Implementation of a Stream-based IP Flow Record Query Language

Kaloyan Kanev, Nikolay Melnikov and Jürgen Schönwälder

Computer Science, Jacobs University Bremen, Germany
{k.kanev,n.melnikov,j.schoenwaelder}@jacobs-university.de

Abstract. Internet traffic analysis via flow records is an important task for network operators. There is a variety of applications, targeted at identifying, filtering or aggregating flows based on certain criteria. Most of these applications exhibit certain limitations when it comes to the identification of complex network activities. To overcome some of these limitations, a new flow query language has been proposed recently, which allows to express complex time relationships between flows. In this paper, we describe a prototype implementation of this query language and we evaluate its performance.

Key words: Flow Query Language, NetFlow, Network Monitoring

1 Introduction

Internet traffic analysis via flow records is an important task for network operators. There is a variety of applications, targeted at identifying, filtering or aggregating flows based on certain criteria. Most of these applications exhibit certain limitations. Query definitions often have a non-uniform structure and are difficult to write and to maintain. Furthermore, query languages are often restricted to very basic flow matches and they usually cannot be used to detect complex flow patterns, mainly due to missing time and concurrency matching mechanisms. To overcome some of these limitations, a new stream-based flow query language has been proposed recently [1]. It allows to express complex time relationships between flows by utilizing Allen's time interval algebra [2] for describing time relationships between flows and flow groups.

In this paper, we focus on implementation aspects of the stream-based flow query language. Our research aim was to address the following questions:

- How can the stream-based flow query language be implemented in an extensible manner?
- What is the performance impact of using a high-level programming language?
- How does the complexity of the merger impact the overall execution time?

While our primary goals for the prototype were completeness and correctness, we analyze our prototype to identify performance critical sections. The rest of the paper is organized as follows. In Section 2 we provide an overview of the Flowy prototype implementation. We evaluate the performance and discuss the complexity of the Flowy implementation in Section 3. Section 4 reviews related work before Section 5 concludes our paper.

2 Flowy Prototype Implementation

In this section we provide an overview of the Flowy architecture and its implementation. Using a sample query, we explain the stages of query execution that correspond to the components of the query language.

Each stage is implemented as a separate Python class and consists of two modules. Validator modules are used to initiate and interconnect (passing one validator as an argument into the following validator) all stages. They also perform all necessary checks (i.e., double filter definition) of the defined query syntax. Execution modules define methods used at execution time. For the purpose of illustration we employ the following query example that makes use of all stages.

```
splitter S {}

filter www_req {
   dstport = 80
}

filter www_res {
   srcport = 80
}

grouper g_www_req {
   module g1 {
      srcip = srcip
      dstip = dstip
      etime < stime delta 1s
   }
   aggregate srcip, dstip, sum(bytes) as bytes, count(rec_id) as n,
             bitOR(tcp_flags) as flags, union(srcport) as srcports
}

grouper g_www_res {
   module g1 {
      srcip = srcip
      dstip = dstip
      etime < stime delta 1s
   }
   aggregate srcip, dstip, sum(bytes) as bytes, count(rec_id) as n,
             bitOR(tcp_flags) as flags, union(dstport) as dstports
29 }
30
31 groupfilter ggf {
32    bitAND(flags, 0x13) = 0x13
33 }
34
35 merger M {
36    module m1 {
37       branches B, A
38       A.srcip = B.dstip
39       A.srcports = B.dstports
40       A.bytes < B.bytes
41       B oi A OR B d A
42    }
43    export m1
44 }
45
46 ungrouper U {}
47
48 "./netflow-trace.h5" -> S
49 S branch A -> www_req -> g_www_req -> ggf -> M
50 S branch B -> www_res -> g_www_res -> ggf -> M
51 M->U->"./ungrouped.h5"
```

