F

A complete system integration of stream-based IP flow-record querier

VAIBHAV BAJPAI

Masters Thesis

School of Engineering and Science
Jacobs University Bremen
Bremen, Germany

June 2012

ABSTRACT

Short summary of the contents in English...

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

IPFIX Internet Protocol Flow Information Export

HDF Hierarchical Data Format

LALR Look-Ahead LR Parser

PLY Python Lex-Yacc

HDFS Hadoop Distributed File System

API Application Programming Interface

Part I

INTRODUCTION

You can put some informational part preamble text here

1 TRAFFIC MEASUREMENT APPROACHES

1.1 capturing packets

1.2 capturing flows

1.3 remote monitoring

1.4 remote metering

2 FLOW EXPORT PROTOCOLS

2.1 NETFLOW

2.2 IPFIX

2.3 SFLOW

3 LANGUAGES AND TOOLS

3.1 SQL-BASED QUERY LANGUAGES

3.1.1 *NetFlow exports as relational DBMS*

3.1.2 *Data Stream Management System*

3.1.3 *Gigascope*

3.1.4 *Tribeca*

3.2 FILTERING LANGUAGES

3.2.1 *flow-tools*

3.2.2 *nfdump*

3.3 PROCEDURAL LANGUAGES

3.3.1 *FlowScan*

3.3.2 *Clustering NetFlow Exports*

3.3.3 *SiLK Analysis Suite*

4 LEGAL CONSIDERATION

Part II

STATE OF THE ART

You can put some informational part preamble text here

5 FLOWY

Flowy [4][5] is the first prototype implementation of a stream-based flow record query language [6][1][7]. The query language allows to describe patterns in flow-records in a declarative and orthogonal fashion, making it easy to read and flexible enough to describe complex relationships among a given set of flows.

5.1 PYTHON FRAMEWORK

Flowy is written in Python. The framework is subdivided into two main modules: the validator module and the execution module. The validator module is used for syntax checking and interconnecting of all the stages of the processing pipeline and the execution module is used to perform actions at each stage of the runtime operation.

5.1.1 *PyTables and PLY*

Flowy uses PyTables [8] to store the flow-records. PyTables is built on top of the Hierarchical Data Format (HDF) library and can exploit the hierarchical nature of the flow-records to efficiently handle large amounts of flow data. The `pytables` module provides methods to read/write to PyTables files. The `FlowRecordsTable` class instance within the module exposes an iterator interface over the records stored in the HDF file. The `GroupsExpander` class instance within the same module on the other hand exposes an iterator interface over the group records and facilitates ungrouping to flow records.

In addition, Flowy uses Python Lex-Yacc (PLY) for generating a Look-Ahead LR Parser (LALR) parser and providing extensive input validation, error reporting and validation on the execution modules.

5.1.2 *Records*

Flow-records are the principal unit of data exchange throughout Flowy's processing pipeline. The prototype implementation allows the `Record` class (defined in the `record` module) to be dynamically generated using `get_record_class(...)` allowing future implementations to easily plug in support for Internet Protocol Flow Information Export (IPFIX) or even newer versions of NetFlow [9] exports. The `FlowToolsReader` class instance (defined in `ftreader` module) provides an iterator over the records defined in `flow-tools` format.

This can be plugged into the `RecordReader` class instance (defined in `record` module) to instantly get `Record` class instances.

5.1.3 *Parsers and Statements*

The `parser` module holds definitions for the lexer and parser. The statements when parsed are implicitly converted into instances of classes defined in the `statement` module. The instances contain meta-information about the parsed statement such as the values, line numbers and sub-statements (if any).

5.2 PROCESSING PIPELINE

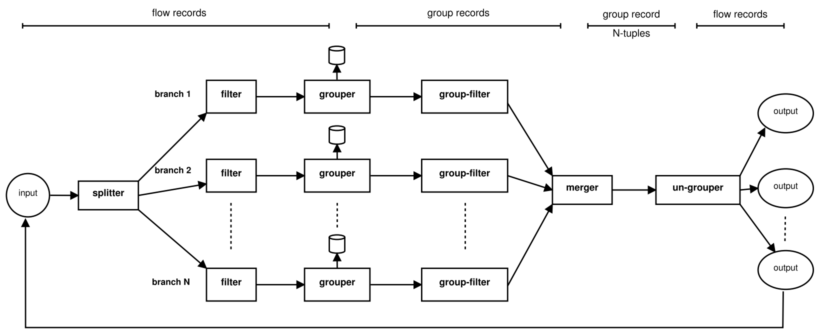

Figure 1: Flowy: Processing Pipeline [1]

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. Figure 1 shows an overview of the processing pipeline. The flow record attributes used in this pipeline exactly correlate with the attributes defines in the IPFIX Information Model specified in RFC 5102 [10]. A complete description on the semantics of each element in the pipeline can be found in [6]

5.2.1 *Splitter*

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.

