more than S sessions grows slower for larger thresholds. For example, for the $S > 160$ TBA approach, even after an investigation period of more than two months, only 33.8% of devices pass this threshold.

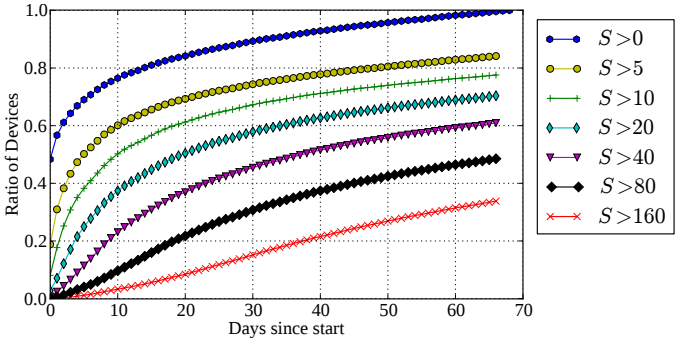


Figure 13: Cumulative ratio of classified devices

9. Conclusion and Future Work

In this paper, we presented DBStream, a Data Stream Warehouse (DSW) tailored for, but not limited to, Network Traffic Monitoring and Analysis (NTMA) applications. We have shown, that if instrumented correctly, a PostgreSQL database engine can process large amounts of data in a fast and efficient way.

In a performance study, we demonstrated that a single-node instance of DBStream can outperform a cluster of 10 Spark nodes by a factor of 2.6, running the same query workload on the same dataset.

The flexibility of DBStream was presented in another application, where it was instrumented to run multiple complex machine learning tasks. The resulting MTRAC approach, based only on the analysis of coarse-grained network descriptors, shows a very high accuracy for the continuous classification of M2M devices in a 3G mobile network.

The current design of DBStream is the result of its usage for several NTMA applications and its deployment in a mobile operational network. This experience allowed us to derive useful insights on how to improve the system to offer increased performance and higher flexibility at the same time. Although current results indicate that DBStream is already very much suited system for typical network monitoring applications, some technical challenges and interesting research questions remain to be solved. For example, we want to investigate the possibility of extending DBStream by replacing the database engine PostgreSQL with either the parallel database system Greenplum [48], or a MapReduce based large-scale data processing framework like, e.g., Spark [16]. Indeed, this would be a logical extension of the current single machine DBStream archi-

tecture to a cluster system, thus enabling scale-out properties found in modern big data processing frameworks.

Furthermore, we have deployed DBStream in the intelligent transportation systems domain, and plan its adoption also in other application domains with similar properties such as smart grid and smart city. In fact, data from those application domains has similar properties. Data arrive as high volume data streams and the analytic questions can be addressed utilizing DBStreams CEL language. Preliminary results show that DBStream can be used to store and analyze data from those domains as successfully as from computer networks.

10. Acknowledgments

The research leading to these results has received funding from the Vienna Science and Technology Fund (WWTF) through project ICT15-129, "BigDAMA", and from the European Union under the FP7 Grant Agreement n. 318627, "mPlane" project. The work has been partially performed within the framework of the projects Darwin 4 and N-0 at the Telecommunications Research Center Vienna (FTW), and has been partially funded by the Austrian Government and the City of Vienna through the program COMET. We would like to thank the anonymous reviewers for their detailed comments and suggestions, which helped us to significantly improve the quality of the paper.

References

[1] F. Ricciato, Traffic monitoring and analysis for the optimization of a 3g network, IEEE Wireless Commun. 13 (6) (2006) 42–49.
URL http://dx.doi.org/10.1109/MWC.2006.275197

[2] K. Keys, D. Moore, R. Koga, E. Lagache, M. Tesch, k. claffy, The architecture of CoralReef: an Internet traffic monitoring software suite, in: Passive and Active Network Measurement Workshop (PAM), RIPE NCC, Amsterdam, Netherlands, 2001.

