NFQL: The Swiss-Army Knife of Efficient Flow-Record Processing

Vaibhav Bajpai, Johannes Schauer, Jürgen Schönwälder

School of Electrical and Computer Science

Campus Ring 1, Jacobs University Bremen

{v.bajpai, j.schauer, j.schoenwaelder}@jacobs-university.de

Abstract—**Cisco's NetFlow protocol and IETF's IPFIX open standard have contributed heavily in pushing IP flow export as the de-facto technique for collecting aggregate network traffic statistics. These flow records have the potential to be used for billing and mediation, bandwidth provisioning, detecting malicious attacks and network performance evaluation. However, understanding certain traffic patterns requires sophisticated flow analysis tools that can mine flow records for such a usage. We recently proposed a flow query language that can cap such flow-records. In this paper, we introduce Network Flow Query Language (NFQL). an efficient implementation of the query language. NFQL can process flow records, aggregate them into groups, apply absolute (or relative) filters, invoke Allen interval algebra rules, and merge group records. The implementation has been evaluated by suite of benchmarks against contemporary flow-processing tools.**

I. Introduction

Researchers, service providers and security analysts have long been interested in network and user behavioral patterns of the traffic crossing the internet backbone. They want to use this information for the purpose of billing and mediation, bandwidth provisioning, detecting malicious attacks, network performance evaluation and overall improvement. Traffic measurement techniques that have been rapidly evolving in the last decade, have matured enough today to provide such an insight.

Flow capture today, has emerged out to be one of the favored network measurement techniques. This has largely been due to the reduction in the monitoring traffic at the flow-level and the fine-grained control which was not previously possible using SNMP interface-level queries. As a result, each networking vendor has tried to come up with a standard protocol that defines the semantics of this flow export. In this pursuit, Cisco eventually managed to make their proprietary protocol so ubiquitously available, that the next-generation universal standard is based on it.

II. Related Work

`tcpdump` and `wireshark` are the most popular tools used for packet capture and analysis. `tcpdump` [2] is a premier command-line utility that uses the `libpcap` [3] library for packet capture. The power of `tcpdump` comes from the richness of its expressions, the ability to combine them using logical connectives and extract specific portions of a packet using filters. `wireshark` [4] is a GUI application, aimed at both journeymen and packet analysis experts. It supports a large number of protocols, has a straightforward layout, excellent documentation, and can run on all major operating systems.

`flow-tools` [5] is a suite of programs for capturing and processing NetFlow v5 flow records. It consists of 24 separate tools that work together by connecting them via UNIX pipes.

`nfdump` [6] works similar to flow-tools but uses a different storage format. Flow records are captured using `nfcapd` and then processed by `nfdump` which can filter as well as display the sorted and filtered result. The power of its filtering rules is similar to that of flow-tools and as such is mostly limited to absolute comparisons of flow attributes.

SiLK[1] is a network traffic collection and analysis tool developed and maintained by the CERT Network Situational Awareness Team (CERT NetSA) at Carnegie Mellon University. SiLK is the tool that comes quite close to providing similar capabilities as provided by NFQL and is therefore used as a reference point to compare the performance of the NFQL execution engine in this paper. The design and implementation of SiLK, however, differs a lot from that of NFQL.For instance, in SiLK there are separate tools to perform the task of each stage of the NFQL processing pipeline. The stage functionality is not full-fledged though. The grouping and merging operations can only be performed using an equality operator. This is assumed in the tool, thereby allowing it to perform optimization such as using hash tables to perform lookups. There are also stringent requirements to how flow-data needs to be organized before it can be piped into a tool. The grouping tool, for instance, assumes that the to-be supplied input flow data is already sorted on the field column. These requirements make it a little cumbersome to design a full-fledged flow query. For instance, trying to mimic a NFQL query in SiLK ends up as a bash script with over a dozen of SiLK tools piped together.