5.2.1.1 *Splitter Implementation*

The `splitter` module handles the duplication of the `Record` instances to separate branches. Instead of duplicating each flow-record to every branch (as specified in the specification), the implementation follows a pragmatic approach by filtering the records beforehand against all the defined filter rules to determine which branches a flow-record might end up in and saves this information in a record-mask tuple of boolean flags. The `go(...)` method in the `Splitter` class then iterates over all the (record, record-mask) pairs to dispatch the records to corresponding branches marked by their masks using the `split(...)` method. The class uses branch names to branch objects mapping to achieve the dispatch.

5.2.1.2 *Splitter Validator*

The `splitter_validator` module handles the splitter processing stage. The `SplitterValidator` class within the module uses the `Parser` and `FilterValidator` instances passed to it to create a `Splitter` instance and its child `Branch` instances.

5.2.2 *Filter*

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

5.2.2.1 *Filter Implementation*

The `filter` module handles the filtering stage of the pipeline. Since in the implementation the filtering stage occurs before the splitting stage, a single `Filter` class instance suffices for all the branches. Within the `filter` module, each filtering statement is converted into a `Rule` class instance, against which the flow-records are matched. The `Rule` instances are constructed using the (branch mask, logical operator, arguments) tuple. After matching the records against the rules, the record's branch mask is set and is then used by the splitter to dispatch the records to the filtered branches.

5.2.2.2 *Filter Validator*

The `filter_validator` module handles the filter processing stage. The `FilterValidator` class within the module uses the `Parser` instance passed to it to create a `Filter` instance once the check on semantical constraints have passed. The constraints involve checking whether records fields referenced in the filter definition exist, whether filters references in composite filter definitions exist and whether duplicate filter definitions are defined.

5.2.3 *Grouper*

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. In addition 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.

5.2.3.1 *Grouper Implementation*

The `grouper` module handles the grouping of flow-records data. The `Group` class instance contains group-record's field information required for absolute filtering. It also contains the first and last records of the group required for relative filtering of the group-records. The `AggrOp` class instance handles the aggregation of group-records. The allowed aggregation operations are defined in `aggr_operators` module. Custom-defined aggregation operations are also supported using `-aggr-import` command line argument.

5.2.3.2 *Grouper Validator*

The `grouper_validator` module handles the grouper processing stage. The `GrouperValidator` class within the module uses the `Parser` and `SplitterValidator` instances passed to it to create a `Grouper` instance once the check on semantical constraints such as the presence of referenced names and non-duplicate names have passed. Three aggregation operations: `union(rec_id)`, `min(stime)`, `max(etime)` are added by default to each `Grouper` instance.

5.2.4 *Group-Filter*

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

5.2.4.1 *Group-Filter Implementation*

The `groupfilter` module handles the filtering of group-records. The `GroupFilter` class within the module iterates over the flow-records within the group and applies filtering rules across them. The filtering rules reuse the `Rule` class from the `filter` module. The flow-records are then added to the time index and stored in a pytables file for further processing. For groups that do *not* have a group-filter defined for them, run through a `AcceptGroupFilter` class instance.

The `timeindex` module handles the mapping of the time intervals to the flow-records. The time index is used by the merger stage to learn about the records that satisfy the Allen relations. The `add(...)` method in the `TimeIndex` class is used to add new records to the time index. The `get_interval_records(...)` method on the other hand is used to retrieve records within a particular time interval.

5.2.4.2 *Group-Filter Validator*

The `groupfilter_validator` module handles the group-filter processing stage. The `GroupFilterValidator` class within the module uses the `Parser` and `Grouper` instances passed to it to create a `GroupFilter` instance. The check for the referenced fields is performed against the aggregate clause defined in grouper statements. The class instance uses the `AcceptGroupFilter` instance in case a branch does *not* have a group filter defined for it.