[3] A. Finamore, M. Mellia, M. Meo, M. M. Munafò, P. D. Torino, D. Rossi, Experiences of internet traffic monitoring with Tstat, IEEE Network 25 (3) (2011) 8–14.
URL http://dx.doi.org/10.1109/MNET.2011.5772055

[4] F. Fusco, L. Deri, High speed network traffic analysis with commodity multi-core systems, in: Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement 2010, Melbourne, Australia - November 1-3, 2010, 2010, pp. 218–224.
URL http://doi.acm.org/10.1145/1879141.1879169

[5] M. Stonebraker, Sql databases v. nosql databases, Commun. ACM 53 (4) (2010) 10–11.
URL http://doi.acm.org/10.1145/1721654.1721659

[6] C. D. Cranor, T. Johnson, O. Spatscheck, V. Shkapenyuk, Gigascope: A stream database for network applications, in: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003, 2003, pp. 647–651.
URL http://doi.acm.org/10.1145/872757.872838

[7] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. B. Zdonik, Aurora: a new model and architecture for data stream management, VLDB J. 12 (2) (2003) 120–139. doi:10.1007/s00778-003-0095-z.
URL http://dx.doi.org/10.1007/s00778-003-0095-z

[8] EsperTech Inc., Esper: Event processing for java (2015).
URL http://www.espertech.com/products/esper.php

[9] StreamBase Inc., Streambase: Real-time, low latency data processing with a stream processing engine. (2014).
URL http://www.streambase.com

[10] E. Liarou, S. Idreos, S. Manegold, M. L. Kersten, Monetdb/datacell: Online analytics in a streaming column-store, PVLDB 5 (12) (2012) 1910–1913.
URL http://vldb.org/pvldb/vol5/p1910_eriettaliarou_vldb2012.pdf

[11] L. Golab, T. Johnson, S. Sen, J. Yates, A sequence-oriented stream warehouse paradigm for network monitoring applications, in: Passive and Active Measurement - 13th International Conference, PAM 2012, Vienna, Austria, March 12-14th, 2012. Proceedings, 2012, pp. 53–63.
URL http://dx.doi.org/10.1007/978-3-642-28537-0_6

[12] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
URL http://doi.acm.org/10.1145/1327452.1327492

[13] T. White, Hadoop: the definitive guide, O'Reilly, 2012.

[14] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy, Hive - A warehousing solution over a map-reduce framework, PVLDB 2 (2) (2009) 1626–1629.
URL http://www.vldb.org/pvldb/2/vldb09-938.pdf

[15] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis, Dremel: Interactive analysis of web-scale datasets, PVLDB 3 (1) (2010) 330–339.
URL http://www.comp.nus.edu.sg/~vldb2010/proceedings/files/papers/R29.pdf

[16] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: Cluster computing with working sets, in: 2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010, 2010.
URL https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets

[17] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, A. Doan, Muppet: Mapreduce-style processing of fast data, PVLDB 5 (12) (2012) 1814–1825.
URL http://vldb.org/pvldb/vol5/p1814_wanglam_vldb2012.pdf

[18] B. Li, E. Mazur, Y. Diao, A. McGregor, P. J. Shenoy, SCALLA: A platform for scalable one-pass analytics using mapreduce, ACM Trans. Database Syst. 37 (4) (2012) 27.
URL http://doi.acm.org/10.1145/2389241.2389246

[19] M. Zaharia, T. Das, H. Li, S. Shenker, I. Stoica, Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters, in: 4th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'12, Boston, MA, USA, June 12-13, 2012, 2012.
URL https://www.usenix.org/conference/hotcloud12/workshop-program/presentation/zaharia

[20] A. Dainotti, A. Pescapè, K. C. Claffy, Issues and future directions in traffic classification, IEEE Network 26 (1) (2012) 35–40.
URL http://dx.doi.org/10.1109/MNET.2012.6135854

[21] S. Valenti, D. Rossi, A. Dainotti, A. Pescapè, A. Finamore, M. Mellia, Reviewing traffic classification, in: Data Traffic Monitoring and Analysis, Springer Berlin Heidelberg, 2013, pp. 123–147.