III. Flow Query Language

The pipeline consists of a number of independent processing elements that are connected to one another using UNIX-based pipes. Each element receives the content from the previous pipe, performs an operation and pushes it to the next element in the pipeline. Fig. 1 shows an overview of the processing
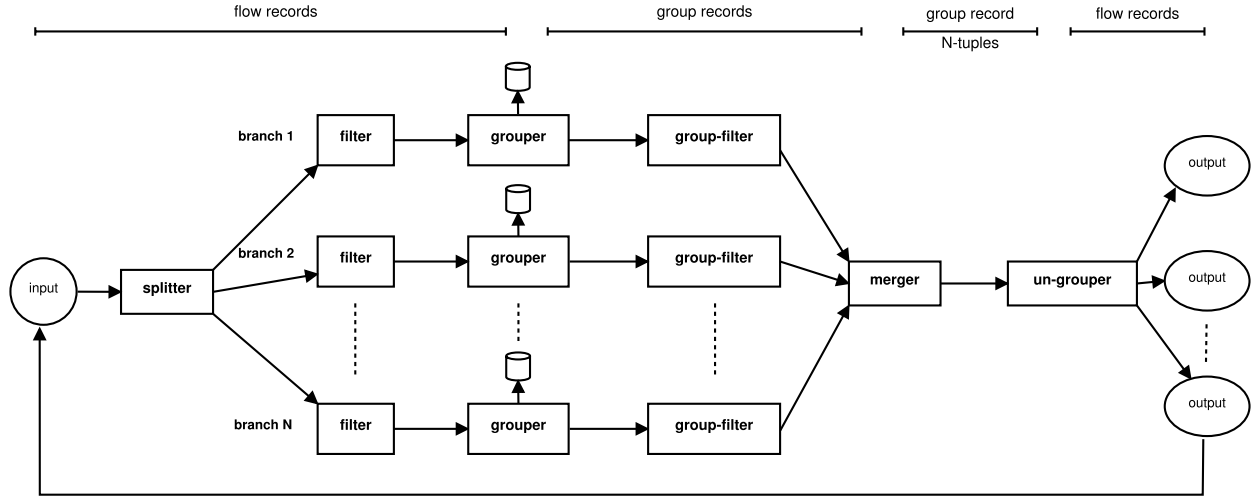
---

[1]http://tools.netsa.cert.org/silk/

Fig. 1. NFQL Pipeline [1]

pipeline. A complete description on the semantics of each element in the pipeline can be found in [1]

The splitter takes the flow-records data as input in the `flow-tools` compatible format. It is responsible to duplicate the input data out to several branches without any processing whatsoever. This allows each of the branches to have an identical copy of the flow data to process it independently.

The filter performs *absolute* filtering on the input flow-records data. The flow-records that pass the filtering criterion are forwarded to the grouper, the rest of the flow-records are dropped. The filter compares separate fields of a flow-record against either a constant value or a value on a different field of the *same* flow-record. The filter cannot *relatively* compare two different incoming flow-records

The grouper performs aggregation of the input flow-records data. It consists of a number of rule modules that correspond to a specific subgroup. A flow-record in order to be a part of the group should be a part of at-least one subgroup. A flow-record can be a part of multiple subgroups within a group. A flow-record cannot be part of multiple groups. The grouping rules can be either absolute or relative. The newly formed groups which are passed on to the group filter can also contain meta-information about the flow-records contained within the group using the aggregate clause defined as part of the grouper query.

The group-filter performs *absolute* filtering on the input group-records data. The group-records that pass the filtering criterion are forwarded to the merger, the rest of the group-records are dropped. The group-filter compares separate fields (or aggregated fields) of a flow-record against either a constant value or a value on a different field of the *same* flow-record. The group-filter cannot *relatively* compare two different incoming group-records

The merger performs relative filtering on the N-tuples of groups formed from the N stream of groups passed on from the group-filter as input. The merger rule module consists of a number of a submodules, where the output of the merger is the set difference of the output of the first submodule with the union of the output of the rest of the submodules. The relative filtering on the groups are applied to express timing and concurrency constraints using Allen interval algebra [7]

The ungrouper unwraps the tuples of group-records into individual flow-records, ordered by their timestamps. The duplicate flow-records appearing from several group-records are eliminated and are sent as output only once.

IV. IMPLEMENTATION

A. *Flow Query Intermediate Format*

The NFQL execution engine reads the flow query at runtime in a `JSON` intermediate format. The entire flow query is a collection of Disjunctive Normal Form (DNF) expression. A DNF expression is a disjunction of conjunctive clauses. The elements of the conjuctive clauses are terms. The clauses in the DNF are OR'd together, while the terms in each clauses themselves are AND'd. The branchsets and each DNF expression of the pipeline stage is a `JSON` `array`. `json-c`[2] is used to parse the flow query file, and python scripts have been written to quickly serialize the python pipeline objects to a `JSON` query.

The `JSON` query can also trigger and disable the stages at runtime. This means that one only has to supply the constructs that one wishes to use. The constructs that are not defined in the `JSON` query are inferred by the execution engine as a disable request. The execution engine uses disable flags that are turned on when the `JSON` query is parsed. These flags

[2]http://oss.metaparadigm.com/json-c/

are used throughout the engine to only enable the requested functionality.

The mapping of the `JSON` query to the structs defined in the execution engine is not trivial. When reading the `JSON` query at runtime, the field offsets of the NetFlow $v5$ record struct are read in as strings. Utility functions are defined that maps the read names to struct offsets and read type of each offset and the operations to unique `enum` members.