5.2.5 *Merger*

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 [11]

5.2.5.1 *Merger Implementation*

The `merger` module handles the merging of stream of groups passed as input. It is implemented as a nested branch loop organized in an alphabetical order where every branch is a separate `for-loop` over its records. During iteration, each branch loop executes the rules that matches the arguments defined in the group record tuple and subsequently passes them to the lower level for further processing. The `Merger` class represents the highest level branch loop and as such it must iterate over all of its records since it does not have any rules to impose restrictions on the possible records. The `MergerBranch` on the other hand represents an ordinary branch loop with rules.

5.2.5.2 *Merger Validator*

The `merger_validator` module handles the merger processing stage. The `MergerValidator` class within the module uses the `Parser` and `GroupFilterValidator` instances passed to it to create a `Merger` instance once the check on referenced fields and branch names has passed. In addition, the validator also ensures semantic checks on Allen algebra such as whether the Allen relation arguments are correctly ordered, whether the Allen rules with the same set of arguments are connected by an OR and whether each branch loop is reachable by an Allen relation (or a chain of Allen relations) from the top level branch.

5.2.6 *Ungrouper*

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.

5.2.6.1 *Ungrouper Implementation*

The `ungrouper` module handles the unwrapping of the group-records. The generation of flow-records can also be suppressed using the `-no-records-ungroup` command line option. The `Ungrouper` class instance is initialized using a merger file and an explicit export order.

5.2.6.2 *Ungrouper Validator*

The `ungrouper_validator` module handles the ungrouper processing stage. The `UngrouperValidator` class within the module uses the `Parser` and `MergerValidator` instances passed to it to create a `Ungrouper` instance. This processing stage does *not* require any validation.

5.3 FUTURE OUTLOOK

5.3.1 *Reduced Copying*

The `reset(...)` method of the `BranchMask` class performs a deepcopy on objects which significantly lowers performance. The invocation of this method can be inhibited by either removing the branch mask mechanism for simpler queries or removing it entirely. In addition avoiding usage of immutable containers (tuples) can also reduce internal copying during mutation.

5.3.2 *Using PyTables in-kernel searches*

PyTables can accelerate flow-records selection using a `where` iterator. The `where` clause is passed to the PyTables kernel which is written in C, therefore the selection can occur at C speed and only the filtered flow-records reach the Python space. This would require PyTables in-kernel search query support in the filtering rules and the `pytables` module would have to be extended to read from PyTables filtered flow-records.

5.3.3 *Multithreaded Merger*

The merger stage in the processing pipeline is currently the most computation intensive operation and is unfortunately single-threaded. As suggested in [4] it should be possible to handle the outermost branch loop using multiple threads in a non-blocking fashion to improve performance.

6 FLOWY IMPROVEMENTS USING MAP/REDUCE

Flowy, although clearly setting itself apart with its additional functionality to query intricate patterns in the flows demonstrates relatively high execution times when compared to contemporary flow-processing tools. A recent study [2] revealed that a sample query run on small record set (around 250MB) took 19 minutes on Flowy as compared to 45 seconds on `flow-tools`. It, therefore is imperative that the application will benefit from distributed and parallel processing. To this end, recent efforts were made to investigate possibility of making Flowy Map/Reduce aware [2]

6.1 MAP/REDUCE FRAMEWORKS

Map/Reduce is a programming model for processing large data sets by automatically parallelizing the computation across large-scale clusters of machines [12]. It defines an abstraction scheme where the users specify the computation in terms of a `map` and `reduce` function and the underlying systems hides away the intricate details of parallelization, fault tolerance, data distribution and load balancing behind an Application Programming Interface (API).

