NFQL: A Tool for Querying Network Flow Records

Vaibhav Bajpai, Johannes Schauer, Jürgen Schönwälder
Computer Science, Jacobs University Bremen, Germany
{v.bajpai, j.schauer, j.schoenwaelder}@jacobs-university.de

Abstract—**Cisco's NetFlow protocol and IETF's IPFIX open standard are the widely deployed techniques for collecting network flow statistics. Understanding certain patterns in these network statistics requires sophisticated flow analysis tools that can efficiently mine flow records. We recently proposed Network Flow Query Language (NFQL) that can process flow records, aggregate them into groups, apply absolute or relative filters, and invoke Allen interval algebra rules to merge group records. In this paper, we introduce an efficient implementation of the query language. It has been evaluated by a suite of benchmarks against contemporary flow-processing tools.**

I. Introduction

NetFlow and Internet Protocol Flow Information Export (IPFIX) are the two popular protocols for IP flow information export. NetFlow [1] is a network protocol designed by Cisco Systems, which allows routers to generate and export flow records to a designated collector. IPFIX [2] is an open standard defined by the IETF based on NetFlow v9. The wide applicability of this approach is easily seen from the pervasive use of flow records for a set of different network analysis applications. For instance, a survey by Sperotto *et al.* [3] gives an overview of how network flow analysis can be used to detect intrusion attacks. In addition, a survey by Callado *et al.* [4] lists behaviour analysis of Internet backbone traffic and general anomaly detection.

Understanding intricate traffic patterns require sophisticated flow analysis tools that can mine flow records for such a usage. Unfortunately current tools fail to deliver owing to their language design and simplistic filtering methods. We recently proposed a flow query language design [5] that aims to cater to such needs. It can process flow records, aggregate them into groups, apply absolute or relative filters and invoke Allen interval algebra rules [6] to merge group records. The expressiveness of the language can be seen from [7] where the authors formulate flow queries to identify flow signatures of popular applications.

Flowy [8] was a first feature complete Python prototype of NFQL. Due to performance problems, its execution engine was rewritten in C, called Flowy 2.0 [9]. In this paper, we introduce `nfql`, which extends Flowy 2.0, making it more feature complete and optimizing its execution engine with crispier algorithms. We show that this iteration of our work is able to scale to real-world sized traces and has comparable execution times to contemporary flow analysis tools.

The paper is organized as follows. In Section II we survey the current state-of-the-art of flow-processing tools and we reason how `nfql` is different from each one of them. In Section III we describe the flow query language by discussing each stage of the processing pipeline. In Section IV we provide an overview of the `nfql` architecture and we describe the workflow of the execution engine. A performance evaluations comparing `nfql` against contemporary flow processing tools alongwith with a discussion on its current limitation and future outlook is in section V. We conclude the paper in Section VI.

II. Related Work

In recent years, a number of tools have been developed that can capture the traffic as flow records and use them for network analysis. Simple analysis of network traffic can be done by a range of graphical utilities like ntop [10], FlowScan [11], NfSen [12] and Stager [13]. All these tools understand the NetFlow format while ntop and Stager can also process IPFIX. `flow-tools` and `nfdump` are among the most popular tools used for analyzing NetFlow data. `flow-tools` [14] 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. It can capture, read, filter, and print flow records internally saved in a fixed-size format. `nfdump` [12] is a very similar tool that uses a different storage format. The power of filtering rules in both the tools is however mostly limited to absolute comparisons of flow attributes. As a result, relative comparison amongst different flows or querying a timing relationship among them is not possible.

SiLK [15] is a network traffic collection and analysis tool that comes quite close to providing similar capabilities as NFQL and is therefore used as a reference point to compare the performance of the `nfql` execution engine. 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 restriction allows the tool to perform optimization such as using hash tables to perform lookups. NFQL on the other hand, provides a much richer set of comparisons operations, such as *equal, not equal, greater than, less than, greater or equal, less or equal*. There are also stringent requirements in SiLK on how the 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 can make it a little cumbersome to design a full-fledged flow query. For instance, trying to mimic a NFQL query in SiLK sometimes ends up as a shell script with over a dozen of SiLK tools piped together.

