Flowy - Network Flow Analysis Application

Kaloyan Kanev

20th August 2009

Abstract

NetFlow/IPFIX records are a very valuable tool for network traffic analysis. Flow analysis has several applications like, gathering network statistics and usage patterns for billing and network planning, or intrusion and attack detection. This project's goal is to develop a network analysis tool called flowy, which is based on an orthogonal query language, as proposed in Vladislav Marinov's M.Sc. thesis[18].

Contents

1 Introduction and Motivation for Research **4**

2 Background and State of the Art **6**
2.1 NetFlow/IPFIX 6
2.2 flow-tools 6
2.3 nfdump 7
2.4 SiLK 8

3 Query Language **9**
3.1 Query language architecture 9
3.2 Query language elements 10
3.2.1 Splitter 10
3.2.2 Filter 11
3.2.3 Grouper 11
3.2.4 Group-filter 12
3.2.5 Merger 12
3.2.6 Ungrouper 13
3.2.7 Linking elements 13
3.3 Language semantics and computational complexity 14
3.4 Implementation-specific changes to the query language 16
3.4.1 Record field references and constants 16
3.4.2 External functions and extension of query rules 16
3.4.3 Merger branches statements and Allen relations delta 17

4 Storage Considerations **19**
4.1 Proprietary binary storage 20
4.2 IPFIX persistent storage format 20
4.3 RDBMS storage 20
4.4 Scientific computation data formats 21
4.5 Final storage considerations 22
4.6 Performance comparison between HDF and MySQL 22

5 Flowy Implementation **25**
5.1 Architecture Overview and Technology Used 25
5.1.1 Python 25
5.1.2 PyTables 25
5.1.3 PLY (Python Lex and Yacc) 27
5.2 Records 27

5.3 Storage readers and writers . . . 28
5.3.1 Ftreader . . . 28
5.3.2 Pytables . . . 29
5.4 Filters . . . 30
5.4.1 Rules . . . 30
5.4.2 Branches and branch masks . . . 30
5.4.3 Subbranches . . . 31
5.4.4 Composite filters and pseudobranches . . . 31
5.4.5 Operators and operator evaluation . . . 32
5.4.6 Reasons for the masking system . . . 33
5.5 Splitter . . . 33
5.6 Grouper . . . 34
5.6.1 Changes to the grouping algorithm . . . 34
5.6.2 Groups export and shortcut rules . . . 35
5.7 Group-Filter . . . 36
5.8 Time Index . . . 37
5.9 Merger . . . 38
5.9.1 Merger branch loops . . . 38
5.9.2 Merger branch loops index optimisation . . . 40
5.9.3 Merger module classes . . . 41
5.10 Ungrouper . . . 41
5.11 Parser and statements . . . 41
5.12 Filter validator . . . 42
5.13 Splitter validator . . . 42
5.14 Grouper validator . . . 43
5.15 Groupfilter validator . . . 43
5.16 Merger validator . . . 43
5.17 Ungrouper validator . . . 45

6 Testing and conclusions **46**
6.1 Speed testing . . . 46
6.1.1 Execution time and input size . . . 46
6.1.2 Speed compared to flow-tools . . . 46
6.2 Profiling flowy execution . . . 47
6.3 Possible improvement . . . 48
6.3.1 Lowering the amount of copying . . . 48
6.3.2 Using PyTables in-kernel searches . . . 48
6.3.3 Multithreaded Merger . . . 49
6.4 Conclusion . . . 49

Chapter 1

Introduction and Motivation for Research

Nowadays computer networks transfer large amounts of data. Bandwidths of Gbit/s are not uncommon. Network traffic analysis if very important for network management and planning. The large size of the data traversing modern networks makes it impractical to store all the data data in the network packets for long-term analysis. It is even impractical to store only each packet's header, as this also requires very significant storage[1]. A widely used way to gather network traffic information is NetFlow. NetFlow is a protocol for exporting traffic statistics in the form of flow records. Flows are unidirectional streams of data identified by network and transport layer endpoints and transport protocol, type of service field, and input interface [10]. Usually data from many packets is aggregated into one record which reduces the storage needs. Flow records contain other useful information such as the number of packets and bytes within the flow, transport protocol specific flags seen in the flow, and others. Despite some loss of data granularity, flow records provide, enough information for traffic analysis tasks such as: Application Monitoring and Profiling, User Monitoring and Profiling, Network Planning, Security Analysis, and Accounting/Billing [10].

There are many existing free open-source network flow analysis tools. The most popular ones suffer from non-uniform syntax. Parts of the queries are written as command line options, and others are written into files with specific syntax. Command line options syntax becomes unwieldy for complex flow analysis queries and is hard to read.

This project aims to create flowy - a network analysis tool, which uses an uniform and orthogonal query language - [18], which is easy to read and understand. One of flowy's main goals is to improve performance compared to existing tools. One way to improve query performance, is choosing better storage methods than existing solutions, and by avoiding performance degrading practises such as using UNIX pipes to construct complex filtering pipelines, and use shared memory instead.

Another area where flowy will try to improve over existing network flow analysis tools is to provide a way to easily group flows on ranged criteria such as range of ports, or on small changes (e.g., port scan where every consecutive port number is

[1] Simplistic calculation for a link carrying 100Mbit/s IPv4 TCP traffic with 1500byte MTU shows that it transfers a maximum of 8333 packets/s. Saving only source and destination addresses and port numbers, and protocol number would generate 8333*(2*32+2*16+8)/8≈100KB/s or more than 8GB per day!

incremented with 1). Such tasks common in security analysis are not well supported in existing software, therefore a new tool is needed to address this problem.

The rest of this document provides a conceptual framework on which Flowy is based, as well as the details of the implementation. This paper is organised as follows. Chapter 2 provides background and overview of the state of the art. Chapter 3 introduces the query language used by this project. Chapter 4 discusses performance issues that where considered in order to choose the storage method for Flowy. Chapter 5 explains the more important implementation details.

Chapter 2

Background and State of the Art

2.1 NetFlow/IPFIX

The typical set key properties that distinguish network flows are: source IP address, destination IP address, corresponding source and destination port numbers, protocol number, TOS byte, and incoming virtual interface [10]. A flow record consists of information about a specific flow that was observed at an observation point and contains the key properties that distinguish the flow, as well as measured properties of the flow (like number of bytes in a flow, number of packets, start/end time and others - see Table 2.1) [23]. Flow records don't always correspond to flows. There are situations where several quickly occurring consecutive flows between the same endpoints may be exported as single record due to packet loss or sampling, or due to connectionless protocol. It is also possible that several records may correspond to a single flow. If a flow has been inactive for predetermined period its record is exported and consecutive packets with the same key properties will constitute a new flow. Records for long lasting flows get exported at regular intervals even though the flows may have not ended. Flows with connection oriented transports (e.g, TCP flows), are usually separated based on connection start and end. That is a TCP flow record start time is recorded when a SYN TCP flag is seen and it's end time is recorded when a RST or a FIN TCP flag is seen within the flow [10].

NetFlow is a protocol developed by Cisco Systems, which is used to export network flow records. It defines a packet format for transferring flow records from observation point to the collection point [10]. The current version is NetFlow v.9. However the most widely used version today is still NetFlow v.5. Prior to v.9, NetFlow had a fixed packet format, which did not allow addition of new properties. NetFlow v.9 provides an extensible data format based on templates. Each flow record references a template, which defines what fields are present in the record and what data types this fields have [10]. Figure 2.1 shows an example of a NetFlow v.9 export packet. IPFIX is an IETF proposed standard based on NetFlow v.9 which aims to standardise flow record export and transfer to a collection point [14].

2.2 flow-tools

Flow-tools is a popular suite of tools for collecting and processing flow records. It is composed of 24 separate programs (commands), which work together via UNIX pipes.

Bytes	Contents	Description
0-3	srcaddr	Source IP address
4-7	dstaddr	Destination IP address
8-11	nexthop	IP address of next hop router
12-13	input	SNMP index of input interface
14-15	output	SNMP index of output interface
16-19	dPkts	Packets in the flow
20-23	dOctets	Total number of Layer 3 bytes in the packets of the flow
24-27	first	SysUptime at start of flow
28-31	last	SysUptime at the time the last packet of the flow was received
32-33	srcport	TCP/UDP source port number or equivalent
34-35	dstport	TCP/UDP destination port number or equivalent
36	pad1	Unused (zero) bytes
37	tcp_flags	Cumulative OR of TCP flags
38	prot	IP protocol type (for example, TCP = 6; UDP = 17)
39	tos	IP type of service (ToS)
40-41	src_as	AS number of the source, either origin or peer
42-43	dst_as	AS number of the destination, either origin or peer
44	src_mask	Source address prefix mask bits
45	dst_mask	Destination address prefix mask bits
46-47	pad2	Unused (zero) bytes

Table 2.1: NetFlow v.5 record format[10]

Flow-capture is the capturing daemon, which collects flow exports, compresses them and stores them. *Flow-cat* reads and concatenates separate flow record files. Its output is usually input for the flow processing tools. *Flow-filter* or *flow-nfilter* is used to filter flow records [2]. *Flow-filter* uses command line options to define the filter while *flow-nfilter* accepts filter definition defined in files [2]. *Flow-tag* tags flows based on IP address or AS number and is used to group flows by customer network. The tags can later be used with *flow-fanout* or *flow-report* to generate customer based traffic reports. *flow-report* generates reports on a flow records data set. Some of the report types have the ability to group and/or aggregate on some of the fields or a group of fields [2]. *Flow-export* is used to export data from the *flow-tools* storage format to other formats such as cflowd format, MySQL, PostgreSQL, and ASCII CSV [2]. *Flow-print* prints flow records in human readable format [2]. Using *flow-tools* for analysis usually involves concatenating a range of flow record files and piping them through *flow-filter* and *flow-report* or *flow print*. The greatest disadvantage of *flow-tools* is the overhead created by piping the data between separate programs instead of using shared memory, and the fact that there is no uniform syntax to control all the flow analysis with a single file.

2.3 nfdump

Nfdump is similar in functionality to flow-tools, but it uses a smaller number of separate programs (only 3). *Ncapd* is the daemon that collects and stores flow records [6]. *Nfdump* is used to filter flow records and generate TopN reports on the flow records. It can also print the resulting flow records in human readable format, or store them back into binary [6]. Nfdump's filter format is similar to tcpdump filter syntax. *Nfdump*'s has

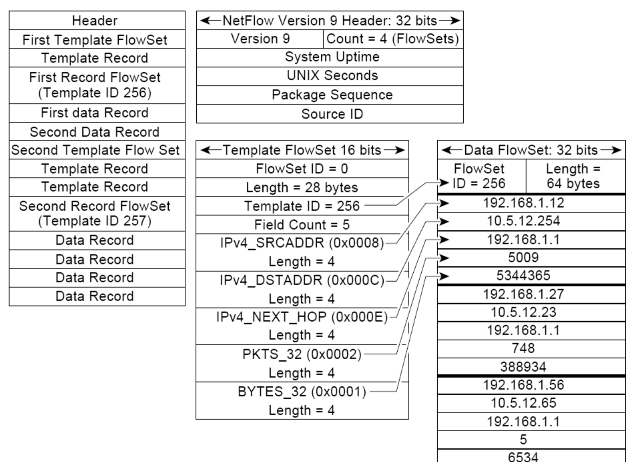

Figure 2.1: NetFlow v.9 Export Packet Example [10]

more limited types of reports, compared to flow-tools' *flow-report*. For example there is no simple way to do a report where flows are aggregated based on input interface, source and destination addresses. Most of the options for flow record processing are given at the command line, only filters may be specified in a separate file.