[22] E. Schikuta, Grid-clustering: an efficient hierarchical clustering method for very large data sets, in: 13th International Conference on Pattern Recognition, ICPR 1996, Vienna, Austria, 25-19 August, 1996, 1996, pp. 101–105. doi:10.1109/ICPR.1996.546732.
URL http://dx.doi.org/10.1109/ICPR.1996.546732

[23] E. Schikuta, M. Erhart, The bang-clustering system: Grid-based data analysis, in: Advances in Intelligent Data Analysis, Reasoning about Data, Second International Symposium, IDA-97, London, UK, August 4-6, 1997, Proceedings, 1997, pp. 513–524.

[24] T. T. T. Nguyen, G. J. Armitage, A survey of techniques for

internet traffic classification using machine learning, IEEE Communications Surveys and Tutorials 10 (1-4) (2008) 56–76.
URL http://dx.doi.org/10.1109/SURV.2008.080406

[25] P. Fiadino, A. Bär, P. Casas, HTTPTag: A flexible on-line http classification system for operational 3g networks, in: Proceedings of IEEE Infocom 2013, Turin, Italy, 2013.

[26] I. Bermudez, M. Mellia, M. M. Munafò, R. Keralapura, A. Nucci, DNS to the rescue: discerning content and services in a tangled web, in: Proceedings of the 12th ACM SIGCOMM Conference on Internet Measurement, IMC '12, Boston, MA, USA, November 14-16, 2012, 2012, pp. 413–426.
URL http://doi.acm.org/10.1145/2398776.2398819

[27] M. Z. Shafiq, L. Ji, A. X. Liu, J. Pang, J. Wang, A first look at cellular machine-to-machine traffic: large scale measurement and characterization, in: ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012, 2012, pp. 65–76.
URL http://doi.acm.org/10.1145/2254756.2254767

[28] N. Dindar, N. Tatbul, R. J. Miller, L. M. Haas, I. Botan, Modeling the execution semantics of stream processing engines with SECRET, VLDB J. 22 (4) (2013) 421–446.
URL http://dx.doi.org/10.1007/s00778-012-0297-3

[29] A. Baer, A. Finamore, P. Casas, L. Golab, M. Mellia, Large-scale network traffic monitoring with DBStream, a system for rolling big data analysis, in: 2014 IEEE International Conference on Big Data, Big Data 2014, Washington, DC, USA, October 27-30, 2014, 2014, pp. 165–170. doi:10.1109/BigData.2014.7004227.
URL http://dx.doi.org/10.1109/BigData.2014.7004227

[30] B. Claise, G. Sadasivan, V. Valluri, M. Djernaes, RFC 3954: Cisco systems NetFlow services export version 9 (2004).
URL http://www.ietf.org/rfc/rfc3954.txt

[31] Berkeley AMPLab, Big data benchmark (2014).
URL https://amplab.cs.berkeley.edu/benchmark/

[32] A. Bär, L. Golab, S. Ruehrup, M. Schiavone, P. Casas, Cache-oblivious scheduling of shared workloads, in: IEEE 31th International Conference on Data Engineering ICDE 2015, April 13-17 2015, Seoul, South Korea, to appear, 2015.

[33] The Telecommunications Research Center Vienna (FTW), Data Analysis and Reporting for Wireless Networks (DARWIN4) (2014).
URL http://www.ftw.at/research-innovation/projects/darwin-4?set_language=en

[34] A. Baer, P. Svoboda, P. Casas, MTRAC - discovering M2M devices in cellular networks from coarse-grained measurements, in: 2015 IEEE International Conference on Communications, ICC 2015, London, United Kingdom, June 8-12, 2015, 2015, pp. 667–672. doi:10.1109/ICC.2015.7248398.
URL http://dx.doi.org/10.1109/ICC.2015.7248398

[35] B. Trammell, P. Casas, D. Rossi, A. Baer, Z. Ben-Houidi, I. Leontiadis, T. Szemethy, M. Mellia, mplane: an intelligent measurement plane for the internet, IEEE Communications Magazine 52 (5) (2014) 148–156. doi:10.1109/MCOM.2014.6815906.
URL http://dx.doi.org/10.1109/MCOM.2014.6815906

[36] mPlane Consortium (project FP7-ICT-318627), mPlane - an Intelligent Measurement Plane for Future Network and Application Management, http://www.ict-mplane.eu/ (2015).