III. Flow Query Language

The flow query language consists of a number of independent stages that are connected to one another to form a processing pipeline as shown in Fig. 1. A complete description
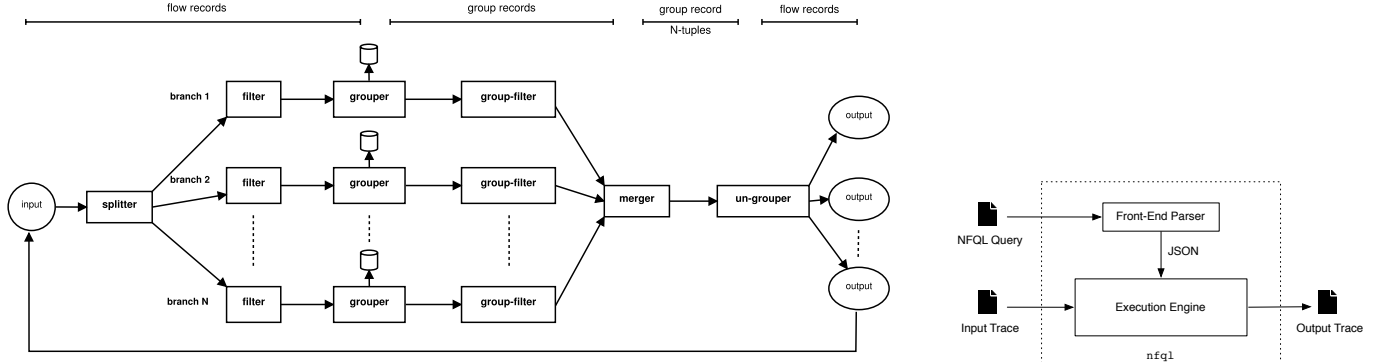
Fig. 1.   NFQL Processing Pipeline [5] (left) and `nfql` Architecture (right)

of the semantics of each pipeline stage can be found in [5]. The pipeline starts with the splitter. It takes the flow-records data as input in `flow-tools` compatible format. The splitter is responsible to duplicate the input data out to several branches without any processing whatsoever. This allows each of the branches to receive an identical copy of the flow data to process it independently.

The filter is the first processing stage of a branch in the pipeline. It 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 grouper performs aggregation of the input flow-records. The grouping terms 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 possible aggregation operations are *static, count, product, sum, logical and/or/xor, arithmetic mean, standard deviation, union, median, minimum and maximum*. The group-filter is the last processing stage of a branch. It performs *absolute* filtering on group-records. 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 group-record against either a constant value or a value on a different field of the *same* group-record. 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 relative filtering on the groups are applied to express timing and concurrency constraints using Allen interval algebra [6]. The ungrouper is the final processing stage. It unwraps the tuples of group-records into individual flow-records, ordered by their timestamps. Any duplicate flow-records appearing from several group-records are eliminated.

IV. IMPLEMENTATION

The `nfql` architecture primarily consists of a front-end parser backed up by an execution engine as shown in Fig. 1. The execution engine is the brain of `nfql` where the complete

pipeline is processed. It receives the flow query at runtime using a JSON [16] intermediate format. The execution engine is written in C, however the front-end parser can be written in any desired language.

A. Flow Query Intermediate Format

Each stage of the pipeline is expressed in the JSON query as a Disjunctive Normal Form (DNF) expression. The elements of the conjuctive clauses are terms. `json-c` [17] is used to parse the flow query file. The mapping of the query to the structs defined in the execution engine is not trivial. When reading the JSON query at runtime, the field names of a NetFlow $v5$ record are read in as strings. Utility functions are defined that map the field names to internal `struct` offsets and the field types and the operations to internal `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 as shown in Listing 1. Each individual branch of the flowquery itself is described in a `branch` struct.

```
1   struct flowquery {
2     size_t                      num_branches;
3     size_t                      num_merger_clauses;
4
5     struct branch**             branchset;
6     struct merger_clause**      merger_clauseset;
7     struct merger_result*       merger_result;
8     struct ungrouper_result*    ungrouper_result;
9   };
10
11  struct branch {
12    int                         branch_id;
13    struct ftio*                ftio_out;
14    struct ft_data*             data;
15
16    size_t                      num_filter_clauses;
17    size_t                      num_grouper_clauses;
18    size_t                      num_aggr_clause_terms;
19    size_t                      num_groupfilter_clauses;
20
21    struct filter_clause**      filter_clauseset;
22    struct grouper_clause**     grouper_clauseset;
23    struct aggr_term**          aggr_clause_termset;
24    struct groupfilter_clause** groupfilter_clauseset;
25
26    struct filter_result*       filter_result;
27    struct grouper_result*      grouper_result;
28    struct groupfilter_result*  gfilter_result;
29  };
```

Listing 1.   Flow Query and Branch Structs

The JSON query can also 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 query is parsed. These flags are used throughout the engine to only enable the requested functionality.