The abstract objects that store the JSON query and the results that incubate from each stage are designed to be self-descriptive and hierarchically chainable. The complete JSON query information for instance, is held within the `flowquery` struct. Each individual branch of the flowquery itself is described in a `branch` struct. A collection of these `branch` structs are referenced in the parent `flowquery` struct.

B. Execution Workflow

1) Splitter: A custom C library was written to directly read/write data in the `flow-tools` format. The library sequentially reads the complete flow-records into memory to support random access required for relative filtering. Each flow-record is stored in a `char` array and the offsets to each field are stored in separate `structs`. The array of such records are indexed allowing fast retrieval in $O(1)$ time. This allowed the flow-records to be easily referenced by an identifier, thereby giving away the need to every time copy all the flow-records when moving ahead in the processing pipeline.

In the default method, there is a comparison function defined for every possible field length (33) and comparison operations (19). These functions are generated using a Python script. The rule definitions are now able to make calls using a function name derived from the combination of field length, delta type and operation. This subverts the need to define complex branching statements and reduces complexity.
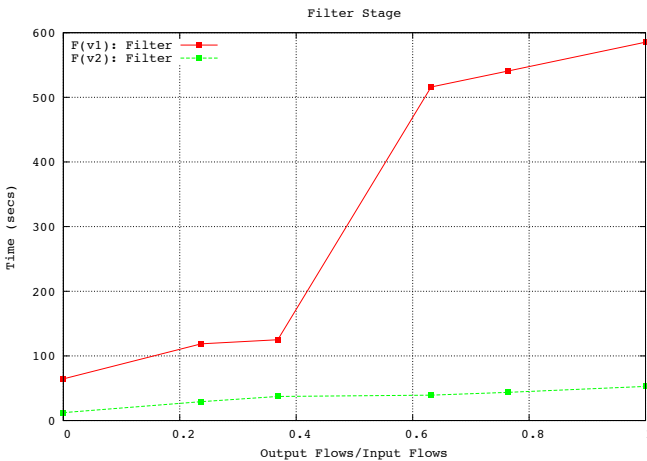

Fig. 2. Filter Stage: F($v1$) vs F($v2$)

2) Filter: The flow query language suggests to read all the flow records of a supplied trace into memory before starting the processing pipeline. Since, the filter stage uses the supplied set of absolute rules to make a decision on whether or not to keep a flow record; it has to pass through the whole in-memory recordset *again* to fill in the filter results. This technique involves multiple linear runs on the trace and therefore slows down when the ratio of number of filtered records to the total number of flow-records is high. The NFQL implementation merges the filter stage with in-memory read of the trace. This means, a decision on whether or not to make room for a record in memory and eventually hold a pointer for it in filter results is done upfront as soon as the record is read from the trace. In addition, if a request to write the filter stage results to a flow-tools file has been made, the writes are also made as soon the filter stage decision is available, thereby allowing reading-filtering-writing to happen in $O(n)$ time, where n is the number of records in the trace. It is important to note that the filtered records are saved in common location from where they are referenced by each branch. This helps keep the memory costs at a minimum when multiple branches are involved. A publically available Netflow $v5$ trace [3] was used to compare the performance of the new filter in F($v2$) with that from F($v1$) as shown in Fig. 2. The trace has around $20M$ flow-records.

3) Grouper: In order to be able to make comparisons on field offsets, a simple approach is to linearly walk through each filtered record against the filtered recordset leading to a complexity of $O(n^2)$, where n is the number of filtered records. A smarter approach is to put the copy in a hash table and then try to map each pointer while walking down the filtered recordset once, leading to a complexity of $O(n)$. The hash table approach, although will work on this specific example, will fail badly on other relative comparisons. A better approach, as implemented in F($v2$) is to sort the filtered recordset on all the requested grouping rules. This helps the execution engine perform a nested `bsearch` to reduce the linear pass to a fairly small filtered recordset. However, the number of elements upon which the search has to be performed needs to be known at each level of the binary lookup. In order to avoid a linear run to count the number of elements, the parent level `bsearch` invocation returns a boundary with the first and last element to enable such a calculation. This helps the grouper perform faster search lookups to find records that must group together in $O(n * lg(k))$ time with a preprocessing step taking $O(n * lg(n)) + O(n)$ in the average case, where n is the number of filtered records and k is the number of unique filtered records.