6.1.1 *Apache Hadoop*

Apache Hadoop is a Map/Reduce Framework written in Java that exposes a simple programming API to distribute large scale processing across clusters of computers [13]. However in order to make Flowy play well with the framework, the implementation either has to use a Python wrapper around the Java API or translate the complete implementation to Java through Jython. Even more since Flowy uses HDF files for it's I/O processing, staging the HDF files properly in the Hadoop Distributed File System (HDFS) [14] and then later streaming them using Hadoop Streaming utility would still be an issue as suggested in [2]

6.1.2 *The Disco Project*

Disco is a distributed computing platform using the Map/Reduce framework for large-scale data intensive applications [15]. The core of the platform is written in Erlang and the standard library to interface with the core is written in Python. Since the `map` and `reduce` jobs can be easily written as Python functions and dispatched to the worker threads in a pre-packaged format, it is less difficult to setup Disco to utilize Flowy as a `map` function. In addition, the usage of HDF files for I/O processing pose no additional modifications whatsoever since the input data files can be anywhere and supplied to the worker threads in absolute paths.

6.2 PARALLELIZING FLOWY

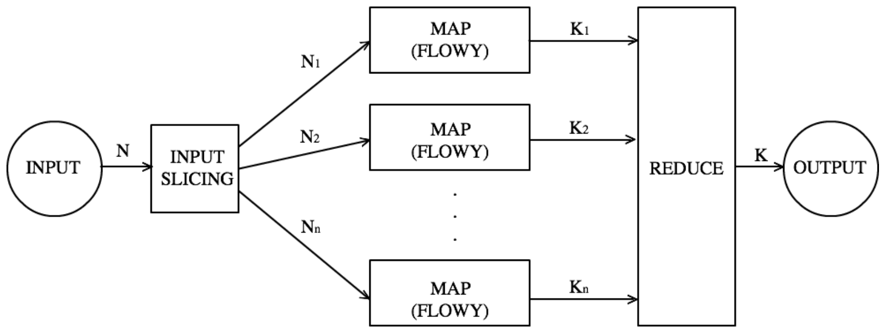

Figure 2: Parallelizing Flowy using Map/Reduce [2]

In an attempt to parallelize Flowy, it was run as a map function on a successful single node Disco installation as shown in 2. Although the setup on a multiple node cluster would be theoretically almost equivalent, Flowy has not yet been tested in such a scenario.

6.2.1 *Slicing Inputs*

When running several instances of Flowy, it is imperative to effectively slice the input flow-records data in such a way so as to minimize the redundancy in distribution of input. To achieve this, the semantics of the flow-query needs to be examined from the simplest to the most complex cases. However, it is also important to realize that as of now it is not possible to *leave* out any stage in the Flowy's processing pipeline and the following examination was based on such an assumption.

6.2.1.1 *Using* only *Filters*

A flow query that involves only the filtering stage of the processing pipeline can slice its input flow data by either adding explicit export timestamps to allow each branch to skip records or separate out the input flow data into multiple input files for each branch.

6.2.1.2 *Using Groupers*

A flow query that also involves groupers and group-filters cannot use static slice boundaries since the grouping rules can be either absolute or relative. As a result, Flowy needs to be made aware of slice boundaries by passing the timestamps as command line parameters. In such a scenario, each branch will skip the pre-slices, whereby the actual slices and the post-slices will be processed to create relevant groups as shown in figure 3. It is advisable to slice the flow-records at low traffic spots to avoid the risk of cutting the records belonging to the same group. The idea of skipping pre-slices and sweeping across

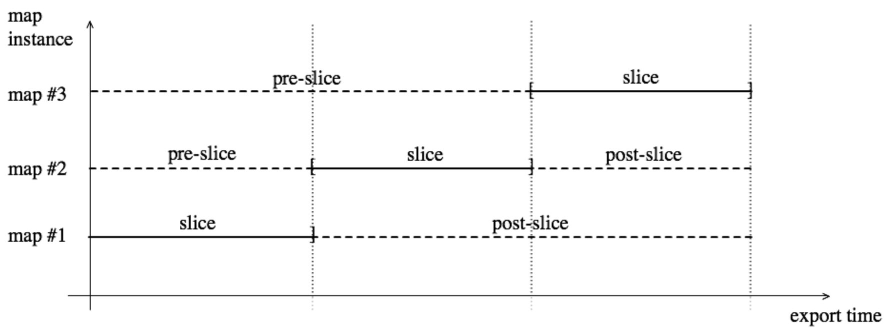

Figure 3: Slice Boundaries Aware Flowy [2]

post-slices can result in many fragmented redundant groups. These can be identified by the `reduce` function by removing the groups that are a proper subset of the previous group in the slice at the cost of additional complexity as shown in figure 4