2.4 SiLK

System for Internet-Level Knowledge(SiLK) is yet another suite of tools for collection and analysis of flow records. Like flow-tools it is composed of many separate programs. *Rwfilter* is used to read and filter flow records [9]. It has two output paths PASS_PATH and FAIL_PATH. The first outputs all records that passed all filter files, and the second passes records that failed any of the rules [9]. Filter rules are defined on the command line. The only way to define more than one match value or range of values is the so called tuples file. Tuples file contains a coma separated table of match values. One can include a header which specifies which columns of the possible flow record fields are included in the tuples file [9]. Filtering with a tuples file selects only flow records, whose fields match a row in the tuples file. A limitation of *rwfilter* is that due to it's syntax expressed in command line options, it is not possible to define compound filters. For example if one needs to define a filter that matches all records that don't belong to certain source address and have certain port, one would have to use two *rwfilter* instances, one that matches the source address, and pipe it's FAIL_PATH to another filter which matches the port number [9]. Reports on flow records are generated using *rwstats* and *rwcount* which as their names suggest group and aggregate data. SiLK includes some commands that provide better convenience compared to *flow-tools*, such as *rwmatch*, which can match records from two separate streams and copy successful matches to a single file. Another convenience tool is *rwscan* which detects scanning activity patterns in the input flow set.

Chapter 3

Query Language

The most distinguishing trait of this project's product will be its query language, which is designed by Vladislav Marinov and is the subject of his M.Sc thesis. It is a declarative language, whose main goal is to express filtering and aggregation analysis stages used to discover patterns in network flow records. There are several advantages that this query language has, compared to the existing network analysis tools discussed in Chapter 2. The first advantage is that it provides a way to define all network analysis stages into a single file, as opposed to separate files and command line options. The language was designed to be orthogonal and easy to read and understand [18]. Another very important advantage over existing tools is that it provides a way to describe time relationships of groups of flows using Allen's time interval algebra [11]. The rest of this chapter discusses in more detail the query language properties, and syntax.

3.1 Query language architecture

The query language follows a stream-oriented approach. It defines processing stages which take input, do some analysis(filtering, aggregation, etc.) and produce output. The processing stages are conceptually connected via pipes, where each non-terminal processing stage's output is the input for another stage. A processing stage consists of one of the following:

- filter
- grouper
- group-filter
- splitter
- merger
- ungrouper

The query language architecture is visualised in Figure 3.1.

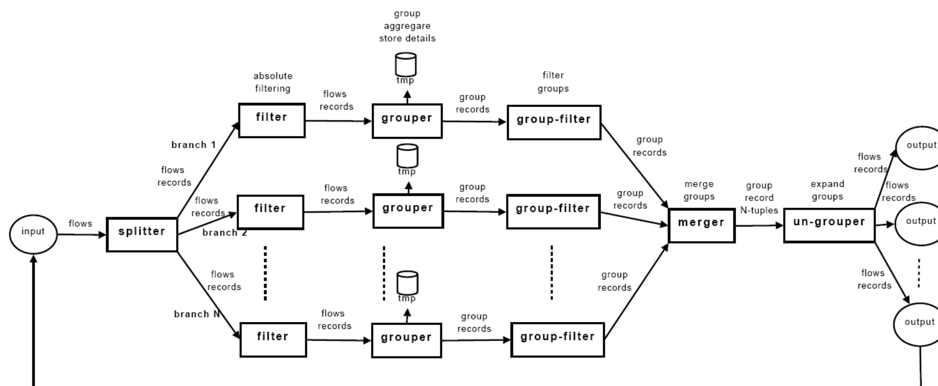

Figure 3.1: Network Flow Query Language Architecture

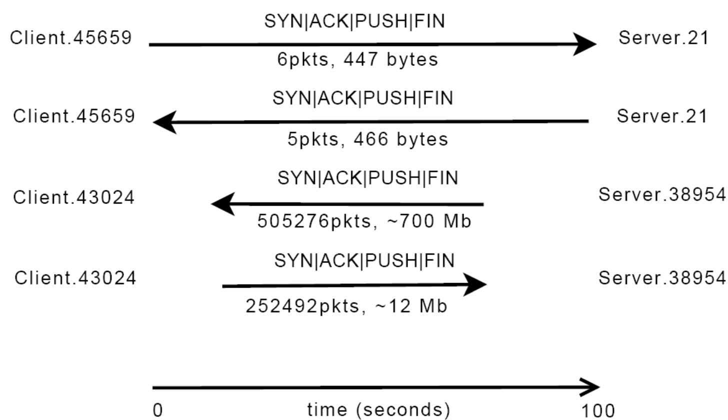

Figure 3.2: Flow level breakdown of a simple FTP transfer

3.2 Query language elements

This section demonstrates the basic elements of the query language. The example statements construct a query used to detect an FTP download. The FTP session consists of control and data connections, that run over TCP. The FTP server listens on port 21 for client control connections, and initiates data connection to the client using a random port above 1024. A flow breakdown of an FTP connection is shown on Figure 3.2. There are two pairs of flows where each pair's elements are close in time. The pair of flows that has destination port 21, should carry less traffic than the other pair and should precede it in time.

3.2.1 Splitter

The splitter is the simplest processing stage, it just takes a stream of flows as input and clones it on several output branches. It does not change the flow records.

Example:

```
splitter s {}
```

3.2.2 Filter

The filter operator takes a stream of flow records and outputs only the records, which match the filtering rules. Filters do absolute filtering, that is they compare flow attributes of a single flow record to absolute values (or ranges of absolute values) defined in the filter rules. Filters don't compare one flow's attribute with another flow's attributes, or in other words they don't do relative filtering.

Example:

```
filter f_control {
  srcport = 21 OR dstport = 21
  proto = TCP
}
filter f_data {
  proto = TCP
}
```

The first filter matches TCP traffic from or to port 21. The second filter matches any TCP traffic.

3.2.3 Grouper

The grouper operator takes as input a stream of flow records and partitions them into groups. Grouping rules may describe either relative or absolute filtering rules, so they may match inter-dependencies between flows. Groupers may also do aggregation on some of the flow attributes. Groupers consist of modules, which match subgroups. Each flow record belongs to only one group but may be a member of more than one subgroup. In order to be added to a group a flow must match at least one subgroup. In other words a group of flow records is the union of subgroups of records. The output of the grouper is a set of group records. Each group consists of records related through the matching rules in the modules and contains aggregated information for the grouped records.

Example:

```
grouper g_group_tcp {
  module g1 {
    srcip = dstip
    dstip = srcip
    srcport = dstport
    dstport = srcport
    stime = stime relative-delta 500ms
  }
  module g2 {
    srcip = srcip
    dstip = dstip
    srcport = srcport
    dstport = dstport
    stime = stime relative-delta 500ms
  }
  aggregate g1.srcip as srcip, g1.dstip as dstip,
            g1.srcport as srcport, sum(bytes) as bytes,
```

```
                min(stime) as stime, max(etime) as etime
}
```

In this example the group filter contains two modules. The first module (g1) matches groups of records which represent bidirectional communication over two endpoints (IP:port pairs). It does so by grouping records whose destination IP is the same as the source IP of the first record in the subgroup (*srcip* = *dstip*) and whose source IP is the same as the destination IP of the first record of the subgroup (*dstip* = *srcip*). Similar matching is done for the transport layer ports. The *stime* = *stime relative-delta 500ms* rule matches only records, whose start time is at most 500ms after the previous record added to the subgroup. The second module (g2) groups flows with the same source and destination IPs and ports. It also groups only flows whose start times are at most 500ms apart. Conceptually g1 adds the opposite direction flow of bidirectional communication, while g2 groups flows records that may represent single flow but were separated during export. The aggregate clause adds the source and destination IP and ports to the group record, and the sum of transferred bytes from all flows participating in the group, and the minimum start time and maximum end time of any of these flows. The resulting groups, which are the union of the corresponding subgroup records (records matched by either g1 or g2 or both) should contain bidirectional flows whose start times at most 500ms apart. The group record contains the communication endpoints of this bidirectional flow, as well as the start and end time of the flow and the size of data transferred.

3.2.4 Group-filter

Group-filters are similar to filters but operate on groups rather than flows. The filtering rules are absolute, i.e. they match group's attributes to the filtering rules, but do not match one group's attributes with the attributes of another group. Group-filter rules may filter on aggregated attributes.

Example:

```
group-filter gf_data {
  bytes > 700MB
}
```

This group-filter removes groups with less than 700MB traffic by doing comparison on the aggregated field bytes.

3.2.5 Merger

The merger operator takes N streams of groups, and produces N-tuples of groups, which match filtering rules. It forms all possible combinations of N-tuples, which contain one element from each group and applies filtering rules on these tuples. Merger filtering rules express relative dependencies between groups, and may use Allen time intervals algebra to define timing and concurrency constraints. Like grouper, merger may be subdivided into modules. However unlike a grouper's output, the merger output does not consist of the union of the merged groups, but consists of the set difference of the first module and the union of all the other modules (m1_output\(m2_output $\bigcap$ m3_output $\bigcap$ m4_output $\bigcap$...)). Merger modules are conceptually connected in series, so that when a group record tuple passes through the first merger module its groups which come from branches common to other merger modules are evaluated by the next

merger module and passed on only if they don't match any other module's rules. In other words the first module matches some groups, while consecutive modules remove some groups which match other rules. This way only group tuples that match m1 rules but don't match other modules' rules are selected.

Example:

```
merger M {
  module m1 {
    branches A,B
    A.srcip = B.dstip
    A.dstip = B.srcip
    A.bytes << B.bytes
    B d A
  }
  export m1
}
```

This merger example, takes groups from two branches A and B. Branch A which is not shown as an example here matches the control connection of the FTP transfer, while branch B, shown which consists of the filter, grouper and group filter in shown in the previous subsections, matches the FTP data transfer connection. The merger rules *A.srcip = B.dstip* and *A.dstip = B.srcip* state that the data and control connections should have reversed endpoints as is the case with active FTP transfer, where data connection is initiated by the server. The rule *A.bytes* << *B.bytes* states that the number bytes transferred in the branch A group should be much less[1] than the bytes transferred by the group from branch B. The last rule *B d A* is Allen time algebra rule [11]. It states that the group from branch B should occur during group from A (the time interval of B should be completely contained within the time interval of A). The result of the grouper is pairs of control and data connection corresponding to FTP sessions.

3.2.6 Ungrouper

The ungrouper takes as input tuples of group records and expands them to their original flow records, which it outputs.

Example:

```
ungrouper U {}
```

3.2.7 Linking elements

Linking elements define how different processing elements are connected between each other.

Example:

```
input -> S
S branch A -> f_control -> g_group_tcp -> M
```

[1] The language definition does not specify what << (much less) and >> (much more) exactly mean. On suggested value is 10 times less or more, but it is up to the implementation to decide. In flowy this meaning may be made adjustable through program options, or some additions to the language can be made to change the meaning in different places within the query. Another possibility is to just add simple arithmetic to the comparison rules and omit << and >> altogether for rules like A.bytes * 10 < B.bytes.

