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

II. Related Work

`flow-tools` [2] 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` [3] 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.

`tcpdump` and `wireshark` are the most popular tools used for packet capture and analysis. `tcpdump` [4] is a premier command-line utility that uses the `libpcap` [5] 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` [6] 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.

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 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

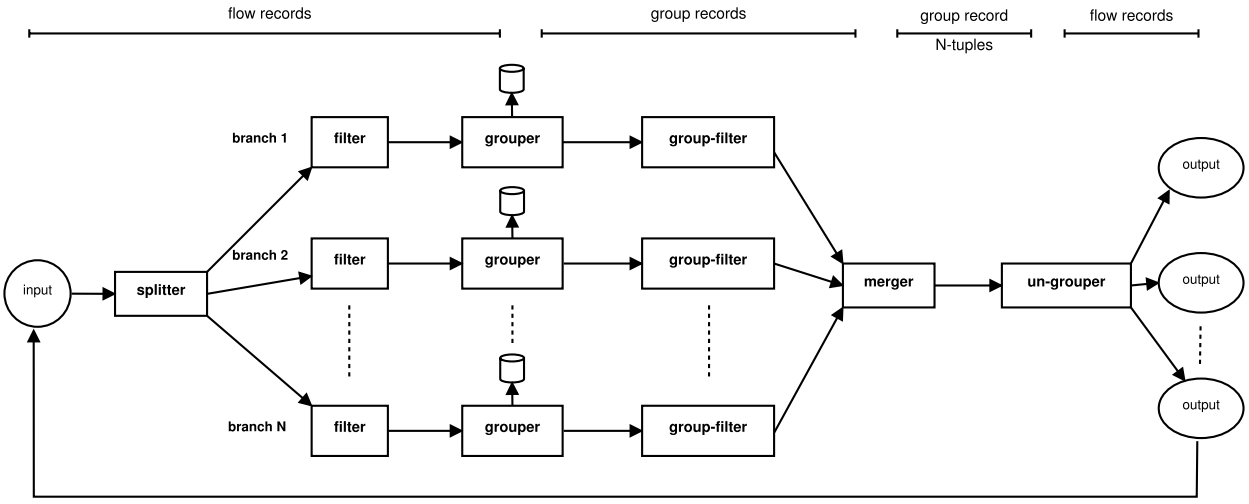

Fig. 1. NFQL Pipeline [1]

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`[1] 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 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.

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

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.

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

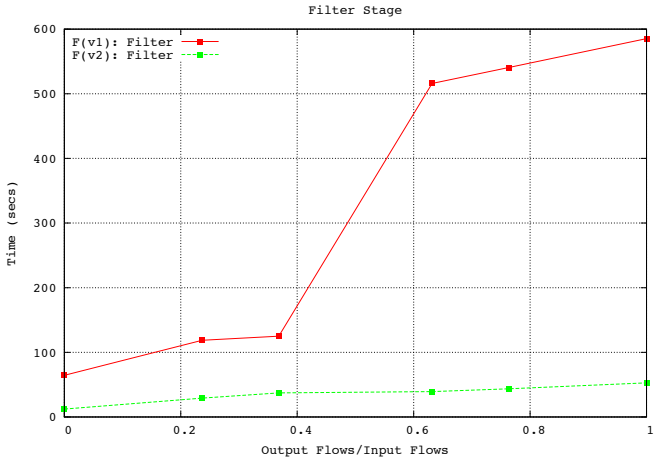

Fig. 2. Filter Stage: F($v1$) vs F($v2$)

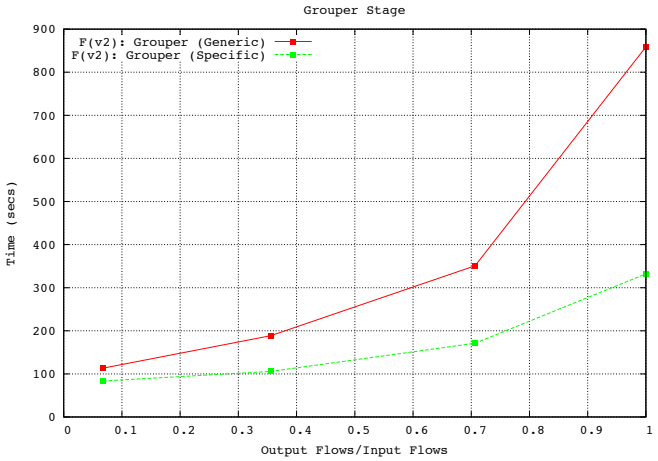

Fig. 3. Grouper Stage: F($v2$) (Generic) vs F($v2$) (Specific)

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 [2] 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 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

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

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

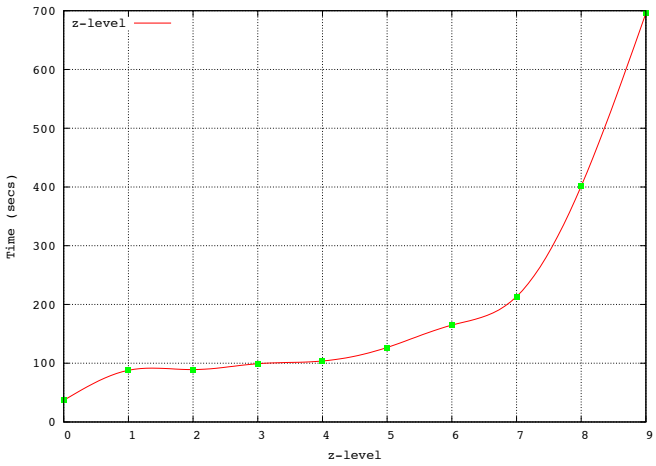

Fig. 4. F($v2$): `z-level` Effect on Performance

V. Performance Evaluation

performance evaluation goes here ...

VI. Conclusion

The NFQL conclusion goes here ...

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] 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

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

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

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

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

[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