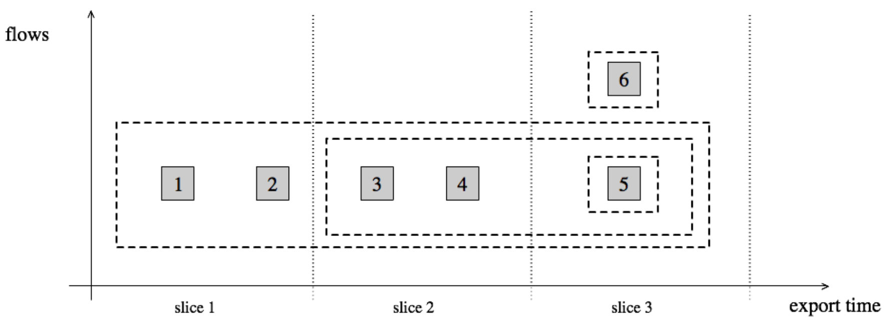

Figure 4: Flowy: Redundant Groups [2]

6.2.1.3 *Using Mergers*

The relative dependency in the merger stage of the pipeline is even worse, since the comparison needs to take place between groups resulting from the output of separate `map` functions. This calls for inhibiting parallelism up to and including the group-filter stage. As a result each worker thread would return back its filtered groups to the master node, which then would apply the rules of the merger stage to all the received groups at once in a `reduce` function. In such a scenario, although the branch with the longest runtime complexity will become the bottleneck for the merger, the overall runtime would still be dramatically reduced when the number of branches are large as suggested in [16]

6.2.2 *Flowy as a Map Function*

A Disco job function is created that contains the `map/reduce` function definitions and a location of an input file of flow-records data. A `sliceIt(...)` function within a newly defined `sliceFileCreator` module is used to create the input file. The function takes a HDF file and number of worker threads as input and writes out the slices in the input file by equally dividing HDF timespan by the number of worker threads.

In this way, the input file gets slice times for each worker thread in a separate line, which the Disco job function eventually reads to spawn a new `map` function with the slice times passed as arguments. The `map` function then starts an instance of Flowy and passes the slice times and the HDF file as command line parameters for processing.

This required modification to the `flowy_exec` module to add support for extra parameters. The filter stage of the pipeline was modified to allow for skipping of the pre-slices in the flow-records data. The grouper stage was modified as well to restrict creation of new groups that do *not* fall within the passed slice boundaries. However, the modification of the `reduce` function to work with the files pushed out by each Flowy instance of the `map` function to merge groups from each branch and eliminate duplicate records is left open.

7 FLOWY 2.0

In an attempt to make the first prototype implementation of Flowy comparable with the contemporary flow-record analysis tools, the substitution of the performance hit sections of the Python code was thought out. Flowy 2.0 [3] is the outcome of a complete rewrite of the core of the prototype implementation in C making it relatively faster in orders of magnitude.

7.1 PERFORMANCE ISSUES AND IMPROVEMENTS

number of records	overall in s	filter in s (%)	grouper in s (%)	merger in s (%)
103k	1177 s	28 s (2%)	240 s (20%)	909 s (77%)
337k	20875 s	110 s (1%)	2777 s (13%)	17988 s (86%)
656k	70035 s	202 s (0%)	8499 s (12%)	61334 s (87%)
868k	131578 s	274 s (0%)	15913 s (12%)	115391 s (87%)
1161k	234714 s	1212 s (1%)	25480 s (11%)	208022 s (88%)

Figure 5: Runtime Breakup of Individual Stages [3]

The runtime breakup of individual stages of the processing pipeline as shown in 5 reveal that the grouper and merger incur a massive performance hit. A quick investigation hints towards usage of large deep nested loops in the merger with a worst-case $O(n^3)$ runtime complexity.

In addition, pushing the flow-records data from one stage of the pipeline to another involved deep copying of the whole flow data whereby a mere passing across of a reference across a pipeline in a branch would have sufficed. Similar behavior is visible when the grouper when passing group records saved the individual flow-records in a temporary location tagged with the groups and/or subgroups they belonged to.

The decision decision to use PyTables to read and write flow-records in HDF format also added to the complexity. Since, the input flow-records were most of the time in either `flow-tools` or `nfdump` file-formats, each time they had to be converted into HDF file formats prior to Flowy's execution which was unnecessary.