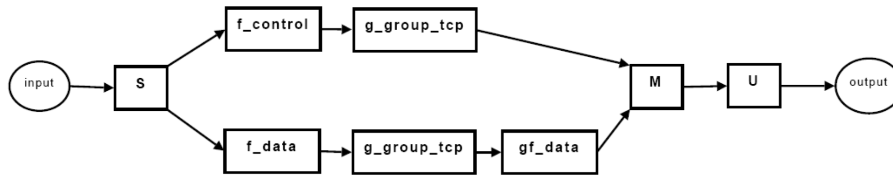

Figure 3.3: Capturing FTP download of Big Files with the IP Flow Filtering Framework

```
S branch B -> f_data -> g_group_tcp -> gf_data -> M
M -> U -> output
```

This example creates the processing pipeline shown on Figure 3.3.

3.3 Language semantics and computational complexity

This section discusses in more detail the conceptual semantics and the computational complexity of the different operations in the query language, as defined in the language specification [18]. The described algorithms, were chosen with simplicity of understanding in mind, not performance. The actual implementation discussed in Chapter 5 improves on the complexity of some of these algorithms.

The splitter, filter, and group-filter iterate over their input just once, so their complexity is linear (O(n)) with the size of input, which is either number of records for splitter and filter, or number of groups for group filter. It is important to note that the size of the input changes between different stages due to filtering.

The ungrouper, has to iterate only once over its input, but then it has to retrieve the original flow records, so it's actual computational complexity depends on the storage and caching mechanisms used. For example if all the flow records are cached on per group basis, then the ungrouping operation only has to concatenate these records. On the other hand if group records are not cached or groups are stored only as tags in original flows, then the ungrouping operation may possibly have to do a matching on all records.

The grouper operation uses a greedy approach. The algorithm starts with the first record, which is tagged with an unique group label and with all the subgroups labels. Then it iterates over all the records and tags all the records that satisfy the grouping module rule with the same subgroup labels. When the end of the flow records sequence is reached, the algorithm takes the first untagged record and gives it new unique label, and tags the remaining untagged records with the same label if they match the grouping rule. The matching and labelling is done until all records have a label. The algorithm can be described in pseudo code as follows:

```
flow_record = file->first;
do {
    if (flow_record.tag == NULL){
        gl= generate_group_label();
        tag(flow_record, gl);
```

```
        for each rule module m {
          sl = generate_subgroup_label(m);
          tag(flow_record, sl);
        }
        for (r=flow_record->next; r != EOF; r=r->next) {
          for each rule module m
           if (match(flow_record, r, rules(m)) &&
                                (r.group_label==NULL)){
               tag(r,gl);
               sl = generate_subgroup_label(m);
               tag(r,sl);
           }
        }
    }
    flow_record = flow_record->next;
} while (flow_record != EOF)
create_group_records();
```

The worst case scenario occurs when each record ends up in its own group. The number of repetitions in this case is $\Sigma_{i=1}^{n} i = \frac{n(n+1)}{2}$ or $O(n^2)$ complexity.

The merger operation takes N branches and creates all possible combinations of N groups, where there is one group from each branch, and matches them to its filtering rules. The following pseudo code describes the algorithm to do this for one-module merger:

```
get_module_output(module m) {
(branch_1, branch_2, ..., branch_n) = get_input_branches(m);
for each g_1 in group_records(branch_1)
  for each g_2 in group_records(branch_2)
    ...
      ...
        for each g_n in group_records(branch_n)
           if match(g_1, g_2, ..., g_n, rules)
             output.add(g_1, g_2,..., g_n)
return output;
}
```

If there is more than one module in the merger, then the algorithm becomes:

```
get_merger_output(module m_1, module m_2,..module m_k) {
  merger_output_stream = 0;
  m1_output = get_module_output(m_1);
  B = get_input_branches(m_1);
  for each tuple t in output_stream {
    output = 0;
    for each rule module M in m_2 through m_k {
      S = get_input_branches(M);
      (b_1, b_2,..b_n) = S\B; /*set difference*/
      for each g_1 in group_records(b_1)
        for each g_2 in group_records(b_2)
```

```
        ...
          ...
          for each g_n in group_records(b_n)
            if match(g_1, g_2, ..., g_n, t, rules(M))
              output++;
    }
    if (output == 0)
      add(merger_output, t);
    return merger_output;
}
```

The worst case complexity of the get_module_output procedure algorithm is $O(m^n)$, where n is the number of branches and m is the number of groups in each branch, assuming all branches have the same number of groups. The complete algorithm potentially checks every group tuple with every tuple from the other modules, so if all modules consist of the same branches and each module outputs all tuples it that come to its input the resulting complexity is $O(km^n)$, where k is the number of modules.

3.4 Implementation-specific changes to the query language

3.4.1 Record field references and constants

The original language definition assumes that flow records will contain a fixed subset of the fields available in NetFlow v.5 [18]. In order to be able to add support for NetFlow v.9 and IPFIX, flowy needs to be able to dynamically define the record's fields and their datatypes. An extensible set of fields and datatypes has the side effect that the semantics of a comparison operations are not always clear. For example a comparison such as *proto* = *TCP* is unambiguous when there is a fixed set of fields in a record, and it is clear that *proto* is a record field and *TCP* is a constant. However, for non static record definitions it is not completely clear, what each argument of the comparison represents, because *proto* and *TCP* can be either fields of the flow record or constants. To solve the ambiguity of such cases, the query language implementation uses different tokens for record fields and constants. Unquoted strings mean field references, while quoted strings mean constant strings. In order to actually test whether the *proto* field of the record equals *TCP*, one needs to call the *protocol* function which converts the *"TCP"* string constant to the same data type as the *proto* field - a protocol number. Here is an example how the test whether the record transport protocol is TCP is done in Flowy:

```
proto = protocol("TCP")
```

The next subsection explains in more detail functions and function calling.

3.4.2 External functions and extension of query rules

The goal to achieve compatibility with the extensible flow record formats, brings the requirement for extensibility of the filtering rule operations. In other words additional record fields may need operations different from the limited set of operations (mostly arithmetic comparison and bitwise logic operations). For example, one may have a

record with fields *A*, *B*, and *chksum*, and wants to check for each record whether the checksum matches fields *A* and *B*. Since the checksum has to be calculated from the *A* and *B* fields, and there is no filter rule operation in the original definition to do checksumming, the user is given the opportunity to define their own function *checksum* and call it in the filter statement. To achieve this the function calling syntax was added:

```
chksum = checksum(A,B)
```

Flowy can not provide every possible function, but it provides an easy way to add such functions. Adding the *checksum* function is as easy as writing a Python module containing a function with the name "checksum", which takes two arguments and calculates the checksum. The function is made available to the processing pipeline by importing the module containing the function through a command line argument during execution.

User defined functions, and some of the built-ins as their name suggest may call external libraries or even programs. For example one might define a function *resolve*, which resolves DNS names to IP addresses and use the resolved IP for the tests in a filter statement.

Another extension from the language specification is that functions can be nested. Only prefix functions nesting is supported, but infix functions, such as =, <, > can have prefix functions as their arguments:

```
fun_a(A,fun_c(C)) = fun_b(A,B)
```

3.4.3 Merger branches statements and Allen relations delta

As discussed in section 3.3 the merging operator has a potentially high-exponent polynomial time complexity. In order to limit the number of evaluated group tuples in the merger, Allen relations between the branches are made mandatory. The implementation uses a time index to skip over groups which can't satisfy the Allen grouping rules. The way the Allen rules and the time index limit the number of evaluations is discussed in detail in Section 5.9.

Some Allen relations, such as < and > (before and after respectively), still allow for potentially very large number of group records to consider. In most cases for large traces when the query specifies $A < B$, it rarely means that all group records from branch A from the start of the trace until the current record from branch B should be considered. In most cases "before" means "before within some range". In order to limit the number of checked records in such cases the delta statement was added to the Allen operators:

```
A < B delta 5min
C > D delta 10s
```

The first example is interpreted as records from A should occur within 5 minutes before the current record from B. The second example means that records from C should occur within 10 seconds after the record from D. The units for the time are min, s, and ms.

Allen relations such as *m, im, s, is, f, if* and = imply that the end or start time of one record should be exactly the same as the start or end time of another record. For example *A m B* means A meets B or in other words the endtime of A is the same as the start time of B. It is very likely that two records which should be conceptually considered meeting each other but in practise are separated by few milliseconds, will

fail an Allen test for matching exactly the end and start times of the two records. In order to give some range withing which testing for conditions requiring exact match, should succeed, the Allen delta statement gives some acceptable margin of mismatch.

```
A m B delta 5ms
```

This example means A meets B with margin of error 5ms. In other words instead of testing *start time B ?= end time A*, the test is done as |*start time B - end time A*| *?< 5ms.*

The < and > relations potentially allow for searching withing large amount of records, therefore the implementation requires a delta for them, and an error is raised if the delta is not provided. The *m, im, s, is, f, if* and = don't require delta but it is advisable to use it because it allows for slightly mismatched times, due to delays and other circumstances to still pass the Allen rule test. The remaining Allen relations don't need exact matching of time, neither do they allow for searching of records withing greater duration than the duration of the current record, therefore, they don't need a delta. If the delta is specified for them it is disregarded.

There is a change to the semantics of the *branches* statement. The language specification does not put any semantics on the order of the branches within the statement. The implementation iterates through merger branches in the order they are specified in the exported module. If there is more than one module, the order of iteration is: first the branches of the export module are iterated, then any branches which are not mentioned in the exported module are iterated in the order they appear.

```
merger M {
    module m1 {
        branches B, C, A
        ...
    }
    module m2 {
        branches B, C, F
        ...
    }
    module m3 {
        branches C,D,E
        ...
    }
    export m1
}
```

In this example the order in which the branches are iterated is B, C, A, F, D, E.

Chapter 4

Storage Considerations

Storage strategy is crucial for the performance of flow analysis tools. The data sets size is generally in the range of tens of Gigabytes. [28] demonstrates that the required storage for NetFlow is typically between 0.1% and 1% of the traffic size, so for a fully saturated 100Mbit/s link, the expected storage per day is between 1GBytes and 10Gbytes. Most of the processing operations are not computationally intensive, which makes the analysis pipeline performance I/O and memory bound.

One of the main considerations when choosing storage format is retrieval strategy and speed. Depending on the type of analysis the data may only need to be accessed sequentially, but for non trivial analysis random access for flow records may be a requirement. Since the data size is expected to be larger than the typical amount of main memory available to a single computer, the data storage format should also allow random access to the records. Random access may require indexing on different type of data keys, for example time, or destination address. Therefore the ability to index on arbitrary key is an important consideration.

Another important factor to consider is the overhead imposed by the storage format. Because of the large number of records in a typical data set, even small overhead per record may lead to large total storage increase. The storage size problem can be alleviated by use of compression at the cost of additional processing needed to decompress records and sometimes loss of random access ability.

Further data format consideration is extensibility. NetFlow v.9 and IPFIX are extensible protocols, so the storage format for the analysis tool might need to be extensible as well.

This project deals with data analysis, therefore when choosing storage format, data insertion and modification in the storage and corresponding performance is of less concern, than the read performance.

There are two general approaches to storing flow records. Most flow analysis tools use a binary storage format with optional compression that corresponds to the structures representing the data in memory. Other applications use a relational database. Both types of storage have their own advantages and shortcomings. Binary formats have less overhead, and consequently need less storage, while database storage provides powerful retrieval in the form of SQL queries, at the price of increased overhead.

The next subsection discusses some of the solutions used in existing flow analysis software, and some solutions used in data storage for large data set scientific experiments.