The splitter is always defined the way it is shown in the example above. It is used for copying records to respective branches. There are two filters defined in the query: `www_req` and `www_res`. The `www_req` filter is a part of branch A, and it selects flow records with destination port value of 80. The `www_res` filter is a part of branch B and performs the same task as `www_req`, but for source ports. Each of the two branches have groupers `g_www_req` and `g_www_res` following the filters. Both groupers define similar rules, but have a difference in the aggregation mechanism. In this case, there are three rules that say: ”form a group based on the same source and destination IPs of the records, and make sure that the start time of the next record does not exceed the end time of the previous record by more than one second”. The aggregation mechanism attaches to each group aggregated information as meta data. In this case it sums up all the bytes of the records present in groups, counts the number of records, bitwise-or all the TCP flags and creates a list of all source ports appearing at group records (branch A) or the list of all destination ports (branch B). The group filter performs absolute filtering on the group records. The rule of our sample groupfilter `ggf` states that the aggregated flags of each group should contain SYN, ACK and FIN flags. Another possibility could have been a rule like `bytes > 1024`, which indicates that groups with an aggregated record size of at least 1024 bytes will pass the rule. The merger `M` contains a module with four rules and is specified to operate on two branches (A, B). It defines a rule (line 38) to match source IP addresses of branch A groups to destination IP addresses of group B. Having matching source and destination IP addresses is not sufficient information to identify an HTTP download session, so the aggregated source/destination ports of each branch’s

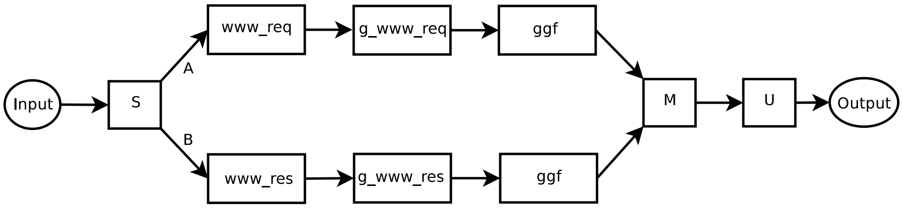

Fig. 1. Stream query for identifying HTTP download sessions

groups should match as well (line 39). Normally, the request for data download should be much smaller in size than the actual data. The next rule (line 40) indicates that the download requests from branch A should be smaller than the responses (with requested data) from branch B. The last rule (line 41) represents an instance of Allen's time algebra. The operator `oi` (overlap inverse) requires for the record groups from branch B (responses) to occur after the record groups from branch A (requests), and `d` (during) indicates that responses from branch B should occur during requests from branch A. The ungrouper `U` expands groups into records. These will represent records that satisfy the query. The last part of the query definition indicates the way modules are interconnected. Essentially, the same conceptual view of this two-branch query for HTTP download sessions is presented in Fig. 1.

2.1 Python, PyTables, and PLY

Flowy was implemented in Python, using PyTables [4] (an HDF V.5 access library) for storing the flow records and Python Lex and Yacc (PLY) [5] for generating the parser. The Python programming language was chosen because it is a very high-level, dynamically typed language, which allows rapid development and provides various convenient features relevant to this project. PyTables is a storage solution used by Flowy. It provides an interface to store large amounts of data and is based on the Hierarchical Data Format (HDF) [6]. HDF was chosen due to its fast and memory-efficient performance at the initial testing stages of the implementation effort. The PLY parser generator was chosen due to familiarity with the Lex and Yacc conventions. Since the query files have a relatively simple structure, and are not expected to become very large, parser performance is not of great importance for the Flowy implementation.

2.2 Records

The main unit of data exchanged through Flowy's processing pipeline is the network flow record. Records can be grouped (i.e., at the grouper stage) to form group-records. A `Record` class that deals with reading the records and storing them into the PyTables is dynamically created at runtime. This was done in order to add flexibility to the tool, and to allow it to process different versions of NetFlow [7] data (NetFlow V.5 flow records are different from NetFlow V.9 or IPFIX flow records).

2.3 Filters and Rules

The filter Python module implements the filtering stage of the pipeline. It is important to note that there is only one `Filter` class instance for all branches. Instead of using a splitter, which copies each record to the filter for each branch of the pipeline, the `Filter` instance reads every record and matches it against all filter rules in the branches. By doing so, the filter stage performs absolute filtering, as none of the records are compared to each other. In order to identify which branch's rule was matched, a record mask is added to the record. The record mask is a tuple of True/False flags, corresponding to the branches where the record should be passed to. Thus, the filter stage produces a stream of (record, record mask) pairs.

Each filtering statement is converted to a `Rule` class instance, against which records are matched. Rule instances are constructed from a branch mask, an operation (=, <, etc.), and arguments. Arguments may be either constants, or references to fields from the record being matched.