[37] mPlane Consortium (project FP7-ICT-318627), Final Implementation and Evaluation of the Data Processing and Storage Layer, www.ict-mplane.eu/sites/default/files//public/public-page/public-deliverables//1187mplane-d34.pdf (2015).

[38] P. Fiadino, A. D'Alconzo, A. Baer, A. Finamore, P. Casas, On the detection of network traffic anomalies in content delivery network services, in: 2014 26th International Teletraffic Congress (ITC), Karlskrona, Sweden, September 9-11, 2014, 2014, pp. 1–9. doi:10.1109/ITC.2014.6932930.
URL http://dx.doi.org/10.1109/ITC.2014.6932930

[39] P. Casas, A. D'Alconzo, P. Fiadino, A. Baer, A. Finamore, On the analysis of qoe-based performance degradation in youtube traffic, in: 10th International Conference on Network and Service Management, CNSM 2014 and Workshop, Rio de Janeiro, Brazil, November 17-21, 2014, 2014, pp. 1–9. doi:10.1109/CNSM.2014.7014135.
URL http://dx.doi.org/10.1109/CNSM.2014.7014135

[40] P. Casas, A. D'Alconzo, P. Fiadino, A. Baer, A. Finamore, T. Zseby, When youtube does not work - analysis of qoe-relevant degradation in google CDN traffic, IEEE Transactions on Network and Service Management 11 (4) (2014) 441–457.
URL http://dx.doi.org/10.1109/TNSM.2014.2377691

[41] P. Casas, P. Fiadino, A. Baer, Understanding HTTP traffic and CDN behavior from the eyes of a mobile ISP, in: Passive and Active Measurement - 15th International Conference, PAM 2014, Los Angeles, CA, USA, March 10-11, 2014, Proceedings, 2014, pp. 268–271.

[42] P. Casas, P. Fiadino, A. Baer, IP mining: Extracting knowledge from the dynamics of the internet addressing space, in: 25th International Teletraffic Congress, ITC 2013, Shanghai, China, September 10-12, 2013, 2013, pp. 1–9. doi:10.1109/ITC.2013.6662933.
URL http://dx.doi.org/10.1109/ITC.2013.6662933

[43] A. Bär, A. Barbuzzi, P. Michiardi, F. Ricciato, Two parallel approaches to network data analysis, in: 5th Workshop on Large Scale Distributed Systems and Middleware (LADIS) 2011, Seattle, USA, 2011.
URL http://www.eurecom.fr/publication/3463

[44] P. Fiadino, M. Schiavone, P. Casas, Vivisecting WhatsApp in Cellular Networks: Servers, Flows, and Quality of Experience, in: M. Steiner, P. Barlet-Ros, O. Bonaventure (Eds.), Traffic Monitoring and Analysis (TMA), Barcelona, Spain, LNCS, 2015.

[45] F. Ricciato, P. Svoboda, J. Motz, W. Fleischer, M. Sedlak, M. Karner, R. Pilz, P. Romirer-Maierhofer, E. Hasenleithner, W. Jäger, Traffic monitoring and analysis in 3g networks: lessons learned from the METAWIN project, Elektrotechnik und Informationstechnik 123 (7-8) (2006) 288–296. doi:10.1007/s00502-006-0362-y.

[46] Emulex, Endace dag cards - 100% packet capture guaranteed - high speed packet capture, any network interface. (2015).
URL http://www.emulex.com/products/network-visibility-products-and-services/endacedag-data-capture-cards/features/

[47] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, Ian H. Witten, The WEKA Data Mining Software: An Update, in: SIGKDD Explorations, Vol. 11, Issue 1, 2009.

[48] Pivotal Software, Inc., Pivotal greenplum database - enable analytic innovation (2015).
URL http://www.gopivotal.com/big-data/pivotal-greenplum-database