The grouping approach has further been optimized when the filtered records are grouped for equality. In such a scenario, the need to search for unique records and a subsequent binary search goes away. The groups can now be formed in $O(n)$ time with a more involved preprocessing step taking $O(p * n * lg(n))$ where is n is the number of filtered records, and p is the number of grouping rules. The performance evaluation of the grouper handling this special case against its behavior when handling generic cases is shown in Fig. 3. Profiling the
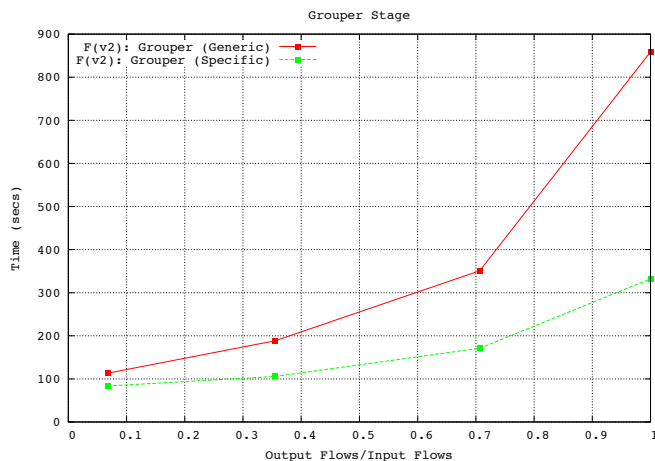
---

[3]http://traces.simpleweb.org/traces/netflow/netflow1/netflow000.tar.bz2

Fig. 3. Grouper Stage: F($v2$) (Generic) vs F($v2$) (Specific)

performance of the grouper on queries that produce higher ratios reveal that a significant amount of time is taken in `bsearch`. The special case of equality comparisons eliminate these calls and further reduce the time in the long tail as shown in Fig. 3.

4) Group Aggregations: Group records are a conglomeration of several flow records with some common characteristics defined by the flow query. Some of the non-common characteristics can also be aggregated into a single value using group aggregations. It is useful if a single group record can be again mapped into a NetFlow $v5$ record template, so that it can be written to file as a representative of all its members.

5) Group Filter: The groupfilter is used to filter the groups produced by the grouper based on some absolute rules defined in a DNF expression. The `struct` term holds information about the flow record offset, the value being compared to and the operator which maps to a unique `enum` value. This enum value is used to map the operation to a specific group-filter function. The group-filter functions are auto-generated using a python script.

6) Merger: The merger is used to relate groups from different branches according to a merging criterean. The implementation is not trivial since the number of branches that need to be spawned is read from the query and is not known until compile time. As a result, an iterator that can provide all possible permutations of $m-$tuple (where m is the number of branches) group record IDs was needed. The result of the iterator can then be used to make a match. The merger stage, begins by initializing this iterator passing it the number of branches, and information about each branch. Then, it loops over to get a new $m-$tuple of group record IDs on each iteration until the iterator returns `false`.

The merger as formulated in the flow query language needs to match each group record from one branch with every other record of each branch. This leads to a complexity of $O(n^m)$ where n is the number of filtered group records and m is the number of branches. The possible number of tries when

matching group records however can be reduced by sorting the group records on the field offsets used for a match. The NFQL implementation optimizes the merger to skip over iterator permutations when a state of a current field offset value may not allow any further match beyond the index in the current branch. For such an optimization to work, the filtered group records must be sorted in the order of field offsets specified in the merger clause. Specifying the filtered group records in any other order may lead to undefined behavior. This means, that if the same field offsets were used in the grouper stage, the terms in the group clause can be rearranged by the query designer to align with the order of terms in the merger clause.

The flow query language also bases the merger matches on the notion of matched tuples. This means that a filtered group record can be written to a file multiple times if it is part of multiple matched tuples. This situation is very common and it worsens when different branches have similar filtered groups records. Since, the function of the merger is to find a match of groups records across branches based on a predefined condition, all the group records across branches that satisfy the condition can be clubbed into one collection instead of separate tuples. All the group records within a collection can then be written to the file. This eliminates the inherent redundancy and significantly improves the merger performance.

The performance comparison of this approach against the one suggested by the specification is tricky. The merger implementation of the original specification is slow. It is so slow that it keeps churning the CPU for days without results. The newer approach takes less than an hour. The performance evaluation of this newer approach is discussed in more details in the next section

7) Ungrouper: The approach of clubbing the merged group records into a collection incurs a reimplementation of the ungrouper. The ungrouper, as a result accepts a collection of matched filtered group records as input. It then iterates over each collection to unfold it groups and write their flow record members to files.

C. Adaptable Compression Levels

The flow-records echoed to the standard output can also be written to a Netflow $v5$ flow-tools file. The `-dirpath` switch allows one to provide a directory path where the results can be stored. Each stream is stored as its own file with an ID to disambiguate it. Results from each stage of the pipeline can also be written to separate files with the increase in the verbosity level. In fact, `-dirpath` and `-verbose` work well together to adjust the level at which the writes are to be made.