B. Execution Workflow

A custom C library has been written to directly read/write data stored in `flow-tools` format. The library sequentially reads the 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` as shown in Listing 2. The array of such records are indexed allowing fast retrieval in $O(1)$ time.

```
1   struct ft_data {
2     int                            fd;
3     struct ftio                    io;
4     struct fts3rec_offsets         offsets;
5     struct ftver                   version;
6     u_int64_t                      xfield;
7     int                            rec_size;
8     char**                         recordset;
9     size_t                         num_records;
10  };
```

Listing 2.   Trace Data Struct

In order to be able to make comparisons on the field offsets of a term, the comparator needs to know the type of the comparison and the length of the field offset. This information is parsed by execution engine once the query is read and is therefore not available at compile time. In order to subvert the need to define complex branching statements, a dedicated comparator is defined for every possible field length and comparison operation. A Python script generates C source code for these comparators at compile time conforming to the structure shown in Listing 3. This allows the term definitions to make runtime calls using a function name derived from the combination of operation type and field length.

```
1   struct filter_term {
2     size_t                         field_offset;
3     uint64_t                       value;
4     uint64_t                       delta;
5     struct filter_op*              op;
6     bool (*func)(
7                 const char* const  record,
8                 size_t             field_offset,
9                 uint64_t           value,
10                uint64_t           delta
11               );
12  };
```

Listing 3.   Filter Term Struct

1) Splitter: `nfql` uses identifiers to reference a flow record in the `char` array. This eliminated the need to copy all the flow-records when moving ahead in the pipeline. As a result, there is no dedicated splitter stage in the execution engine. Each branch references the flow records from a common memory location. This helps keep the memory costs at a minimum when multiple branches are involved.

2) Filter: The execution engine, as defined by the flow query language must 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 produce 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
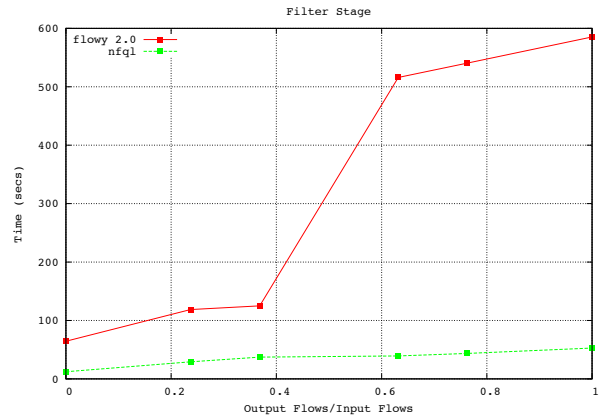

Fig. 2.   Filter Stage: Flowy 2.0 vs `nfql`

flow-records is high. We optimized this behavior in `nfql` by merging 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. We used the publicly available Trace 7 from the SimpleWeb [18] to compare the performance of `nfql` against the one defined by the flow query language as shown in Fig. 2. The filter stage implementation with these optimizations runs 10 times faster and is more pronounced on higher ratios.
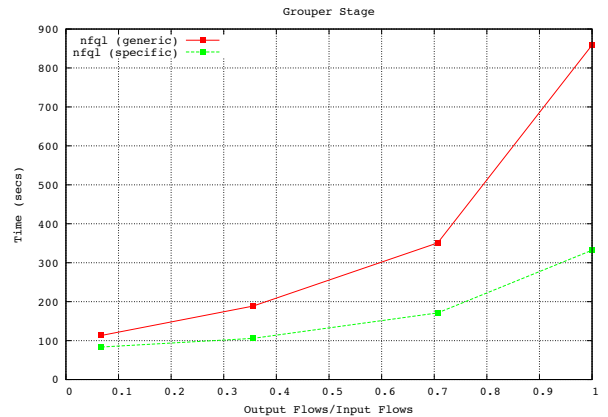

Fig. 3.   Grouper Stage: `nfql` (Generic) vs `nfql` (Specific)

3) Grouper: In order to be able to make relative 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 however, will only work on equality comparisons. A better approach, as implemented in `nfql` is to sort the filtered recordset on the field offsets of

all the requested grouping terms of a clause. This helps the execution engine perform a nested binary search to reduce the linear pass to a fairly small filtered recordset. 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 $O(p * n * lg(n))$ in the average case, where n is the number of filtered records, p is the number of grouping terms in a clause 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 the same preprocessing step taking $O(p * n * lg(n))$. The performance evaluation of the grouper handling this special case against when handling generic cases is shown in Fig. 3.