4.1 Proprietary binary storage

Custom binary storage seems to be the most widely used storage scheme for popular network analysis tools. Some examples are flow-tools, SiLK, nfdump, and flowd [6, 9, 2, 3]. Most applications usually divide the stored data into files, each of which contains 5 to 15 minutes of records [6, 2]. The format generally corresponds to the data structures used in the main memory. There is no documentation, apart from the source code, on the exact format applications use. SiLK, flow-tools, and nfdump are able to store the information of NetFlow v5. PDUs (see Figure 2.1) [6, 9, 2]. Although documentation says that additional fields may be present in the records, filters in these applications match only on the attributes defined in NetFlow v.5 [6, 2]. Usually there is optional compression for the data, and most popular algorithms are gzip [15] and LZO [21]. The aforementioned application's storage formats are not extensible therefore they store only a subset of the information available for NetFlow v.9/IPFIX records [6, 9, 2]. The data in these binary formats is organised in time-wise fashion (by export time); flows are sequentially stored, but there is no indexing. The division into five-minute duration files may serve as a crude index, but generally there is no way to extract an arbitrary record, without searching the whole file. These applications provide export capabilities or programming language bindings to their format, so external software can use the data collected by them.

4.2 IPFIX persistent storage format

Another portable binary format which may be suitable for this project is the IPFIX persistent storage format. It is used to store IPFIX PDU's in a serialised form in a file [25]. One of the main goals of this format is to be self-describing and consistent. The self-describing property is achieved by including all metadata in the file. Consistency is achieved by storing meta-data using IPFIX PDUs [25]. This is possible because the IPFIX packet format is extensible and self describing (see Section 2.1 for more details). The format does not define internal compression, so compression has to be done by external tools. There is a portable C library called libfixbuf, that implements the data format [26]. The format's main advantage is that it describes IPFIX/NetFlow-v.9 flow records without loss of information [25], unlike proprietary formats of the network analysis tools mentioned in the previous section, which store only a subset of the information which is present in NetFlow v.5. Another important advantage is the fact that the format is well documented. The IPFIX file format unfortunately does not have the ability to index the records. Currently only SiLK has support for importing files in this format through libfixbuf, but it only stores and processes a subset of the available flow fields, which is equivalent to NetFlow v.5 flow records [9].

4.3 RDBMS storage

Network flow analysis consists of operations such as filtering, aggregation, and grouping, which can be described with general purpose query languages like SQL. It seems natural that flow records should be stored in a relational database to leverage these operations to the database engine, which is generally optimised for this type of data access. Most flow capture tools, however, prefer binary storage for it's compactness and simplicity of implementation.

Some network accounting tools use Round-robin Database [8, 7, 22], to present aggregated statistics based on flow record. Round-robin databases, automatically aggregate old records with decreasing granularity in order to keep storage requirements constant. For example a RRD may keep all records for the current week, but save only aggregated records over whole days for days before current week, and for even earlier records it can aggregate over months and years. Aggregation is done automatically when records become older than some threshold. RRDs' main application is usage statistics, but since granularity decreases for older records, they are generally not useful for tasks such as security analysis.

There are some attempts at using RDBMS, for flow analysis and they demonstrate that the most crucial decision to be made when designing the database is, which fields to index [19, 24]. [19] demonstrates a query that matches on two fields: unindexed and indexed ones. For 175 million entries table it took 8 minutes to execute the query, because it needed to walk the whole table. The same query but without matching on the unindexed field took only 0.04 seconds, so evidently indexing can speed up queries dramatically. Unfortunately indexing is very costly. The storage space needed for the index of a table containing the equivalent information of NetFlow v.5 packets on every field is almost double the storage needed for the table. The indexing operation may take days for data consisting of hundreds of millions of queries [24]. Furthermore as the table grows indexing slows insertion significantly.

A database table for storing flow-records will consist of columns corresponding to the flow record's attributes. In order to limit the impact of indexing during insertion, partitioning a data set for long period of time into tables, each of which represents a smaller interval within this period, may greatly improve insertion time, especially when indexing is used [24].

4.4 Scientific computation data formats

The Common Data Format (CDF) [1] and the Hierarchical Data Format (HDF) [4], are used by the scientific computing community to store large amounts of data from experiments. The two formats have similar capabilities.

Conceptually CDF data is accessed like rows in a table, where each record is a row of values, that has a data type determined by the column type. CDF values can be scalar or multidimensional. HDF allows for more complex structures, where data can be organised into a graph, where values are placed in the nodes and can also be multidimensional or scalar. HDF also provides heterogeneous tables, by using compound data types, which are roughly equivalent to C structs where each instance represents a single row in the table. Both data formats allow subset access for array variables.

The HDF data model is similar to an UNIX file system, and it even has the equivalent of hard links and symlinks. It has support for bitmap and projection indexes, but this support is still experimental, and does not work for compound fields, which represent heterogeneous rows in a table [5]. These index types are unfortunately of very limited use for this project. Bitmap indexes are suitable for data fields with small number of distinct values, so they are not useful for arbitrary valued fields [29]. Projection indexes are suitable for multidimensional arrays (tables where all columns have the same data type), so they are of no use as well [5]. A more general indexing solution is included in the commercial version of PyTables HDF based storage module for the Python programming language [17]. It is based on partially sorted indexes (PSI), which as the name suggest are only partially sorted structures similar to B-tree indexes

[12]. This indexing implementation is claimed to be as fast as PostgreSQL's B-tree index for data retrieval and at least an order of magnitude faster when building the index [12]. Another feature of HDF is that the logical data model is separated from the storage model, so storage is extensible. There are different storage modules, that provide functionality such as single or multi-file storage and MPI I/O. HDF also offers several compression method for data sets, so depending on the data it may greatly reduce storage requirements.

These two storage formats are general enough to be able to describe tables of flow records. Both format's libraries provide internal caching mechanism, which is can be adjusted to improve performance according the the specific usage pattern [1, 4]. Of the two formats HDF provides more features and seems to be more suitable for implementation of indexing by arbitrary fields. It already has a B-tree index for row numbers [4]. CDF uses chunking of data, to improve random row access. Each chunk header contains the numbers of the rows present in it [1]. When seeking arbitrary rows, chunks which do not contain rows of interest are skipped. Furthermore chunk headers are clustered into groups, and contain pointers to the actual chunks. If the needed chunk header is not found within the current chunk header cluster, the pointer to the next chunk cluster is followed.

HDF has the ability to store multiple heterogeneous datasets in a single file, whereas CDF only stores a single table in a single file. This means that HDF can store indexing information together with the data and access is uniform - it uses only the libhdf API [4]. CDF could also implement indexing in a separate table, but it would have to resort to the file system or another mechanism to express dependence between different datasets and indexes, which means access has to be done through both the libcdf API and the file system.

4.5 Final storage considerations

Flow collection tools generally use proprietary binary storage formats, so a flow analysis tool will almost certainly have to be able to access at least one of these formats. Fortunately the most popular of them have good interoperability and can access and export to each others formats. Using RDBMS for querying the data may improve performance, but it has its own associated performance drawbacks. It seems that a combined approach may be a good idea, where data is stored in binary file format and temporarily or in cases of repetitive analysis permanently in an SQL database. Some of the network collection tools can export their data directly to popular SQL database engines, such as MySQL and PostgreSQL. As mentioned in the previous subsection, limiting the number of data entries in a table is crucial, as well as choosing the indexed fields, so a possible approach is to insert records in the DB after the initial filtering stages of the flow analysis, to limit table sizes. A middle ground between RDBMS and plain binary format is the HDF. It provides the opportunity for compact storage and possibility to build indexes, but unfortunately there isn't a comprehensive set of free indexing tools for HDF.

4.6 Performance comparison between HDF and MySQL

In order to determine differences between performance of binary flow record storage (in flow-tools format), HDF and a RDBMS (in this case MySQL), an experiment was

	flow-tools	MySQL	HDF uncompressed	HDF zlib level 9	HDF lzo level 9
storage size [MB]	2116	7710	8133	**2364**	2780
size ratio to original	1	3.64	3.84	**1.12**	1.31
insert time real	-	200m41s	**60m40s**	77m14s	60m54s
insert time user	-	147m16s	55m57s	72m51s	57m58s
insert time system	-	1m56s	0m55s	2m21s	0m32s
query time total	4m21s	2m53s	3m9s	3m19s	**1m58s**
query time user	3m19s	0m0.040s	0m28s	3m4s	1m34s
query time system	0m15s	0m0.024s	0m15s	0m7.536s	0m8s

Table 4.1: Storage size and increase ratio compared to flow-tools format, time to insert data into database/HDF file, and time to do a query

done. The experimental data set consisted of flow records collected by flow-capture, which is part of flow-tools. The data set spans 9 days, during which 134 745 876 records were exported. Stored these records take with total size of 2116MB of storage. The actual size of the records is larger than this storage size, because flow-capture uses zlib to transparently compress records during export time. This data set was chosen because of its storage size, which is bigger than the main memory on the test machine (2.8GHz Pentium D with 1GB RAM). Since the storage size is larger than the main memory, it is not possible to buffer the whole data set there, which better demonstrates a storage method's ability to deal with slow (compared to RAM) storage. As mentioned before, flow records traces tend to be large, so this scenario better matches reality of flow record analysis, where a data set does not fit main memory.

The first step of the experiment was to export data from flow-tools format to HDF file and MySQL table. The programming language of choice for the export programs was Python, because it has access modules for the three storage types (flow-tools, HDF, and MySQL), and development in Python is generally faster than development in C. In both export applications, PyFlowTools was used to read the data. HDF export was done using PyTables [17], and export to MySQL table was done using Python's MySQL access module - MySQLDB [16]. The records had the following attributes: *dOctets, dPkts, dst_as, dst_mask, dstaddr, dstport, engine_id, engine_type, first, input, last, nexthop, output, prot, src_as, src_mask, srcaddr, srcport, sysUpTime, tcp_flags, tos, unix_nsecs, unix_secs*. For more information on the meaning of the fields refer to Table 2.1. The HDF and MySQL tables consisted of columns corresponding to these attributes. The column data type sizes were chosen to match the sizes of the corresponding record attribute data type size. The HDF format supports compression so it was tested without compression enabled, and with two different compression methods enabled.

The second step was to compare sequential read performance by executing a simple query on the original records files, on thee HDF files (uncompressed, zlib, and lzo compressed), and on the MySQL table, all containing the same same records. The query used can be written in SQL as "SELECT srcaddr FROM flows WHERE dstport='12121'". It was also translated as PyTables statement, and as flow-filter command. The query was chosen so that it does not produce too much output (only 277 rows), which would impact flow-filter unfavourably, because it's output has to be piped through flow-print to check that the same rows were returned.

Insertion and query times were recorded using UNIX time command. The results of the experiment are summarised in 4.1.

The best overall performer in this test is HDF compressed using lzo at level 9. It needed just a few seconds more time than the uncompressed HDF and was more than

three times faster than MySQL when inserting the data into the table. It does well with keeping storage requirements low at only 30% increase of space needed, which is significantly better than MySQL and uncompressed HDF, which need more than 3 times the original storage. Lzo compressed HDF was also the fastest with the test query.

Comparing uncompressed HDF to HDF with lzo compression, demonstrates that storage size has significant impact on performance, so wasting a few CPU cycles improves performance by lowering the need to transfer large amounts of data from the hard drive. However comparing lzo and zlib compression shows that nevertheless, spending too many CPU cycles to uncompress the data may also be performance degrading. Lzo is a stream compression algorithm designed with low computational requirements in mind [21]. Although zlib compresses the data better, the gains from smaller time needed for I/O, are offset the computationally intensive decompression algorithm, which wastes time. Another argument towards the theory that at least for this query storage size has the greatest impact is to look at the user and system CPU times of uncompressed HDF and MySQL and compare them to the real time passed. It is obvious that there was little computation involved as the sum of user plus system time is much smaller than the wall clock time passed, which is usually the result of waiting for I/O.