The engine uses the `zlib` [8] software library to compress the results written to the flow-tools files. `zlib` supports 9 compression levels with 9 being the highest. The NFQL engine supports `-zlevel` switch to allow the user to supply its desired choice of the of the of the compression level. A default level of 5 is used for writes if the switch is not supplied during runtime. Fig. 4 shows the time taken to write a sample of records passing the filter stage for each `z-level`. It goes to show
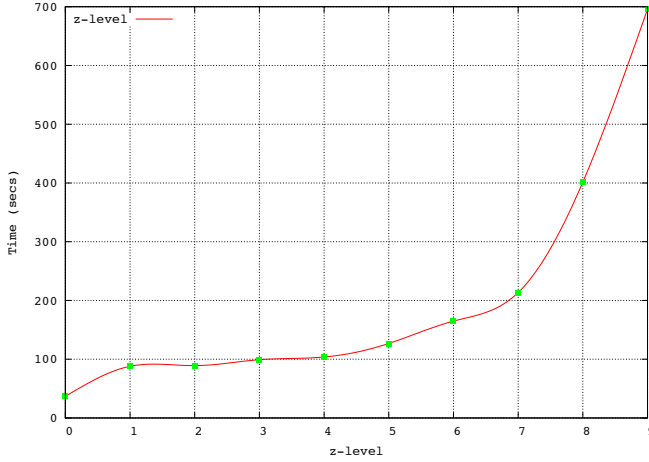
Fig. 4.   F($v2$): `z-level` Effect on Performance

that each level adds its own performance overhead and must be used with discretion.

D. Performance Optimizations

There can be a situation where the query designer may incorrectly ask for aggregation on a field already specified in a grouper (or filter) clause. If the relative operator is an equality comparison, the aggregation on such a field becomes less useful, since the members of the grouped record will always have the same value for that field. NFQL realizes this redundant request and ignores such aggregations.

NFQL has dedicated comparator functions for each type of operation and the type of the field offset it operates upon. It is not guaranteed that given the type of the query and trace, the program will eventually go through each stage of the pipeline. It is also possible that the program exits before, because there is nothing more for the next stage to compute. The function pointers are therefore set as late as possible and are are called from their respective stages just before the comparison is needed. As a result, we save the computation time wasted in setting the function pointer for stage if it is is never executed.

NFQL pushes the filter stage out of the branch and closer to where the trace is originally read. This enables the memory allocation of only records that passed the filter stage thereby reducing the runtime memory footprint of the execution engine.

Each branch runs in its own thread. If any of the stages of the branch return a `NULL` when returning from their public interface function, there is no reason to continue the thread. The subsequent stages of the branch cannot do much with a `NULL` result. Therefore, the branch thread returns with a `EXIT_FAILURE` if either stage returns `NULL`, and with `EXIT_SUCCESS` on normal exit.

Each stage of the processing pipeline is dependent on the result of the previous one. As a result, the stages should only proceed and process, when the previous returned results. Implementing such a response was straightforward for the grouper and group filter, the merger although was a little trickier. The merger stage proceeds only when every branch has non-zero filtered groups. The iterator initializer `iter_init(...)` deallocates and returns `NULL` if any one branch has 0 filtered groups. Consequently a check is performed in the merger to make sure `iter` is *not* `NULL`.

The results from each stage of the pipeline are echoed to the standard output just before the execution engine exits. This leads to an additional loop to echo the results, however echoes to the standard output are only used for debugging purposes. The writes to a file can however be legitimately requested in a real network analysis task. As a result, it is essential to avoid additional loops when performing writes to a file. The execution engine, therefore writes each result record to a file as soon as it is seen by the pipeline stage.

V. Performance Evaluation

We used a public flow trace containing $20M$ records to run a number of flow queries using our benchmarking suite. We used trace 7 [4], from Simpleweb [5]. The input trace was compressed at `ZLIB_LEVEL` 5 using the `zlib` suite. It was also converted to `nfdump` and SiLK compatible formats and compressed with `zlib` keeping the same compression level. The suite was run on a high-end machine[6] with 24 cores of 2.5 GHz clock speed and 18 GiB of memory.