2.4 Branches and Branch Masks

The record mask shows which branches the record being filtered should be passed to. This is a simplified example of how two filters `a` and `b` from branches A and B respectively, are turned into a set of rules with their corresponding branch masks:

```
filter a {                         filter b {
   prot = protocol("TCP")             prot = protocol("TCP")
   dstport = 80                       bytes > 1024
}                                  }

# protocol("TCP") is evaluated during parsing to its numeric value 6
Rule(((True), (True)), EQ, [Field("prot"), 6])
Rule(((True), (False)), EQ, [Field("srcport"), 80])
Rule(((False), (True)), GT, [Field("bytes"), 1024])
```

The first argument of the Rule constructor is a simplified branch mask, used here instead of a real one, for clarity. The first rule's branch mask indicates that it is applicable to both filters and it should match flow records representing TCP connections. The second Rule is applicable only to branch A, and it checks that the source port is equal to 80. The last Rule is valid only for branch B and it states that the size of the record should be greater than 1024. After matching against all filter rules, the record is passed to the splitter, which copies it only to branches for which the branch mask is True. For more complicated queries that may involve logical OR, a subbranch mechanism is implemented, but its discussion is left out of this paper. It is also possible to construct composite filters - new filters from existing filters.

2.5 Splitter

The splitter module is used for copying records to respective branches based on their record masks. Branches do not necessarily go through all records following the filtering stage, therefore, separate threads are used for each branch, in order to be able to do work even if other branches are record-starved. The `Splitter` class takes a mapping from branch names to branch objects. It has a method `split()`, which dispatches the record to the branches marked by its mask. The method `go()`, iterates through all the records available from the filter and splits them to the corresponding branches.

2.6 Grouper

The grouper module forms groups of flow records based on the defined rules. The structure of the rules is similar to those of the filter module. Group objects contain group records' information for the absolute rule matching, as well as the first and last records of the group - needed for matching relative rules (i.e., group occurrence times comparison). Aggregation operations are performed on Python callable `AggrOp` objects. They aggregate on the given records. Each group has its own set of `AggrOp` objects. Their initialisation happens by passing the record field they should read, the field of the group record they should store the end result in, and the data type of the record field. The data type is needed because some operations, like average, should return the result in the same type as the record field.

The implementation of the grouping algorithm uses a different approach from the theoretical description presented in [1]. The nesting order of the loops, which iterate over groups and records are reversed. Rather than tagging the records by passing over the whole set of untagged records for each new group, the implementation keeps the groups list in memory, and for each record it checks whether it belongs to the group. This removes the need for random reads from slow permanent storage and achieves the grouping in a single pass over the records. User-defined aggregation operations may be imported using the `--aggr-import` command line argument.

2.7 Group Filter

Group filters work like a simplified version of the normal record filter. The branch masks mechanism is not used with group filters, since group filters read and export records from a single branch. Besides plain filtering, each group filter adds the records to the group record time index for the branch, and stores output records to a PyTables file, so that the groups can be read using random access, and ungrouped in the ungrouper stage. The time index is an index that maps time intervals to the records, which occur during this interval.

2.8 Merger

The merger is organized as nested branch loops. An output of the merger is an N-tuple of groups, where N is the number of branches. Every merger branch represents a `for`-loop over its records. Each branch loop reads a record from the corresponding record group and executes the matching rules which have their arguments in the current record group tuple. The branches are organized into a nested structure (by alphabetical order). After matching the tuple with its rule, the branch passes it to the lower level, which adds a record from its branch to the tuple and executes any further rules. One of the merger requirements is that at least one of the Allen operators must be present. In order to improve the efficiency of the merger operation, a time index is used to find only records which have the possibility of satisfying the Allen operators used in the merger. If there is an Allen relation $A < B$, the branch B loop does not need to iterate over all of its records for each record in A. It can iterate only over records that occur after the current record. In general, if the left argument of an Allen relation is known, it imposes a restriction on the possible right arguments.

2.9 Ungrouper

Ungrouper objects are used to ungroup merger output. Record group tuples are expanded into records. The order of the flow records is determined by the order of the groups that are expanded, while those records within a group are ordered by their record IDs.