This test demonstrates another performance consideration, and that is that using shared memory can also increase performance. If we compare the flow-cat/flow-filter query performance to that of zlib 9 compressed HDF, we see that HDF fares significantly better even though the total storage size is similar and the compression algorithm is the same. The most probable reason for this difference is the use of pipes (i.e. copying of memory between applications) to connect flow-cat and flow-filter.

Chapter 5

Flowy Implementation

5.1 Architecture Overview and Technology Used

Conceptually the application architecture of Flowy follows the query language architecture with the exception of the splitter and filter stages. The implementation combines the filtering and splitting into one stage. The splitter's only task is to queue and dispatch the already filtered records to the corresponding threads. The process is explained in more detail in Section 5.4 and Section 5.5. Figure 5.1 visualises the high-level application architectures.

5.1.1 Python

Flowy was implemented in Python [27], using PyTables (an HDF v.5 access library) [17] for storing the flow records, and PLY (Python Lex and Yacc) [13] for generating the parser. The Python programming language was chosen because it is a very high-level, dynamically typed language which allows rapid development and provides various convenient features relevant to this project. The main advantage over statically-typed languages such as C, C++ and Java is the easiness of writing clean code for dynamic creation of new datatypes, dispatching of functions and importing of additional modules during runtime. As dynamically typed language, Python has a speed disadvantage compared to statically typed languages. However as discussed in Chapter 4, under common usage Flowy is expected to be limited by I/O performance. Consequently a modern CPU is expected to be idling under normal usage, which means some additional CPU cycles spent on Python inefficiency will probably not affect the total execution time.

5.1.2 PyTables

PyTables is the storage solution used in Flowy. It provides an interface to store large amounts of data. The storage engine itself is based on HDF. PyTables,0 however, does not implement all HDF features [17]. As discussed in Chapter 4, HDF provides quicker read times than other proposed storage solutions. The two storage containers used by Flowy are Table and VLArray. Table is a heterogeneous dataset arranged into rows. Tables have columns which may be of different datatypes. Unfortunately, the Table class does not support variable length columns or rows. The VLArray class represent a variable length (ragged) array. The variable length in this context means that each

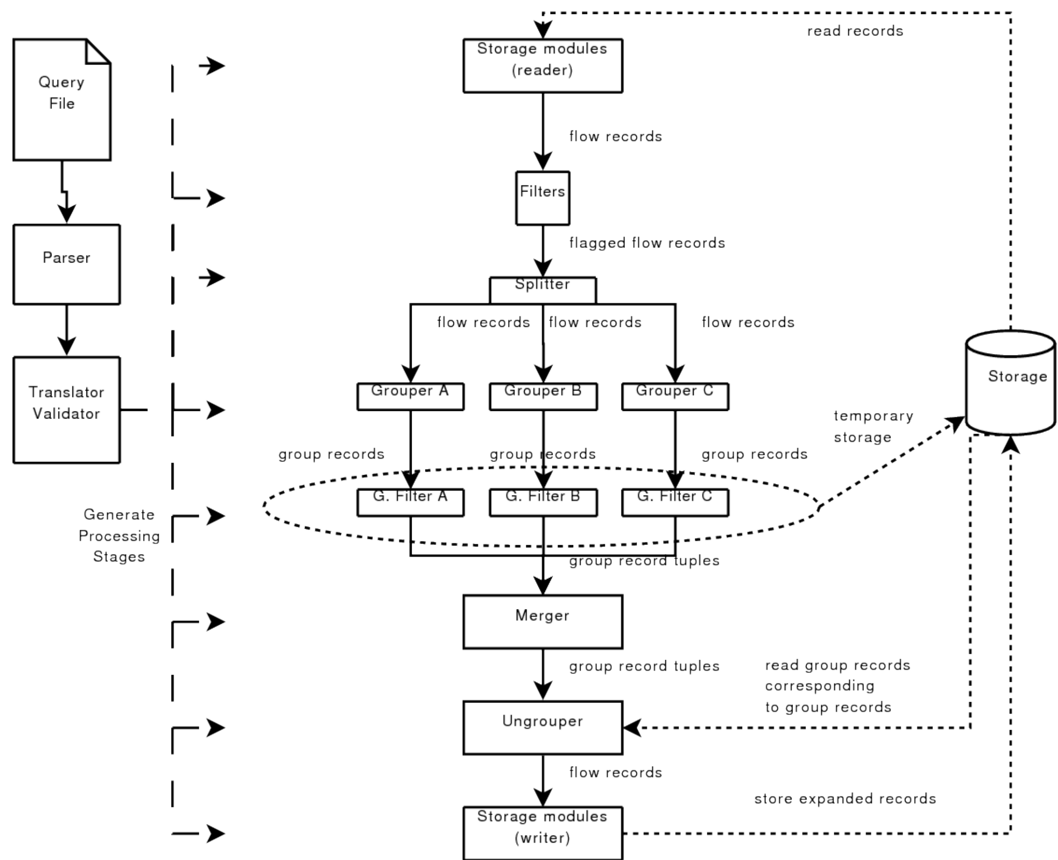

Figure 5.1: Application Architecture

array cell may hold more than one instance of the datatype of the array. The arrays are also extensible (non-fixed length). PyTables is very efficient in terms of storage space and provides good retrieval speed for sequential access of all table rows. Each row number is internally indexed so random retrieval of certain rows is also efficient. There is, however, no support for indexing on other columns than the column numbers in the free version of PyTables. For this reason, the row number is used as key field for records when records have to be retrieved again in the ungrouping phase.

5.1.3 PLY (Python Lex and Yacc)

PLY is a parser generator for Python, which used similar conventions to Lex and Yacc for describing the lexer and the parser. The lexer and parser are written in native Python code, and grammar and lexer rules are contained in Python docstrings [13]. PLY was chosen mainly due to familiarity with the Lex and Yacc conventions. Since query files have relatively simple structure and are not expected to become very large, parser performance is not of great importance for this project.

5.2 Records

The main unit of data exchange through Flowy's processing pipeline is the record. Flowy uses the same class[1] for flow records and group records. This means that there is no difference in treating flow records and group records. Group filters and flow record filters are identical. In fact, once stored, group records become indistinguishable (apart from some metadata) from flow records and they can be used as an input to a full processing pipeline (with filtering, further grouping, merging, and ungrouping).

The *Record* class is not a fixed implementation. The class is dynamically created at runtime through the *get_record_class()* function of the *record* module. The function accepts as arguments the names of the record fields, and optionally their respective datatypes if relevant to the situation, and a list of default values used with the default constructor. The datatypes of each field vary according to the storage format used for the record. The most often used datatypes are the column description datatype of PyTables, since PyTables is the default storage format. The data needed for the creation of the *Record* class is usually provided by the reader object, which has access to the description of the stored records, and is able to create a list of the available fields.

The *record* module also provides the *RecordReader* class. *RecordReader* objects are used to iterate over reader objects, and create *Record* instances from the tuples created by the reader object. *RecordReader* objects provide an iterator over the produced *Record* instances.

The other processing stages are also exposed through an iterator interface, so they act as a wrapper over the previous stage output. The following code demonstrates the usage of the *records* module. It creates a *Record* class for a record that has field *id*, *srcip*, and *dstip*, and instantiates a single instance of the class. The rest of the code demonstrates the creation of a *RecordReader* object, from a reader_object (reader objects use is explained in Section 5.3) .

```
1  Record = get_record_class(['id','srcip','dstip'],
2                                  [uint64, uint32, uint32],
```

[1]The implementation actually produces a new class for each situation depending on the fields present during runtime.

```
3                               [0, 0, 0])
4  new_record = FlowRecord(id, src_ip, dst_ip)
5
6  # actually RecordReader() implicitly calls
7  # get_record_class() with data from reader_object
8  record_reader = RecordReader(reader_object)
9  # print record ids:
10 for record in record_reader:
11     print record.id
```

5.3 Storage readers and writers

Currently there are two storage access modules implemented. The main one is *pytables* for reading and writing records into PyTables HDF files and for creation of such files. The module *ftreader* is used to read records stored in the flow-tools format.

5.3.1 Ftreader

The main class provided by *ftreader* is *FlowToolsReader,* which provides an iterator over the records in a flow-tools records file. The constructor takes as arguments the path to the file, and a list of fields of interest to be included in the produced records. Fields which are not in this list (i.e. they are not of interest for the current execution) are discarded. The fields of interest list creates more compact records, compared to records which have all the fields present in the original flow-tools file. Another useful feature of the *ftreader* module is a Python dictionary which maps the flow-tools field names to the preferred naming format for the *pytables* module. Here is an example of opening a *flow-tools* trace file and reading tuples of source IP and source port and printing them, and of wrapping the FlowToolsReader with a RecordReader object:

```
1  flowtools_iter = FlowToolsReader('path/to/ft-file',
2                                   ['srcaddr', 'srcport'])
3  # flowtools_iter returns tuples of (<srcaddr>, <srcport>)
4  # These tuples are unpacked to the src_ip and src_port variables
5  for src_ip, src_port in flowtools_iter:
6      print src_ip, src_port
7
8  # To read Record instance wrap flowtools_iter with RecordReader:
9  record_reader = RecordReader(flowtools_iter)
10 for record in record_reader:
11     print record.srcaddr, record.srcport
```

The difference between reading as tuples and as records is that when reading as a tuple (through flowtools_iter), we need to know the order of the attributes in order to unpack the tuple correctly. When read as records, the attributes can be accessed as object attributes by their name (as opposed to position). If the access to the attributes is defined dynamically, one can use the Python built-in *getattr* method with string containing the name:

```
1  # print attribute srcaddr:
2  print record.srcaddr
3  # the same with getattr:
4  wanted_attribute = 'srcaddr' # string which can be set during runtime
5  print getattr(record, wanted_attribute)
```

5.3.2 Pytables

The *pytables* module of Flowy provides a more complete functionality than *ftreader*. There are classes and methods for both reading/writing to and creation of PyTables HDF files.

The classes used for reading are *FlowRecordsTable*, and *GroupsExpander*. *FlowRecordsTable* objects are instantiated with a path to a PyTables HDF file, and the expected size of the record ids. The expected id size argument determines the size of the id space. There is no inherent limit on the number of rows in a table, but specifying large id size means that each row will have a field of this size, which for large number of records will incur significant storage overhead. The default size is 4 bytes but if more than 2^32 records are to be inserted in the table the size should be extended to 8 bytes. Alternatively if very few records are to be inserted one ca use 2 bytes id size. The id size can be 1, 2, 4 or 8 bytes.

The *FlowRecordsTable* class is an iterator over the tuples of values for record fields in a HDF file. PyTables does not provide support for table columns with variable-length values, so tables which contain such columns are implemented as a normal PyTables *Table* and several *VLArray* instances which contain the table columns with variable lengths. The Table class of PyTables, provides a heterogeneous (with different data type columns) table with fixed-width rows, while the *VLArray* class represents a homogeneous (one column of one type) table, where row width may vary. Reading and writing to a table with both variable and fixed-length fields is done in parallel over the *Table* and *VLArray* objects, and records are transparently assembled/dataset to/from the separate data containers. These implementation details are transparent, and are not revealed through *FlowRecordsTable*'s API. *FlowRecordsTable* objects are usually wrapped with a *RecordsReader* object, so that attributes can be accessed by name. The *FlowRecordsTable* class also has the method *append()* which appends Record instances, to the table.