The first set of queries attempt to stress the filter stage. We use varying values on the packet field offset to determine the amount of flow records that are passed by the filter. The resultant filtered records are written to `flow-tools` compatible file format and compressed at `ZLIB_LEVEL` 5 using `zlib` suite. The ratio of the number of filtered records in the output trace to the number of the flow records in the input trace is plotted against time. The evaluation results are shown in Fig. 5.
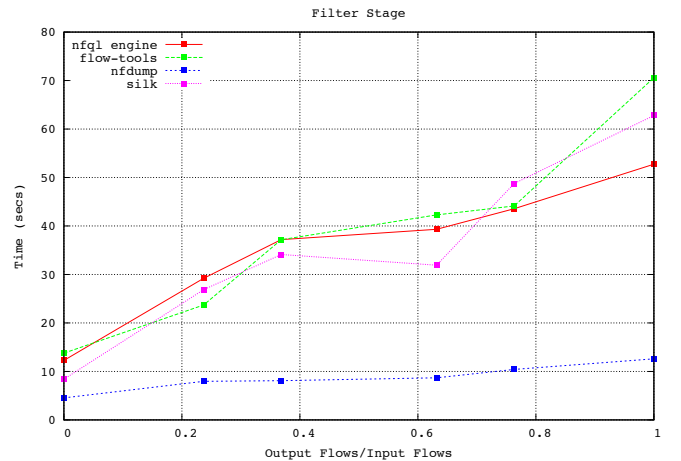

Fig. 5.   Filter Stage: NFQL vs SiLK, Flow-Tools, Nfdump

[4]http://traces.simpleweb.org/traces/netflow/netflow1/netflow000.tar.bz2
[5]Simpleweb is a data repository of traffic traces from University of Twente
[6]crystal.eecs.jacobs-university.de

It can be seen that the performance of the filter stage in NFQL is comparable to that of `flowtools` and SiLK. SiLK takes less time on lower ratios, but then again SiLK and `nfdump` also use their own proprietary format for trace files. As a result, the amount of data that needs to read (or written) may be different to what it is for NFQL and `flowtools`. On the other hand, `nfdump` appears to be significantly faster than the rest. This is because `nfdump` lacks `zlib` support, and as such the files that are read and written used `lzo` compression scheme which trades space for achieving faster compression and decompression. It is important to note, that all the tools were single-threaded in this evaluation, and did not completely utilize the 24 cores that were at their disposal. It comes as a realization, that filtering the input using multiple threads by memory mapping the trace and adding `lzo` compression will drastically improve NFQL's filter performance.

The second set of queries attempt to stress the grouper stage. We reuse the filter query that produces a 1.0 ratio to allow the grouper to receive the entire trace as a filtered recordset. The grouper part of the query then gradually increases the number of grouping terms in the DNF expression to increase the output/input ratio. The resultant groups are again written as `flowtools` files using the same `zlib` compression level. The ratio of the number of groups formed to the number of the input filtered records is plotted against time. `nfdump` and `flowtools` do not support grouping, and therefore are not considered in this evaluation. The evaluation results are shown in Fig. 6.
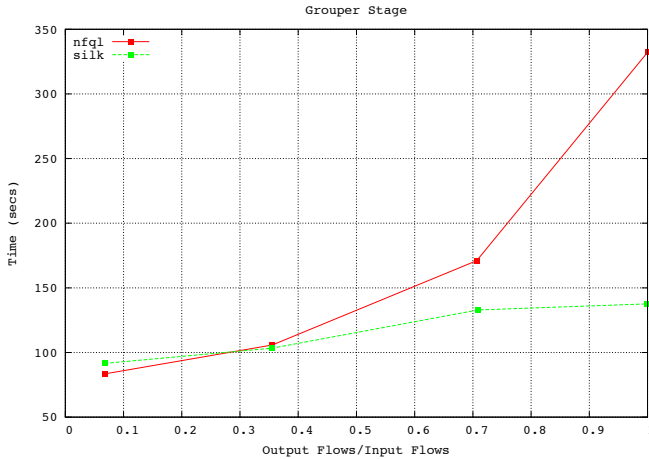

Fig. 6. Grouper Stage: NFQL vs SiLK

The evaluation graph reveals that the performance of the NFQL grouper stage is close. The time taken by the tools are comparable on lower ratios, but on higher ratios, NFQL starts to drift apart. Since most of the time is taken in writing the records to files, it is unclear whether SiLK's usage of a proprietary format which may reduce reads/writes is responsible for the drift on higher ratios. SiLK's `rwgroup` tool is also supplied a `-summarize` flag in all the queries. This gives SiLK the leverage to not store information about

which members are part of the group. NFQL on the other hand needs to allocate resources (which may take time) to keep this information in its data structures, since the ungrouper later may request to write the members of a group while unfolding the tuples. It is also important to note that both the tools again remained single-threaded throughout the evaluation. SiLK took an advantage of an inherent concurrency arising from how the query is structured as one single `bash` script using pipes. The pipe between `rwsort` and `rwgroup` makes the two process run concurrently, the effect of which gets more pronounced on higher ratios and can be a drift determining factor. The profiling results from GNU `gprof` [9] indicate that 60% of the time is taken in `qsort` comparator calls. As a result, it comes as no surprise, that bifurcating `qsort` invocation to multiple threads and later merging the results back using merge sort will help parallelize the grouper stage and maybe reduce the drift on higher ratios. In addition, since all of the evaluation queries had grouping terms using an equality comparator, NFQL can introspect such a grouping rule to dynamically optimize processing searches using a hashtable and turn to `qsort` based grouping only as a fallback.