3 Performance Evaluation

In our performance analysis we employed a Python profiler function written by Maciej Obarski [8], to achieve multi-threaded profiling of the Python program execution. The output of a profiler consists of three metrics: the total number of calls to each function and the total wall clock and system times spent between function call and function return. A sample output of a profiler for a `flush` method looks as follows:

```
(('flush','/usr/local/[... ]/tables/table.py',2408), (12, 0.02, 0.01))
```

This output displays the function name and the location in the source code it was initiated at, as well as the three other metrics we mentioned above. Establishing a common comparison metric out of these three is rather evident. Obviously, the number of function calls alone is not a good metric, since the cumulative time of these calls may be much less than the time spent on some other functions with less calls. Execution times experience a similar problem, due to calling other functions internally, which would mean that a certain function may be called only a few times, but result in a large execution time. The metric used for finding worst performing functions was time spent per function call or the average time per function call. This value has been calculated by dividing the wall clock time spent between function call and function return, divided by the total number of calls. We executed a simple two-branch Flowy query on a very short flow trace, as a baseline for comparison with other tools. The time it has taken to initiate, validate and process the query was ≈ 3 seconds (Intel(R) Pentium(R) 4 CPU 3.00GHz, 512MiB System Memory).

At the evaluation stage we employed several queries (simple source/destination ports or addresses extraction, queries with more than two branches, and so on) in order to gain more understanding of the processes that are happening at different stages. All the queries were executed on differently sized flow traces. The traces have been collected by regular users, who had the necessary exporting and capturing tools installed on their machines. The queries have been evaluated on the traces of $\approx 26K$, $\approx 57K$, $\approx 100K$ and $\approx 300K$ records. Here we present the profiling results of a single query, which was defined and discussed in Section 2.

The profiler has shown that the heaviest processing load was experienced in the filter, grouper and merger stages. The profiler results show that for all four traces the worst performing functions/methods were similar. In a run with $\approx 57K$ records we see that the functions coming from the filter, grouper and merger stages are among worst performers, as shown below:

```
(('reset', '/[...]/flowy/filter.py', 45), (56992, 255.76, 262.48))
(('match', '/[...]/flowy/merger.py', 23), (527995, 64.41, 62.12))
(('match', '/[...]/flowy/grouper.py', 126), (1740570, 498.78, 495.35))
```

Almost the same set of functions is performing the worst for other flow traces. We can see that two out of those are `match()` functions defined in the corresponding classes. Internally, the `match()` functions perform different tasks, for instance the `match()` of the grouper stage performs rule comparisons on the records, and the number of these comparisons for a trace of $\approx 57K$ records is 1740570. The `match()` function of the merger module also performs record comparisons based on the specified rules and with the use of Allen's relations. We can see that the number of `match()` comparisons is significantly smaller at the merger stage, since less records arrive at this stage than at the grouper stage. The `reset()` function of the filter stage internally performs deep copying for each record and is specified by the standard `deepcopy()` Python method. The `deepcopy()` operation is heavy in itself, since it needs to consider various data structures, but its use in Flowy grows linearly with the number of records.

The prevalence of the grouping operation time requirement increases with the number of records at each branch, and follows the performance trend of the two-branch merger up to a certain amount of branch records that passed the filter, approximately 1000 records. However, the number of records is not the only factor for decreasing performance. The number of rules at each of the groupers also influences the overall running time, since each record needs to be compared to more rules. A simple test has shown that an increase in the number of rules at each of the groupers by one, increases the running time of the grouper stage roughly by a factor of two (extra iteration over filtered records). The above statement may vary significantly based on many factors and conditions,