The *GroupsExpander* class is an iterator, which takes as input an iterator of group records and an iterator over flow records, and expands the groups to their respective flow records. In other words it does ungrouping.

The *pytables* module also provides, the method *create_records_file()*, which is used to create a new records file based on table description. The following code demonstrates the use of *create_records_file()*, and *GroupsExpander*:

```
1  # Create records file with two columns - IPv4 and port
2  table = create_table_file('path/to/records-file',
3                            {'IPv4':UInt32Col(),
4                             'port',UInt16Col()})
5  # Expand groups:
6  records_iter = FlowRecordsTable('path/to/records-file')
7  groups_iter = FlowRecordsTable('path/to/groups-file')
8  expanded_records_iter = GroupsExpander(records_iter,
9                                         groups_iter)
10 record_reader = RecordReader(expanded_records_iter)
11 for record in record_reader:
12     print record.id
```

5.4 Filters

The *filter* module provides the filtering stage of the pipeline. The main class of the module is the *Filter* class. As with other processing stages, the *Filter* class is an iterator wrapper over a *RecordsReader*. It is important to note that there is only one *Filter* instance for the whole processing pipeline. Instead of using a splitter, which copies each record to the filter for each branch of the pipeline, *Filter* reads every record, and matches it with all the rules in the branches, and creates a record mask. The record mask is a tuple of True/False flags, elements of which correspond to the branches to which the record should be passed to. The filter produces a stream of (record, record_mask) pairs.

5.4.1 Rules

Each filtering statement is converted to a *Rule* class instance against which records are matched. *Rule* instances are constructed from a branch mask, an operation, such as equals, less than, greater than, etc., and arguments to the operation(*Rule(br_mask, operation, arguments)*). Arguments may be either constants, or references to fields from the record being matched. Field references are filled from the current record being matched.

5.4.2 Branches and branch masks

The record mask shows which branches the record being filtered should be passed to. If a Rule test returns True, the record is matched against the next rule in the filter. Instead of creating several *Filter* instances, Flowy uses branch masks, which show which branches each rule affects. Here is a simplified example how two filters *a* and *b*, which are respectively from branches *A* and *B*, are turned into a set of rules with their corresponding branch masks:

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

The Rules created are:

```
1 # protocol(TCP) is evaluated during parsing
2 # to the TCP protocol number 6
3 Rule(((True), (True)), EQ, [Field("prot"), 6])
4 Rule(((True), (False)), EQ, [Field("srcport"), 80])
5 Rule(((False), (True)), GT, [Field("bytes"), 1024])
```

The first argument of the Rule constructor is a simplified branch mask, used here instead of a real *BranchMask* instance for clarity. The first rule corresponds, to *prot = protocol("TCP")* statement. As evident by the branch mask the rule affects both

branches. The second rule corresponds to the *srcport = 80* statement, and its branch mask means that it only affects branch A, whereas the third rule affects only branch B. After matching against all the filter rules, the record is passed to the splitter, which copies it only to branches, for which the record mask is True. If the branch mask is True, the result of the operation is combined with logical AND with the current corresponding value of the record mask. If the branch mask value is False, the record mask is unaffected for the corresponding branch. For example for operation result is False and branch mask (True, False) and the record mask (True, True), the resulting record mask is (False, True).

5.4.3 Subbranches

The previous section discusses only the simple case where all matching rules are connected through logical AND. However Flowy's query language allows for rules to be connected by logical OR. Branch and record masks cannot express whether a Rule's result should be ORed or ANDed with the current record mask. Therefore, the concept of subbranches was added. For statements connected with logical OR, a subbranch is added to the record and branch masks. The first position of a branch/record mask is always for rules which are connected with logical AND, every subsequent subbranch means logical OR group. Here is the previous example extended to include ORed rules:

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

These filters are converted as follows:

```
1 Rule(((True, False), (True)), EQ, [Field("prot"), 6])
2 Rule(((False, True), (False)), EQ, [Field("srcport"), 80])
3 Rule(((False, True), (False)), EQ, [Field("dstport"), 80])
4 Rule(((False, False), (True)), GT, [Field("bytes"), 1024])
```

Notice how the first branch got an additional subbranch compared the previous example. The first rule's branch mask shows that it is one of the rules connected with logical AND, because its branch mask is true only for the first element. The second and third rules' branch masks, show that they are in the same group of ORed rules. When they are matched their results are ORed with the current record mask. Each group of rules connected with OR gets its own subbranch. After the record is matched with all rules, the subbranches are collapsed by replacing them with a single boolean value, which is the result of logical AND of all subbranches.

5.4.4 Composite filters and pseudobranches

Composite filters are used to construct filters from existing filters. The previous example extended with a composite filter for branch C is:

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

In this example filter *c* is a composite of filters *a* and *b*. Conceptually it can be thought as these two filters connected in series. Therefore, parsing and converting this example results in just the addition of branch C, to the branch masks of all the filters by combining the masks (logical OR) for branches A and B but still without combining the subbranches of ORed rules:

```
1 Rule(((True, False), (True), (True, False)), EQ, [Field("prot"), 6])
2 Rule(((False, True), (False), (False, True)), EQ, [Field("srcport"), 80])
3 Rule(((False, True), (False), (False, True)), EQ, [Field("dstport"), 80])
4 Rule(((False, False), (True), (True, False)), GT, [Field("bytes"), 1024])
```

For composite filters which contain filter references connected by logical OR, a concept similar to subbranches exists. It is called pseudobranches. Each filter in a group connected by OR, gets a new branch, whose output at the end is ORed with their rest of the branches in the group. At the end all different OR groups and the AND group get collapsed to a single value using logical AND.

5.4.5 Operators and operator evaluation

Operations used by filter rules are defined in the *operators* module. User-defined operation may be imported using a command line argument (–filter-import=user_filter.py). User defined operations should have the same function name as the operation name. The best practise is to make functions work on arbitrary number of arguments if possible. When query files are parsed and later validated and translated, all references to operations, record fields are checked for accuracy. At this stage functions, whose arguments contain only constants are replaced with their values. This saves some time during execution. However sometimes it may be desirable to call such function for every record. In order to do this, the function should be called with a dummy field reference, which is just ignored by the function. Id is good candidate for this purpose as every record has an id, while other fields might not be present.

```
filter a {

    sth = fun("TCP") # called once before execution
    dyn_sth = fun("TCP", rec_id) # called once for each record
}
```

5.4.6 Reasons for the masking system

The main reason for using the masking system instead of daisy-chaining the output of separate rules is that rule operations can call external functions which can be potentially very slow. Since optimising external function is not possible within Flowy, the masking system ensures each rule operation is called only once. It is possible that there are operations whose result may change between two function calls for the same arguments. Calling each operation only once also ensures that for a given record the same result of the rule is achieved for every branch in which it is used. If one needs different, behaviour across different branches, the may add a different dummy constant argument, for each different branch. This constant is not used in computation, but changes the signature of the rule, so it is considered a different rule. Note that rule operation should be written in such a way, as to ignore the dummy argument.

It is possible to flatten the branch and record masks to a bit mask, which can be evaluated very efficiently and inexpensively checked each time whether it has become zero in order to skip any further rule matching. Currently a different approach is used and only rules which are required by all branches may skip all further rule matching. In the above examples such rule is *prot = protocol("TCP")*, whose branch mask is True for all first sub branches.

5.5 Splitter

The *splitter* module is used for copying records to respective branches based on their record mask. Because after the filters stage, branches don't necessarily go through all records, separate threads are used for each branch in order to be able to do work even if other branches are record-starved.

The *Branch* class constructor takes just the branch name as argument. Its objects provide an iterator over the records put into them. Each *Branch* instance contains synchronised queue, so when further stages in the branch request a record, they block until a new record is available.

The *Splitter* class takes a mapping from branch names to branches objects. It has a method *split()*, which dispatches the record to the branches marked by its mask. The *ready()* method signals any branches blocking on their queues that the splitter has read the last record available from the filter, so that they don't stay blocked forever and remain in a deadlock, waiting for further records. The method *go()* iterates through all the records available from the filter and splits them to the corresponding branches.

Below is an example of creating two branches and splitting already filtered records (using a Filter instance) to these branches:

```
1 brA = Branch('A')
2 brB = Branch('B')
3 splitter = Splitter({'A': brA, 'B': brB})
4 splitter.go()
```

In order to read the records available in a certain branch one has to iterate over it in a thread different from the splitter thread:

```
1 def print_records():
2     for record in brA:
3         print record
4 Thread(target=print_records).start()
```

Note that the branch is a blocking queue, so if the reading is done in the same thread as the one that runs the splitter *go()* method, which puts elements on the queue, a deadlock will occur.

5.6 Grouper

The grouper module takes care of grouping. The main class is *Grouper*. There is one *Grouper* object per branch. The *Grouper* provides an iterator interface over the produced groups. It iterates over its branch's records and passes them through *GrouperRule* objects and *AggrOp* objects.

Group objects contain group records information as well as the first and last records of the group which are needed to match it respectively with absolute and relative rules. Each record is matched against each group using all the grouper rules. If the record matches a group and a certain module, the corresponding aggregation operations are executed on the record.

AggrOp objects are Python callable objects - they can be used as functions, but are initialised with a certain state which they keep between consecutive calls. All of them are defined in the *aggr_operations* module. They aggregate on the given records. Each group has its own set of *AggrOp* objects. *AggrOp* objects are initialised with the record field they should read, the field of the group record they should store the end result in and the data type of the record field. The data type is needed because some operations like average should return the result in the same type as the record field. In other words if the field is a float the result of the average operation is return as a float, if it is an int the return value will be an int. Internally the average is always calculated as a float and the result is rounded if it needs to be returned as int. When an *AggrOp* object is called with no argument it returns the current value of the aggregation operations. User defined aggregation operations may be imported using the *–aggr-import* command line argument. The next example shows how an *AggrOp* object is used to aggregate a union of values:

```
1  un = union('rec_id', 'records', UInt32())
2  rec1.rec_id = 0
3  rec2.rec_id = 3
4  rec3.rec_ie = 8
5
6  un(rec1)
7  un(rec2)
8  un(rec3)
9
10 prints un() # prints 0 3 8
```

GrouperRule objects do not keep state so they are shared between groups. Grouper rules are initialised as either, relative or absolute. Relative rules take the last added record of the group and match it to the new record, whereas absolute rules, match the record to the first record of the group.

5.6.1 Changes to the grouping algorithm

The matching of records to groups is done using a different approach from the one described in the pseudocode in Section 3.3. The nesting order of the loops, which iterate over groups and records are reversed. Rather than tagging the records by passing

over the whole set of untagged records for each new group, the implementation keeps the groups list in memory and for each record it checks whether it belongs to the group. This removes the need for random reads from the slow permanent storage, and achieves the grouping in a single pass over the records. The pseudocode for the implementation is:

```
1  groups_list = new list()
2  function group_record(record):
3      for each group in groups_list:
4          n_matched = 0
5          for each rule in grouper_rules:
6              if match(rule, group, record):
7                  matched ++
8              else:
9                  break
10         if n_matched == n_grouper_rules:
11             add record to group
12             return
13     //If the loop finishes the record has not matched
14     //any group so new group is created:
15     groups_list.add(new group(record))
16     return
17
18 for each record in records:
19     group_record(record)
20
21 return groups_list
```

Note that the groups_list grows over the course of execution. Records later in the loop need to be matched with more groups. The next subsection discusses what measures are taken to limit the number of groups in memory and consequently the number of matchings done for each record.