The third set of queries attempt to stress the group filter stage. We reuse the filter and grouper queries that produce a 1.0 ratio to allow the group filter to receive the entire trace as input. This means, each flow record of the original trace now becomes a group record for the group filter. The group filter then reuses the same varying values of the packet offset to determine the amount of groups that are filtered ahead. The filtered groups are again written as `flowtools` files using the same `zlib` compression level. The ratio of the number of filtered groups formed to the number of the input group records is plotted against time. The evaluation results are shown in Fig. 7.
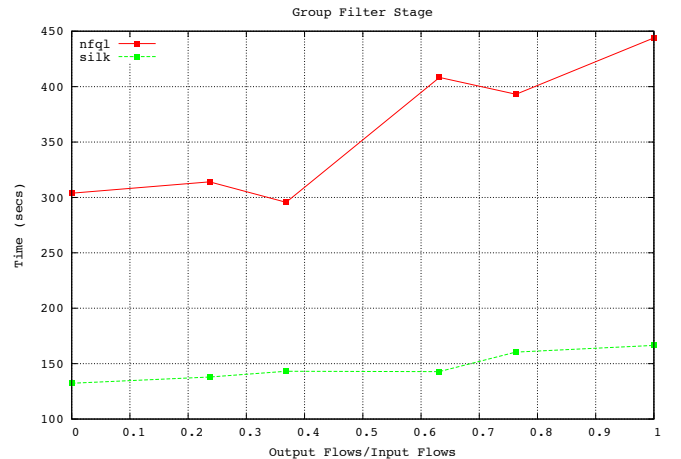

Fig. 7. Group Filter Stage: NFQL vs SiLK

It can be seen that the timings of NFQL are far apart from that of SiLK. It is due to the drift already created by the grouper at 1.0 ratio in the previous stage. As a result, the group filter comes into play only after 300 seconds, whereas

SiLK's group filtering already starts just below 150 seconds. Even if we normalize the graph, it can be observed that the group filter has a significantly higher slope. This is because it is only executed once the grouper returns, and therefore has to reiterate the groups to make a filtering decision.

The fourth set of queries attempt to stress the merger stage. We reuse the filter, grouper and group filter queries that produce a 1.0 ratio. These queries are then run in two separate branches to produce identical filtered group records. The merger then applies match rules to produce different output to input ratios. The groups that are merged are again written as `flowtools` files using the same `zlib` compression level. The ratio of the number of merged groups to twice[7] the number of flow records in the original trace is plotted against time. A data point for SiLK for the 0.2 ratio is not available since the NFQL query executed at that data point uses `OR` expressions which are not supported by SiLK. As a result, an equivalent SiLK query is not formulated.
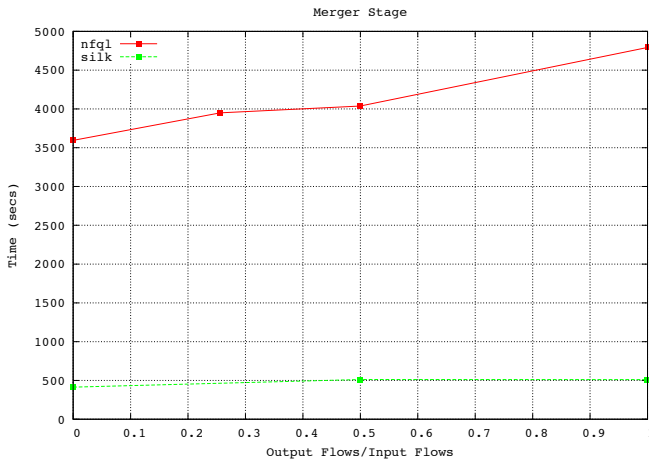

Fig. 8.   Merger Stage

It can be seen that the merger is the most performance hit and time consuming stage of the NFQL pipeline thus far. It is due to the fact that the merger is working on twice the number of flow records than any other previous stage. In addition, each branch is writing the results of the filter, grouper and group filter stage to `flowtools` files. As a result, the amount of disk I/O involved is twice as much as well. Even though each branch is delegated to a separate core using affinity masks, most of time is taken in writing these results to the file. These results although look less promising, they are way better than the previous merger implementation. The newer merger takes advantage of sorted nature of filtered groups and therefore can signifantly reduce the number of merger matches. It also writes a merged group record to file only once despite the number of times it has matched. Without these optimizations, running such queries on the merger would keep the CPU churning for days without results.