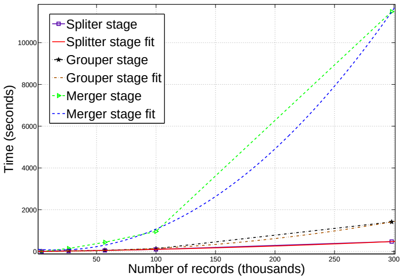

Fig. 2. Total records vs. run time

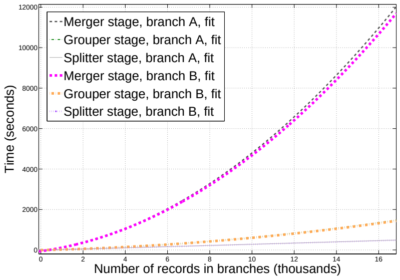

Fig. 3. Branch records vs. run time

of course. Very high grouper performance deterioration comes with small `delta` values. The grouper potentially needs to traverse through many more records in order to find a matching record that is "close enough" in time. The merger consumed a large share of the execution time compared to other stages. This became more evident with an increase of the number of filtered records at each branch. In general the running time requirements of the HTTP download query can be summarized by Fig. 2. It can easily be noticed that time requirements of the two-branch merger module dominates those of the other modules, especially evident with an increase in record numbers. Nevertheless, Fig. 2 is somewhat inaccurate, since if we executed another simple query (that makes use only of the filter stage) on the same flow traces, most of the time would be consumed in the filtering section, and not in the merger. For that reason we produced another plot in Fig. 3, which is "more objective" towards the stages that follow the filtering process, and shows the number of filtered records at each branch versus the running time. Similar to the grouper stage, the merger stage depends on the number of defined rules. The larger the number of rules that needs to be specified, the higher is the running time. Results presented in Figures 2 and 3 confirm the worst-case running times of the stages. From previous sections we are aware that both filter and group-filter perform absolute filtering of the records, and, therefore, directly depend on the number of records. The worst time of these modules is $O(n)$, where n is the number of input records or record groups.

Similarly, the ungrouper needs to iterate over the resulting record groups only ones, but with consideration that it needs to retrieve the original records, which could influence its performance depending on the storage and retrieval methods used.

The grouper stage needs to find a group match for each of the considered records. It selects one record and iterates through the rest of the records trying to find a match. In the worst case, each of the records will form a self-contained group and given that scenario, the program will need to iterate at most $O(n^2)$ times, where n is the number of input records.

The merger is considered last, and it is actually the most time-demanding stage. It directly depends on the number of branches present and the number of branches in each of the groups. It will iterate through all the groups of each of the branches trying to find a match, which would result in the complexity of $O(m^n)$ times, where n is the number of branches and m is the number of groups in each branch. As a simple confirmation of significant performance degradation of the merger stage, we evaluated a query with three branches. This caused the evaluation time to increase by a large factor.

The problem with performance is connected both to the structure of the program, i.e., an increase in the number of branches causes high overall complexity of the tool, and to the language the tool was implemented in. A Python implementation might provide a good start as a prototype tool, but not when it comes to real usage, with potential processing of a large number of records. Many standard methods of Python that are used in the program consider different cases (i.e., `deepcopy()`) and are, thus, not optimal. Rewriting those funtions, but optimized for the performed tasks could be a potential run-time improvement. As such, we redefined the `deepcopy()` method for a simple dictionary data set inside the `reset()` method of the filter module. This improved the runtime of that particular code piece by a factor of eight:

```
original - ((’reset’, ’/[...]/filter.py’, 45), (56992, 255.76, 262.48))
modified - ((’reset’, ’/[...]/filter.py’, 64), (56992, 31.72, 31.21))
```

Abandoning Python altogether, and switching to a language like C instead, could potentially improve the run time significantly.

4 Related Work

Existing flow query languages can be divided into three categories: filtering languages, procedural languages and SQL-based languages [9]. The Berkley Packet Filter (BPF) [10] allows to filter network traces by fields such as source/destination IP address and source/destination ports. The filtering mechanism consists of simple filter expressions converted into executable programs. Popular tools, like `tcpdump` and `nfdump`, are based on BPF expressions. The `CoralReef` network analysis tool [11] also uses BPF expressions for generating reports from collected trace files. Another well-known filtering language is the `flow-tools` suite [12]. It consists of several applications that allow to collect and analyze NetFlow data. Flow tools capture and filter flows, create reports based on different flow record fields, and display filtered or original flow records.

The Simple Rules Language (SRL) [13] is a procedural language for defining traffic flows. It is used to specify filtering rulesets that instruct flow meters about which traffic flows are of interest and which flow attributes are to be collected and stored on the meters. `FlowScan` [14] is another instance of a procedural language. It consists of a number of perl scripts that produce a flow collection tool. FlowScan can generate rather general and high-level traffic reports that could be helpful in detecting particular traffic patterns. The script-based language `SiLK`

[15] is a collection of commands for querying NetFlow data with its own filter expression primitives. SiLK allows to label a set of flows aggregated by a common attribute. The `rwgroup` application iterates over flow records and groups those that have common attributes. A possible use-case scenario would be the aggregation of flows belonging to an FTP session. The `rwmatch` application, on the other hand, creates matched groups that consist of an initial record, followed by many more. An example of `rwmatch`'s application could have been the HTTP download session from Section 2.