5.6.2 Groups export and shortcut rules

In order to limit the memory usage some groups may be exported before the last record is read into the grouper. Exporting a group puts its contents into a group record and sends it to the next processing stage (usually a group filter). When a group is exported it is removed from the groups list and it is not matched to any further records. This speeds up execution as it lowers the number of groups against which a record has to be matched.

To export groups early there is a shortcut rules mechanism. Shortcut rules are grouper rules with deltas which match on the *stime* and/or *etime* fields. Since the flow records are ordered in time it is expected that if time constraint is not satisfied for a certain amount of records it won't be satisfied in the future, so there is no need to match this particular group to any other rules.

```
stime < stime rdelta 500ms
```

The rule above means that each consecutive record should start at most 500ms after the last record added to the group. Internally this rule is implemented as shortcut rule because it concerns the *stime* field and has a delta. When the rule fails to match a group to a record, an *UnsatisfiableDelta* exception is raised. The exception is caught

internally, and added to the count for the group which produced it. If the number of exceptions reaches the predetermined threshold, the group gets exported. If the group is matched to a record its *UnsatisfiableDelta* count is set to 0.

A group record is not exported immediately on failing to match a shortcut rule, because flow records are not generally ordered by either their start or end times, but by their export times. In the above example, even if a record fails to match the rule, there might be a record later in the trace that does match it, but was just exported later. Testing with small data sets reveals that indeed sometimes this is the case, and early export partitions longer lasting groups if the threshold is 1.

There are two options which set the thresholds used by shortcut rules. The *–udelta-mult* command line option specifies the threshold at which *UnsatisfiableDelta* exception is raised. It is a multiple of the delta argument. If the unsatisfiable delta multiplier is 3 (*–udelta-mult*=3), in the example above, *UnsatisfiableDelta* exception will be raised if the next record starts more than 1500ms after the last record in the group. The other threshold to be reached is specified by the *–max-udelta* command line argument. It is the number of *UnsatisfiableDelta* exceptions in a row, which have to be produced for a group before it is exported. In the example above if the number is 100, and the unsatisfiable delta multiplier is 3, there have to be 100 records which start at least 1500ms than the last record of the group, before it is exported. The default values for *udelta-mult* and *max-udelta* are 5 and 20 respectively. Lowering the thresholds speeds up execution, but too low numbers result in splitting of groups with long durations into smaller groups.

5.7 Group-Filter

Group filters work like a simplified version of the normal record filter. The branch masks mechanism is not used with group filters because group filters read and export records from a single branch. They use the same operators as flow record filters so any user imported operators will work in group filters as well.

Group filters have another function except filtering of group records. When filtering branches each group filter adds the records to the group record index for the branch, and stores the output records to a pytables file, so that the groups can be read using random access, and if specified in later stages, ungrouped. Time index creation is explained in the next section and its use in the merger is explained in Section 5.9.

Group filter classes are defined in the *groupfilter* module. There are just two classes the *GroupFilter* and *AcceptGroupFilter*. The first class represents an ordinary group filter, which iterates through the records of the grouper of its branch, and passes them through filtering rules. There is no separate class for group filter rules and instead the *Rule* class from the *filter* module is reused. Each record that passes through the group filter rules is stored in the branches group records table.

The *AcceptGroupFilter* class is just a group filter that passes through all records. It is needed in order to run the time indexing task and to store the grouper records for branches which do not have group filters. The *AcceptGroupFilters* are added transparently (they are not specified in the query file).

5.8 Time Index

Time indexes are created by the group filters for each branch. Their main purpose is to limit the number of records that are matched by the merger by returning only group records which potentially satisfy Allen relations. This section is mainly concerned with index creation. Application in the merger stage is explained in more detail in Subsection 5.9.2.

The time index is an index which maps time intervals to the records which occur during this interval. The time space is divided into equal intervals. A record is added to an interval if its execution time overlaps with that of the interval. The length of the intervals can be chosen through the *–tindex-interval* command line option, which specifies it in milliseconds. The default value is 1000ms. Larger values lower the memory needed for the index, whereas smaller values give better resolution. Here is an example pseudocode of how a records are added to the time index:

```
1  interval_length = 1000
2  function get_intervals(start_time, end_time):
3      start = int(floor(start_time/interval_length))
4      end = int(floor(end_time/interval_length) + 1)
5      return [all ints between and including start and end]
6
7  function index_record(record)
8      for each interval in get_intervals(record.start_time, record.end_time):
9          index[interval] = record
10
11 for each record in records:
12     index_record(record)
```

The *TimeIndex* class is defined in the *timeindex* module. Its objects have methods *add* and *get_interval_records*. The *add* method adds new records to the index. Only the records id's are stored in the index. The *get_interval_records* method is used to retrieve the ids of records which occur in the given interval. The result is returned sorted and with no duplicates:

```
1  record1.rec_id = 1
2  record1.stime = 5002
3  record1.etime = 7999
4  record2.rec_id = 2
5  record2.stime = 7001
6  record2.etime = 10001
7  record3.rec_id = 3
8  record3.stime = 11001
9  record3.etime = 11100
10
11 index = TimeIndex(interval=1000)
12 index.add(record1)
13 index.add(record2)
14 index.add(record3)
15
16 print index
17 # prints
18 # {5:[1], 6:[1], 7:[1,2], 8:[1,2], 9:[2], 10:[2], 11[2,3]}
19 # where the key is the interval the value is a list of record ids
20
```

```
21 print index.get_interval_records(5,9) # prints: [1,2]
22 print index.get_interval_records(5,5) # prints: [1]
```

5.9 Merger

The *merger* module provides the classes needed for the merging processing stage. The merger is organised as nested branch loops. Each merger branch loop represents a for-loop over its records. Group record tuples used in merger rules evaluation are built incrementally. Each merger branch loop reads a record from the corresponding group records set and executes the matching rules which have their arguments in the current group record tuple. If the branch is higher in the nesting it won't have the needed arguments for rules which reference lower level branches. After matching the tuple with its rule the branch passes it to the lower level which adds a record from its branch to the tuple and executes any further rules which now have their arguments available. The mechanism is best explained by example.

5.9.1 Merger branch loops

```
merger M{
    module m1{
        branches A,B,D
        A.bytes < B.bytes OR A.bytes < D.bytes
        B.srcip = A.srcip
        A.srcip = D.srcip
        A < B
        A < D
    }
    module m2{
        branches B,C
        B.bytes < C.bytes
        B o C
    }
    export m1
}
```

The above merger will be implemented as four merger branch loops with nesting order A,B,D,C. The branches nesting order is explained in Subsection 3.4.3. Branch loop A will read the group records from branch A. Since it only has record for branch A it cannot evaluate any of the merger rules, therefore it passes the record tuple to branch loop B which reads a record from branch B. At this point the record tuple is (rec_from_A, rec_from_B), so it has enough information to evaluate rule *B.srcip = A.srcip* and Allen rule $A < B$. The order of the arguments does not matter as long as all of the needed information is available. If the record passes these rules, the tuple is passed to branch C. If the rules are not satisfied, branch loop B goes to its next record and evaluates the rules with this record. It does this until it reaches the end of its records, at which point control is returned to branch loop A, which advances to its next record. Note that the rule *A.bytes < B.bytes OR A.bytes < D.bytes* is not evaluated in branch loop B because

it needs a record from branch D for the *D.bytes* field. If a record from branch loop B passes through its rules, the record tuple with this record is passed to branch loop D. Branch loop D evaluates rules *A.srcip = D.srcip*, *A.bytes < B.bytes OR A.bytes < D.bytes* and *A < D*.

If the record tuple extended with a record from branch D passes these rule the extended record tuple is passed to branch loop C. Branch loop C is a so called reject branch loop, because it is used only in a non-export module. It works in a similar fashion to normal branches by extending and passing the records tuple to lower nested branches, but rule results have the opposite effect. If a rule does not match a record from the branch, the record tuple is extended with this record and passed to the lower branch loop. If the record matches the rules, a record tuple rejection is signalled, and the first non-rejecting branch loop continues with its next record. Below is a pseudocode of the merger rule evaluation:

```
1  next(branch_loop, rec_tuple):
2      for record in branch_loop.branch.records:
3          if rec_tupe + record does not match some of the branch_loop rules:
4              if branch_loop is normal branch:
5                  continue
6              else:
7                  # reject branch
8                  if branch_loop->next != Null:
9                      rec_tuple = rec_tuple + record
10                     next(branch_loop->next)
11                 else:
12                     continue
13         else:
14             if branch_loop is normal branch:
15                 if branch_loop->next != Null:
16                     rec_tuple = rec_tuple + record
17                     next(branch_loop->next, rec_tuple)
18                 else:
19                     export(rec_tuple)
20             else:
21                 signal rec_tuple rejection
22         if branch_loop is reject branch and reject is signalled:
23             signal rec_tuple rejection
24         if branch is normal and accept is signaled:
25             export(rec_tuple)
26     if branch_loop is reject branch and reject is not signalled:
27         # this if is outside the loop which means that the branch loop
28         # finished without match, therefore the tuple is accepted
29         signal rec_tuple accept
```

Note that there are two ways to export a group record tuple. One is if the branch is the last branch loop and it is not a reject branch loop, it directly exports a tuple, which matches its rules. The other way is if a reject branch signals accept, to a non-reject branch loop. When a branch exhausts its records, it signals an acceptance, because it has not matched any rules. If the upper branch loop is a reject-branch loop, it disregards the signal, and continues its execution until it exhausts its own records. If the upper branch loop is normal it exports the record, because it is the last non-rejecting branch loop, so the accept means that the tuple was not rejected by any reject branch loop and it was accepted by current branch loop and the ones above.

5.9.2 Merger branch loops index optimisation

In order to improve the efficiency of the merger operation, a TimeIndex is used to find only records which have the possibility of satisfying the Allen relations set in the merger. If there is an Allen relation $A < B$ the branch loop does not need to iterate over all of its branch records for each record in A. It can iterate only over records which occur after the current record. In general if the left argument of an Allen relation is known this imposes a restriction on the possible right arguments. The looping aspect of the merging algorithm changes as follows:

```
1  function find_possible(Allen_op, rec, target_branch):
2      interval = calculate the needed interval
3                 based on Allen_op and rec.stime and rec.endtime
4      return target_branch.time_index.get_interval_rows(interval)
5  function next(branch_loop, rec_tuple):
6      for record in branch_loop.possible_records:
7          for each ar in Allen relations:
8              if ar.first_arg ?= branch_loop.br_name:
9                  possible_rec = find_possible(Allen_op, rec, ar.second_arg)
10                 next_branch_loop = find branch with name ar.second_arg
11                 next_branch.possible_rec =
12                         = next_branch.possible_rec intersect possible_rec
```

The find_possible function uses the Allen operation and the record's time interval to find which records occur in the interval for which the Allen operation holds. For example if a record from branch A has a start time 1000 and end time 3000, and the Allen relation is B d A (B during A), the records from B which have to be read by the lower level branch loop - B are the ones which occur between times 1000 and 3000. If the relation is B > A delta[2] 5000, the records which have to be iterated are the ones which occur between times 3000 and 8000. In a similar fashion all Allen relations can produce a finite interval which should be checked by the lower branch loops. The produced interval is usually smaller than the total time interval of the group records trace, so a significant speed-up is achieved. In order to find the actual records in this interval the TimeIndex object get_interval_records is used. Note that the possible_records of the other branch are not overwritten, but are added as a set intersection of the possible_records set by other branch loops. This is needed because, two higher branch loops may together set the possible_rec attribute. For example there might be two Allen relations concerning branch C: C d A and C d B. Assuming the current record tuple contains records from A and B with time intervals respectively (1000, 2000) and (1500, 5000), branch A will set the possible_rec interval as (1000, 2000). Since both Allen relations should be satisfied, when B adds its Allen relation information it forms the intersection with the one set from A, so the final time interval which is used to create possible_rec is (1500, 2000).