---

[7]Each branch pushes the entire trace as an input to the merger.

The last set of queries attempt to stress the ungrouper stage. They reuse the entire merger queries as is, but enable the ungrouper now as well. This means, that the ungrouper now attempts to unfold the merged groups returned by the merger to write their member flow records to `flowtools` files. However, since the merger receives each flow record as its own filtered group, each merged group has only one member. As a result, the ungrouper ends up executing its logic and rewriting the merged groups as `flowtools` files using the using the same `zlib` compression level. The ratio of the number of result flow records to twice the number of the flow records in the input trace is plotted against time. The evaluation results are shown in Fig. 9.
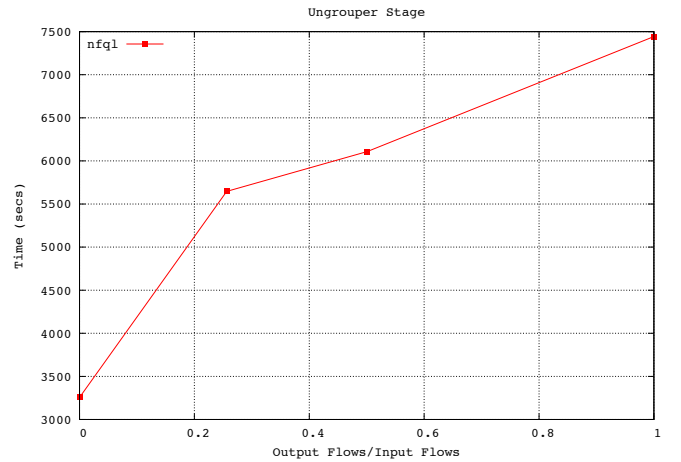

Fig. 9.   Ungrouper Stage

It can be seen that the evaluation behavior is similar to that shown by the merger, but is just more involved. This is because the ungrouper also has to write the entire merged groupset to file, as is done by merger. This is the reason for the execution engine in taking twice the amount of time. In fact, this evaluation is performed only to stress the functioning of the ungrouper and running queries that merge each filtered group filtered individually is less useful in practise. Ideally, the execution engine must shutdown the ungrouper if the merger merges every filtered group record in its output. This is because, such a behavior implicitly eliminates the need of the ungrouper.

VI. CONCLUSION

We presented NFQL, an efficient C implementation of the stream-based flow query language. The language allows applying absolute (or relative) filters, aggregating flows into groups, evaluating intricate timing relationships among them, and merging them into one collection. NFQL can execute such complex query tasks in matter of minutes, thereby expanding the scope of current flow record processing tools. The conducted performance evaluations reveal that NFQL is on par with tools that support only absolute filters. SiLK, the only package that provides tools that are similar to the rest

of NFQL's processing pipeline appears faster. This is because it can optimize its operations in favor of the limited set of comparisons that are only based on equality, and its usage of a proprietary file storage format.

REFERENCES

[1] V. Marinov and J. Schönwälder, "Design of a Stream-Based IP Flow Record Query Language," in *Proceedings of the 20th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management: Integrated Management of Systems, Services, Processes and People in IT*, ser. DSOM '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 15–28. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04989-7_2

[2] V. Jacobson, C. Leres, and S. McCanne, *tcpdump - dump traffic on a network*, Lawrence Berkeley National Laboratory, University of California, Berkeley, CA.

[3] ——, *pcap - Packet Capture library*, Lawrence Berkeley National Laboratory, University of California, Berkeley, CA.

[4] G. Combs, *wireshark - Interactively dump and analyze network traffic*, University of Missouri, Kansas City.

[5] S. Romig, "The OSU Flow-tools Package and CISCO NetFlow Logs," in *Proceedings of the 14th USENIX conference on System administration*. Berkeley, CA, USA: USENIX Association, 2000, pp. 291–304. [Online]. Available: http://dl.acm.org/citation.cfm?id=1045502.1045521

[6] P. Haag, "Netflow Tools NfSen and NFDUMP," in *Proceedings of the 18th Annual FIRST conference*, 2006.

[7] J. F. Allen, "Maintaining knowledge about temporal intervals," *Communications of the ACM*, vol. 26, pp. 832–843, November 1983. [Online]. Available: http://doi.acm.org/10.1145/182.358434

[8] P. Deutsch and J.-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3," RFC 1950 (Informational), Internet Engineering Task Force, May 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1950.txt

[9] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, Jun. 1982. [Online]. Available: http://doi.acm.org/10.1145/872726.806987