The resultant group records are a conglomeration of multiple flow records with some common characteristics. Some of the non-common characteristics can also be aggregated into a single value using group aggregations as defined in the query. Such an aggregated group record is again mapped to a NetFlow $v5$ record template. This allows the aggregated group records to be written to a file as a representative of all its members.

4) Group Filter: The groupfilter stage is used to filter the groups 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.

5) Merger: The merger is used to relate groups from different branches according to a merging criterion. 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 runtime. 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 implemented. The result of the iterator is then be used to make a match.

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 $O(g^m)$ where g 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. `nfql` 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 grouper 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 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.

6) 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. 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 called from their respective stages just before the comparison is needed.

Each stage of the processing pipeline is dependent on the result of the previous one. As a result, the stages should only proceed, 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 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 flow-records echoed to the standard output can also be written as `flow-tools` files. In fact, results from each stage of the pipeline can be written to separate files with the increase in the verbosity level. This leads to additional loops over the resultsets if the writes are made at the end of the processing pipeline. The execution engine therefore writes each result record to a file as soon as it is seen by the pipeline stage.

D. Adaptable Compression Levels

The engine uses the `zlib` [19] software library to compress the results written to the `flow-tools` files. `zlib` supports 9 compression levels with 9 being the highest. `nfql` allows the user to supply its desired choice of the compression level. A default level of 5 is used for writes if a choice is not indicated. Fig. 4 shows the time taken to write a sample of records passing the filter stage for each `z-level`. It goes to show that each level adds its own performance overhead and must be used with discretion. It is also important to note that other tools may use different compression algorithms. `nfdump` for
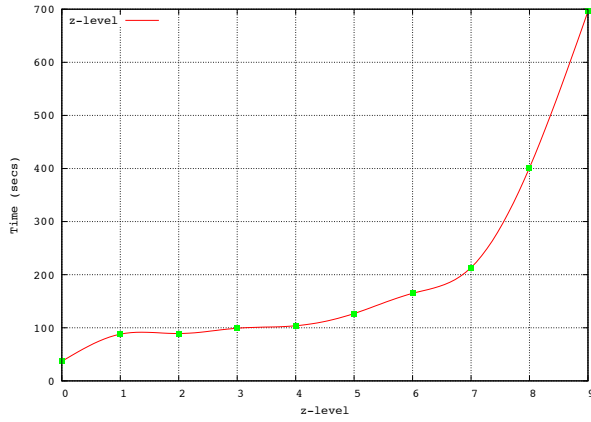
Fig. 4. `z-level` Effect on Performance

instance, uses `lzo` [20] compression to trade space for faster compression and decompression.

V. PERFORMANCE EVALUATION

We used the first $20M$ records from the publically available Trace 7 from the SimpleWeb [18] repository for our performance evaluations. The input trace was compressed at `ZLIB_LEVEL` 5 using the `zlib` suite. It was also converted to `nfdump` and SiLK compatible formats keeping the same compression level. The suite was run on a machine with 24 cores of 2.5 GHz clock speed and 18 GiB of memory.

The first set of queries stress the filter stage. We use varying values on the `packet` field offset to control the amount of flow records that are passed by the filter. The resultant filtered records are written as `flow-tools` files and compressed at `ZLIB_LEVEL` 5. 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 results are shown in Fig. 5.
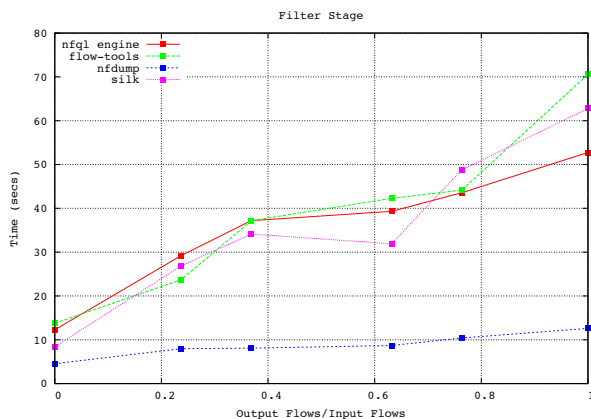

Fig. 5. Filter Stage: `nfql` vs SiLK, `flow-tools`, `nfdump`

It can be seen that the performance of the filter stage in `nfql` is comparable to that of `flow-tools` and SiLK. SiLK takes less time on lower ratios, but then SiLK and `nfdump` also use their own file format. `nfdump` appears to be significantly faster than the rest. This is because `nfdump` uses

the `lzo` compression scheme. It goes to show that adding `lzo` compression will likely improve `nfql`'s filter performance. Note that all the tools were single-threaded in this evaluation, and did not completely utilize the 24 available cores.