7.1.1 *Unmodified Parts*

The flow-querier parser written in PLY and the validators written for each stage of the processing pipeline that check for semantics correctness were left unmodified, since their execution time was invariant of the size of the input data and slightly varying on the query complexity in itself.

7.1.2 *Early Improvements*

Thread affinity masks were set for each new thread created to delegate the thread to a separate processor core. `try/except` blocks were narrowed down to only code that needed to be exception handled. A test-suite was developed with few sample queries and input traces to validate Flowy's results for regression analysis. A `setup.py` script was written to facilitate installation of Flowy and its dependencies and `options.py` was replaced with `flowy.conf` configuration file with the standard human-readable key-value pairs. The command line option handling was switched from `optparse` to `argparse` module and a switch was added for easy profiling. The profiling output was modified as well to allow standard tab delimiters which can be easily parsed by other tools. The flow query was also extended to allow file contents to be supplied using `stdin`. Variable names that are now part of Python identifiers were renamed.

A C library was written to parse and read/write flow-records in `flow-tools` compatible format. The C library was connected to the Python prototype using Cython [17][18]. 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. Cython was used since it allowed to write C extensions in a Pythonic way by strong-typing variables, calling native C libraries and allowing usage of pointers and structs, thereby providing the best of both worlds [19].

7.1.3 *Data Format*

A custom C library was written to directly read/write data in the `flow-tools` format to provide a drop-in replacement for PyTables and overcome the overhead of format conversions. 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 a separate `struct`. The array of such records are indexed allowing fast retrieval in $O(1)$ time. The C library is currently limited to support *only* `flow-tools` formats; `nfdump` file formats are yet to be supported.

7.1.4 *Rewrite of Core Algorithms in C*

A design decision was made to rewrite the entire processing pipeline in C. However, currently the core cannot parse the flow-query file, therefore the execution is triggered by a tedious manual filling of the `structs` by the contents of the query.

```
struct filter_rule {
  size_t field_offset;
  uint64_t value;
  uint64_t delta;
  bool (*func)(
    char *record,
    size_t field_offset,
    uint64_t value,
    uint64_t delta);
};
```

Listing 1: Filter Rule Struct [3]

A filter stage `struct` is shown in listing 1. The field to be filtered is indicated using a `field_offset` and `field_length` in the `char` array of a records. The value to be compared against with is also supplied which can be either a static value or another field of a record. `func` is a function pointer to the operation that is to be carried out on a record whose record identifier is passed to it. The filter runs in $O(n)$ time as it needs to traverse through all the records of the `char` array.

```
struct merger_rule {
  size_t branch1;
  size_t field1;
  size_t branch2;
  size_t field2;
  uint64_t delta;
  bool (*func)(struct group *group1,
    size_t field1,
    struct group *group2,
    size_t field2,
    uint64_t delta);
};
```

Listing 2: Merger Rule Struct [3]

Similarly, a merger stage `struct` is shown in listing 2. `branch{1,2}` are branch identifiers and `field{1,2}` are the aggregated field identifers in the order of aggregation. `func` is a function pointer pointing to the operation to be carried out. The merger runs in $O(n^k)$ time where k is the number of branches. The `char` arrays in each branch are disjoint since a record cannot be part of more than one group.

The current core implementation also strictly adheres to the processing pipeline shown in figure 1. As such, it is not currently possible to skip stages. In addition it is not currently possible to have more than one merger or grouper in the flow-query or aggregate fields in the grouper module since `char` array storage is not possible.

To overcome the limitations, the future outlook at this stage is to allow the Python prototype to parse and validate the flow query file which in turn would pass the contents to a Cython wrapper which on the fly will forward them to the core to properly fill in the `structs`.

7.2 BENCHMARKS

number of records	runtime old Python Flowy in s	runtime new C Flowy in s
103k	1177	0.3
337k	20875	3.4
656k	70035	13
868k	131578	23
1161k	234714 (2.7 days)	86

Figure 6: Flowy vs Flowy2 [3]

A flow query with the union aggregations stripped off was used as a sample to compare the runtime performance of Flowy [4] with Flowy2 [3] . The benchmarks are shown in figure 6. It is conspicuous how well the replacement of the core algorithms from Python to C turned out to be.

```
$ time sh -c "flow-cat traces | flow-filter -P80"
$ time sh -c "flow-cat traces | ./flowy"
```

Listing 3: Flowy2 vs `flow-tools` [3]

In another test, Flowy2's functionality was reduced to absolute filtering to compare its performance with a state-of-the-art `flow-tools` analysis tool using 3. It turned out Flowy2 performed just as comparable if not better on an average.