The last group of applications considered here is based on SQL. A system that uses standard MySQL and an Oracle DBMS for storing attributes of NetFlow records is described in [16]. Using SQL queries, the tool can provide strong support for basic intrusion detection and usage statistics. The queries could be as complex as establishing a list of IP addresses from external autonomous system that have contacted a large number of internal IP addresses. The Data Stream Management System (DSMS) [17] was an improvement over DBMS. DSMS supports some of the DBMS' missing features, i.e., modeling flows as transient data streams, as opposed to the persistent relational data model. Another example of SQL-based flow query language systems is `Gigascope` [18], a stream database for network monitoring applications that uses GSQL for query and filtering. GSQL is a modified version of SQL, which allows to define time windows inside the query. GSQL supports selection, aggregation, join and merge operations. A typical query is subdivided into low-level (preliminary filtering and simple aggregation) and high-level (possible BPF invocation and complex aggregation operations using data cube computation algorithms) processing parts.

5 Conclusion

A prototype implementation of a stream-based flow query language called Flowy has been described and evaluated by a number of profiling tests. The test results indicated certain bottlenecks when the number of flow input records increases substantially. We identified two primary reasons for the performance problems:

1. The implementation in a high-level language causes performance problems due to slow execution on critical code sections.
2. The growths of complexity with the number of branches and group records in the merger.

The implementation exploits concurrency by processing branches in different independent threads. Our current work explores a more coarse grained level of concurrency by applying the MapReduce framework [3] and distributing flow processing over multiple machines.

Acknowledgement

The work reported in this paper is supported by the EC IST-EMANICS Network of Excellence (#26854).

References

1. Vladislav Marinov and Jürgen Schönwälder. Design of a Stream-Based IP Flow Record Query Language. In *DSOM '09*, pages 15–28, Berlin, Heidelberg, 2009. Springer-Verlag.
2. James F. Allen. Maintaining Knowledge About Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.
3. Francesc Alted, Ivan Vilata, et al. PyTables: Hierarchical datasets in Python. *http://www.pytables.org/*, 2002–.
4. David M. Beazley. Ply (python lex-yacc). *http://www.dabeaz.com/ply/*, 2001–.
5. Mike Folk, Robert E. McGrath, and Kent Yang. Mapping HDF4 Objects to HDF5 Objects. Technical report, National center for supercomputing applications, University of Illinois, 2002.
6. Benoit Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954, Cisco Systems, October 2004.
7. M. Obarski. Profiling python threads. *http://code.activestate.com/recipes/465831/*, Available on-line: 01-02-2010.
8. Vladislav Marinov and Jürgen Schönwälder. Design of an IP Flow Record Query Language. In *AIMS '08*, pages 205–210, Berlin, Heidelberg, 2008. Springer-Verlag.
9. Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX '93*, pages 2–2, Berkeley, CA, USA, 1993. USENIX.
10. David Moore, Ken Keys, Ryan Koga, Edouard Lagache, and K. C. Claffy. The CoralReef Software Suite as a Tool for System and Network Administrators. In *LISA '01*, pages 133–144, Berkeley, CA, USA, 2001. USENIX.
11. Steve Romig. The OSU Flow-tools Package and CISCO NetFlow Logs. In *LISA '00*, pages 291–304, Berkeley, CA, USA, 2000. USENIX.
12. N. Brownlee. SRL: A Language for Describing Traffic Flows and Specifying Actions for Flow Groups. RFC 2723, University of Auckland, October 1999.
13. Dave Plonka. FlowScan: A Network Traffic Flow Reporting and Visualization Tool. In *LISA '00*, pages 305–318, Berkeley, CA, USA, 2000. USENIX.
14. CERT/NetSA at Carnegie Mellon University. SiLK (System for Internet-Level Knowledge). [Online]. Available: `http://tools.netsa.cert.org/silk`. [Accessed: July 13, 2009].
15. Bill Nickless. Combining Cisco NetFlow Exports with Relational Database Technology for Usage Statistics, Intrusion Detection, and Network Forensics. In *LISA '00*, pages 285–290, Berkeley, CA, USA, 2000. USENIX.
16. Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *PODS '02*, pages 1–16, New York, NY, USA, 2002. ACM.
17. Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: a Stream Database for Network Applications. In *SIGMOD '03*, pages 647–651, New York, NY, USA, 2003. ACM.
18. Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI '04*, pages 10–10, Berkeley, CA, USA, 2004. USENIX.