The second set of queries 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 `flow-tools` 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 `flow-tools` do not support grouping, and therefore are not considered in this evaluation. The results are shown in Fig. 6.
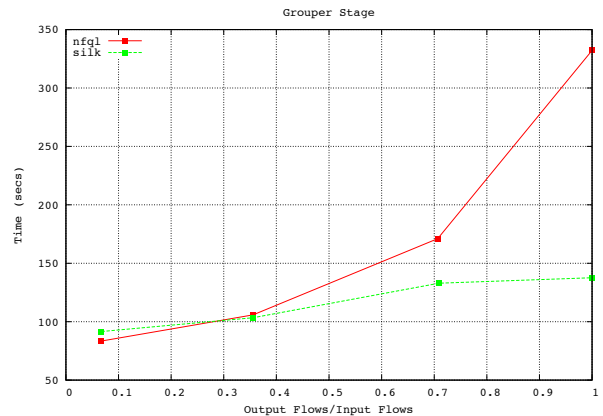

Fig. 6. Grouper Stage: `nfql` vs SiLK

The evaluation graph reveals that 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 its own file format is responsible for the drift. SiLK's `rwgroup` tool is also supplied a `--summarize` flag to force it to write only the first record of each group, to make both tools write the same number of records. 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 advantage of an inherent concurrency arising from a pipe between `rwsort` and `rwgroup`, which makes the two processes run concurrently, the effect of which gets more pronounced on higher ratios. The profiling results from GNU `gprof` [21] 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 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` field offset to control the amount of groups that are filtered ahead. The filtered groups are again written as `flow-tools` files using the same `zlib` compression level. The ratio of the number of output filtered groups to the number of the input groups is plotted against time. The results are shown in Fig. 7.
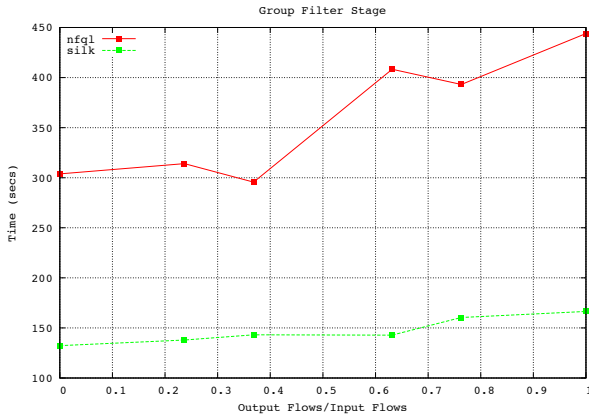

Fig. 7.   Group Filter Stage: `nfql` vs SiLK

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 the 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 `nfql` group filter has a 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 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 `flow-tools` files using the same `zlib` compression level. The ratio of the number of merged groups to twice[1] the number of flow records in the original trace is plotted against time. The results are shown in Fig. 8. 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.

It can be seen that the merger is the most performance critical 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 `flow-tools` 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
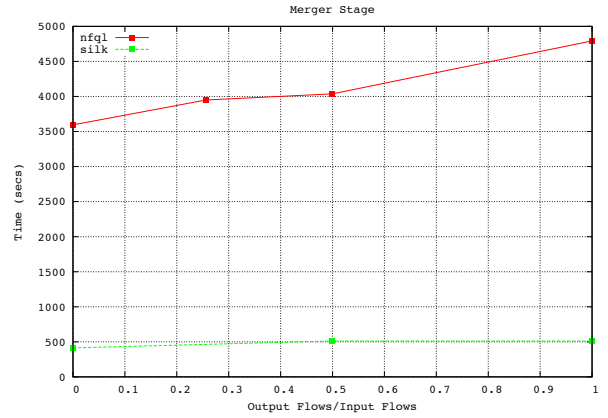

Fig. 8.   Merger Stage: `nfql` vs SiLK

time is taken in writing these results to the file. These results although look less promising, they are way better than the original NFQL merger implementation. The optimized merger implementation takes advantage of sorted nature of filtered groups and therefore can significantly reduce the number of merger matches. It also writes a merged group record to a file only once despite the number of times it has matched. Without these optimizations, running such queries on the merger kept the CPU churning for days without results.