7.3 FUTURE OUTLOOK

7.3.1 *System Integration*

7.3.2 *Searching with Trees*

7.3.3 *Specialized Functions in Inner Loops*

7.3.4 *Efficient Multithreading*

7.3.5 *Additional Functionality*

8 FLOWY: APPLICATIONS

8.1 IPv6 transition failure identification

8.2 Cybermetrics: user identification

8.3 Application identification using flow signatures

8.4 TCP level spam detection

Part III

MOTIVATION

Part IV

WORK PLAN

You can put some informational part preamble text here

9 DESIGN

10 IMPLEMENTATION

11 PERFORMANCE EVALUATION

12 CONCLUSION

Part V

IMPLEMENTATION AND EVALUATION

You can put some informational part preamble text here

13 DESIGN

14 IMPLEMENTATION

15 PERFORMANCE EVALUATION

16 FUTURE WORK

17 CONCLUSION

Part VI

APPENDIX

A APPENDIX

Put your appendix here.

BIBLIOGRAPHY

[1] Vladislav Marinov and Jürgen 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*, DSOM '09, pages 15–28, Berlin, Heidelberg, 2009. Springer-Verlag.

[2] Peter Nemeth. Flowy Improvements using Map/Reduce. Bachelor's thesis, Jacobs University Bremen, Campus Ring 1, 28759 Bremen, Germany, May 2010.

[3] Johannes 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.

[4] Kaloyan Kanev. Flowy - Network Flow Analysis Application. Master's thesis, Jacobs University Bremen, Campus Ring 1, 28759 Bremen, Germany, August 2009.

[5] Kaloyan Kanev, Nikolay Melnikov, and Jürgen 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*, AIMS'10, pages 147–158, Berlin, Heidelberg, 2010. Springer-Verlag.

[6] Vladislav Marinov. Design of an IP Flow Record Query Language. Master's thesis, Jacobs University Bremen, Campus Ring 1, 28759 Bremen, Germany, August 2009.

[7] Vladislav Marinov and Jürgen Schönwälder. Design of an IP Flow Record Query Language. In *Proceedings of the 2nd international conference on Autonomous Infrastructure, Management and Security: Resilient Networks and Services*, AIMS '08, pages 205–210, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Francesc Alted and Mercedes FernÃ¡ndez-Alonso. PyTables: Processing And Analyzing Extremely Large Amounts Of Data In Python. 2003.

[9] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), October 2004.

[10] J. Quittek, S. Bryant, B. Claise, P. Aitken, and J. Meyer. Information Model for IP Flow Information Export. RFC 5102 (Proposed Standard), January 2008. Updated by RFC 6313.

[11] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, November 1983.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[13] T. White. *Hadoop: The Definitive Guide*. Definitive Guide Series. O'Reilly, 2010.

[14] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1 –10, May 2010.

[15] Prashanth Mundkur, Ville Tuulos, and Jared Flatow. Disco: A Computing Platform for Large-Scale Data Analytics. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, Erlang '11, pages 84–89, New York, NY, USA, 2011. ACM.

[16] Johannes Schauer, Nikolay Melnikov, and Jürgen Schönwälder. F. 2012.

[17] Dag Sverre Seljebotn. Fast numerical computations with Cython. In Gaël Varoquaux, Stéfan van der Walt, and Jarrod Millman, editors, *Proceedings of the 8th Python in Science Conference*, pages 15 – 22, Pasadena, CA USA, 2009.

[18] Ilmar Wilbers, Hans Petter Langtangen, and Åsmund Ødegård. Using Cython to Speed up Numerical Python Programs. In B. Skallerud and H. I. Andersson, editors, *Proceedings of MekIT'09*, pages 495–512. NTNU, Tapir, 2009.

[19] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The Best of Both Worlds. *Computing in Science Engineering*, 13(2):31 –39, march-april 2011.

DECLARATION

Put your declaration here.

Bremen, Germany, June 2012

Vaibhav Bajpai