The last set of queries stress the ungrouper stage. They reuse the entire merger queries as is, but enable the ungrouper 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 `flow-tools` 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 rewriting the merged groups as `flow-tools` files using the same `zlib` compression level. This means that the execution engine ends up taking twice the amount of time than the merger. It is important to note, that SiLK does not have such an equivalent ungrouping tool and is therefore not considered in this final evaluation.

VI. CONCLUSION

We presented `nfql`[2], an efficient C implementation of the stream-based flow query language, NFQL. The language allows applying absolute or relative filters, aggregating flows into groups, evaluating timing relationships among them, and merging them into one collection. `nfql` can execute such complex queries 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 openly available 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 different file storage format. The evaluation queries developed as a part of this research work may also develop into a more general benchmark suite for flow query tools and platforms.

---

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

[2]`http://nfql.vaibhavbajpai.com`

REFERENCES

[1] B. Claise, "Cisco Systems NetFlow Services Export Version 9," RFC 3954 (Informational), Internet Engineering Task Force, Oct. 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3954.txt

[2] ——, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information," RFC 5101 (Proposed Standard), Internet Engineering Task Force, Jan. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5101.txt

[3] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller, "An Overview of IP Flow-Based Intrusion Detection," *Communications Surveys Tutorials, IEEE*, vol. 12, no. 3, pp. 343 –356, quarter 2010.

[4] A. Callado, C. Kamienski, G. Szabo, B. Gero, J. Kelner, S. Fernandes, and D. Sadok, "A survey on internet traffic identification," *Communications Surveys Tutorials, IEEE*, vol. 11, no. 3, pp. 37 –52, quarter 2009.

[5] 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: Springer-Verlag, 2009, pp. 15–28.

[6] 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

[7] V. Perelman, N. Melnikov, and J. Schönwälder, "Flow signatures of Popular Applications," in *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, May 2011, pp. 9 –16.

[8] K. Kanev, N. Melnikov, and J. Schönwälder, "Implementation of a stream-based IP flow record query language," in *Proceedings of the Mechanisms for autonomous management of networks and services, and 4th international conference on Autonomous infrastructure, management and security*, ser. AIMS'10. Springer-Verlag, 2010, pp. 147–158.

[9] J. Schauer, "Flowy 2.0: Fast Execution of Stream based IP Flow Queries," Bachelor's Thesis, Jacobs University Bremen, Campus Ring 1, 28759 Bremen, Germany, May 2011.

[10] L. Deri and S. Suin, "Effective traffic measurement using ntop," *Communications Magazine, IEEE*, vol. 38, no. 5, pp. 138 –143, may 2000.

[11] D. Plonka, "Flowscan: A network traffic flow reporting and visualization tool," in *Proceedings of the 14th USENIX conference on System administration*, ser. LISA '00. Berkeley, CA, USA: USENIX Association, 2000, pp. 305–318. [Online]. Available: http://dl.acm.org/citation.cfm?id=1045502.1045522

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

[13] A. Oslebo, "Stager a web based application for presenting network statistics," in *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, april 2006, pp. 1 –15.

[14] 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

[15] CERT/NetSA at Carnegie Mellon University, "SiLK (System for Internet-Level Knowledge)," [Online]. Available: http://tools.netsa.cert.org/silk [Accessed: Sep 4, 2012].

[16] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)," RFC 4627 (Informational), Internet Engineering Task Force, Jul. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4627.txt

[17] Metaparadigm Pte Ltd., "json-c - JSON implementation in C," [Online]. Available: http://oss.metaparadigm.com/json-c [Accessed: Sep 4, 2012].

[18] R. R. R. Barbosa, R. Sadre, A. Pras, and R. van de Meent, "Simpleweb/University of Twente Traffic Traces Data Repository," http://eprints.eemcs.utwente.nl/17829/, University of Twente, Technical Report TR-CTIT-10-19, April 2010.

[19] 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

[20] M Oberhumer, "LZO real-time data compression library," [Online]. Available: http://www.oberhumer.com/opensource/lzo [Accessed: Sep 4, 2012].

[21] 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