Intervals which satisfy Allen relations are obtained through an object from the *allen_index* module. Allen_index module objects are callable objects, which take the left-hand argument of the Allen relation and return an interval in which the right-hand argument should occur based on the left-hand argument start and end times, and the time of Allen relation.

[2] For explanation on what deltas in Allen relations mean see Subsection 3.4.3

5.9.3 Merger module classes

The merger module provides branch loop classes *Merger*, *MergerBranch, MergerLastBranch, MergerRejectBranch, MergerLastRejectBranch*. The *Merger* class represents the first branch loop in the merger it differs by ordinary *MergerBranch*, because it does not have any rules and iterates over all of its records since there are no higher nested branch loops which would provide restriction on the possible records through Allen relations. The *MergerBranch* represents an ordinary merger branch loop, whereas *MergerRejectBranch* represents a reject branch, as explained in Subsection 5.9.2. The corresponding LastBranch classes objects just don't call the next() method on the next branch loop as there is not next branch loop.

The merge module also provides a *MergerStorage* class, which encapsulates a FlowRecordsTable which holds the group records tuple output of the merger.

There are some implicit restrictions on how mergers are constructed, which are discussed in Section <> (Merger validator).

5.10 Ungrouper

Ungrouper objects are used to ungroup merger output. They use the group record tuples to expand them first into groups and then to records. If the output file name for the whole processing file is "filename.h5" the expanded groups are stored in "filename-<merger_name>-groups.h5". If the groups records are the desired output, the generation of expanded flow records file can be suppressed with *–no-records-ungroup* command line option. Ungroupers are initialised with a merger file and export order, which defines the order into which the separate tuple elements are exported. For example if the export order is ['C','D','A'], every three iterations over the *Ungrouper* object will return consecutively group records from the specified branches:

```
1 u = Ungrouper('merger-file', ['C','D','A'])
2 #print group records from the ungrouper
3 for gr in u.groups:
4     print gr
5 # prints group records in the order CDACDACDA...
```

In order to print the flow records one needs to iterate over the records attribute of the *Ungrouper* instance:

```
1 #print flow records from the ungrouper
2 for rec in u.records:
3     print rec
```

The order of the flow records is determined by the order of the groups which are expanded go get the records. Records are ordered within the group, by their record id.

5.11 Parser and statements

The *parser* module holds a lexer and parser definitions for the flowy query language. When statements are parsed they are converted to instances of the classes from the *statements* module. The objects of the classes in the *statements* module hold the statement information. The type of the statement is represented by the object's type (class), and the object holds information about statement's substatement, values, and line number. Most query language statements have their own class in the *statements* module,

but some such as *Module* and *Rule* are reused. For example the *Rule* class is used for both filters and group filters.

The parsed statements are converted into processing stages by the various validator modules, which also check on some semantical constraints. A *Parser* has a list of statements for each statement type. Each type of validator checks the corresponding list for errors and checks for consistency with the other validators. The validation of statements goes from filters to ungroupers. Below is an example of how the parser and validators are chained to produce the processing stages:

```
1 parser = Parser('query_file.flw')
2 filter_v = FilterValidator(parser)
3 splitter_v = SplitterValidator(parser, filter_v)
4 grouper_v = GrouperValidator(parser, splitter_v)
5 gr_filter_v = GroupFilterValidator(parser, grouper_v)
6 merger_v = MergerValidator(parser, gr_filter_v)
7 ungrouper_v = UngrouperValidator(parser, merger_v)
```

Each validator object produces a list of its statements' implementations and stores it in the *impl* attribute. For example the grouper validator *grouper_v* stores the actual *Grouper* instances list in *grouper_v.impl*. Each validator is initialised with both the parser and the previous stage validator, because it needs to connect its processing stages to the previous processing stages. The impl list is used to create the processing stage threads. Here is an example of making GroupFilter threads and starting them:

```
1 # Create a list of threads which have as targets the go()
2 # method of each GroupFilter instance:
3 gf_threads = [Thread(target=gf.go)for gf in gr_filter_v.impl]
4 # Start the threads. This calls each group filter's
5 # go() method in a new thread:
6 for gf_thread in gf_threads:
7     gf_thread.start()
```

5.12 Filter validator

The filter validator checks whether any references to fields of the record are actually present in the input file. It also checks whether all filters referenced in composite filters are defined, whether there are duplicate filter names and warns if there are unused filters (filters which are not present in any of the branching statements). After the filters consistency is checked, a filter implementation is created, which is a *Filter* object from the *filter* module. The details on how branch masks and rule implementations are created and work are discussed in Section 5.4. *FilterValidator* objects are initialised with a *Parser* object.

5.13 Splitter validator

There is not much to validate for a splitter statement. The *SplitterValidator* class just creates the *Branch* instances and the *Splitter* instance which contains them. Its initialisation arguments are the *Parser* object and the *FilterValidator* object which has already created the Filter implementation. The information about the branches and their correspondence to the record masks is taken from the *FilterValidator* object.

5.14 Grouper validator

The grouper validator like the filter validator checks for the presence of the referenced fields. It also checks for duplicate module names and duplicate grouper names. For each grouper statement the grouper validator check the aggregation statements and links module aggregation statements to the corresponding modules. An important detail in the creation of the *Grouper* instance is that three aggregation operations are added for each grouper even if they are not present in the query file. They are *union(rec_id) as records*, *min(stime)* and *max(etime)*. The first is used to store the records present in the group, for later ungrouping. The *min(stime)* and *max(etime)* are used to establish the group's start and end times. These fields are used later for the time indexing of the group records. The grouper validator is initialised with a parser and a splitter validator.

5.15 Groupfilter validator

Group filter validator does the same tasks as a filter validator, but the check for the referenced field names is done against the branch grouper aggregate statements, as they define the fields present in the group records. It also does not group rules through branch masks as each group filter reads records from a single branch. The objects of this validator are initialised with a parser object and a grouper validator. If some branches don't have group filter defined, the *GroupfilterValidator* instance fills the gaps with *AcceptGroupFilter*s.

5.16 Merger validator

The Merger validator, imposes some restrictions on the way mergers are defined. Apart from checking for the presence of the referenced fields and branches, it makes sure that all Allen relations arguments are in the order imposed by the branches statements (see Subsection 3.4.3 for details on iteration order). Ordering of the Allen relations' arguments is possible because each Allen rule has an inverse. For example if the branches iteration order is *A, B, C* and the Allen rules are $A < B$, $C > A$, $C\ d\ B$, the rules are converted to $A < B$, $A < C$, $B\ di\ C$. The reordering is useful because it allows for earlier evaluation of Allen restrictions, and allows for passing of restrictions in a single direction. As mentioned in Subsection 5.9.2 Allen indexing operations take the left-hand argument of the Allen relation and return the interval in which the right-hand argument should occur. In the example above if the rule $\mathrm{C} > \mathrm{A}$ is evaluated in this order, the restriction imposed on A by the records in C will be found only after iteration in the branch loop A has already started. Having only one direction of passing of Allen restrictions greatly simplified the implementation of passing Allen relation restrictions. Furthermore when the rule's arguments order is reversed the restriction $\mathrm{A} < \mathrm{C}$ is evaluated higher in the nesting hierarchy, which means that lower levels are expected to need less rule matching evaluations. This allows for some optimisation by putting branch loops with more group records lower in the nesting hierarchy, so that by the time they are iterated, Allen restriction reduce the number of records which need to be checked.

Merger validator objects also attach the rules to the branch loops, based on the availability of their arguments. In the example above the rules $A < C$ and $B\ id\ C$ will be attached to branch loop C because the records from branch C are available only

after branch loop C is reached. The rule $A < B$ is attached to branch loop B. Non-Allen rules are also attached on the availability of all their needed arguments. It is important to note that Allen indexing rules need only an argument for the left-hand side of the relation, therefore, the Allen indexing rules for $A < B$ and $A < C$ are attached to branch loop A and B *id* C - to branch loop B.

Another check done by the merger validator, is whether Allen rules with the same set of arguments are connected by an *OR*. For example if rules $A < B$ and A *o* B, are defined on separate lines, this means implicit *AND*. Since Allen relations are exclusive of one another, connecting them by an *AND* makes the unsatisfiable. Therefore an error is raised if the Allen relations concerning a pair of branches are not connected with an *OR*:

```
# raises a syntax error:
module m1 {
    ...
    A < B
    A o B
}
# correct way to do it:
module m1 {
    ...
    A < B OR A o B
}
```

Yet another check done by the merger validator is that each branch loop is reachable through an Allen relation or a chain of Allen relations from the first branch in the exported module.

```
merger M {
    module m1 {
        branches A, B, C
        ...
        A < B
        B < C
    }
    module m2 {
        branches D, E
        ...
        D < E
    }
    export m1
}
```

In the example above branches D and E do not have any Allen relation to the first branch in the export module m1, so the merger validator will raise SyntaxError.

```
merger M {
module m1 {
```

```
    branches A, B, C
        ...
        A < B
        B < C
    }
    module m2 {
        branches C, D
        ...
        C < D
    }
    export m1
}
```

This second example won't raise an error, because all branches have relation to the first exported branch. The most indirect Allen relation chain in this example is *A < B -> B < C -> C < D*.

5.17 Ungrouper validator

Like the splitter validator, the ungrouper validator does not need validation, so it just creates the Ungrouper objects needed for the ungrouping stage.

Chapter 6

Testing and conclusions

6.1 Speed testing

6.1.1 Execution time and input size

In order to find out the time complexity of flowy in practise, the same query was done with 15 minute, 30 minute and 1 hour traces.

The query used for the test, groups all records for TCP transfers to or from port 80 into two branches representing the two direction of an HTTP transfer. The merger stage merges the two branches into group records tuple representing downloads (request transferred bytes are less than response bytes, and request starts before response).

The results are shown in table 6.1. As one would expect from a two branch query the execution time rises as the square of the input set. The 15 minute trace does not follow this trend, because its execution time is still dominated by the time needed for the filter stage, because of the small number of group records produced by the particular trace. Although it was not precisely timed, it was obvious that for 30 and 60 minute traces the longest part of the execution was the merging stage, whereas for the 15 minute trace it was the filtering stage.

time length of NetFlow traces	15 minutes	30 minutes	60 minutes
approx. number of flow records	200 000	400 000	800 000
execution time in minutes	1m59s	16m11s	77m11s

Table 6.1: Execution time against length of NetFlow trace

6.1.2 Speed compared to flow-tools

During planning stages it was expected that flowy will provide an appreciable speed-up compared to flow-tools due to superior storage. Unfortunately flowy does not live up to the expectations. It is significantly slower than flow-tools, because its performance is not I/O bound. The same query as the one described in Section 4.6 (dstport = 12121) run on a smaller records set (single day with 250MB worth of records) takes in flow-tools 45 seconds. Flowy's execution time for this query on the same set of records converted to pytables HDF format is 19:30 minutes. On the other hand running the same query with two, three and four branch queries in flowy, each branch having a

different filter selecting a different port, took respectively 24:18, 29:00 and 32:38 minutes. Looking at CPU usage statistics during flowy's exectution shows that it does not wait too much for I/O. The probable reasons for the low performance are discussed in the next section.

6.2 Profiling flowy execution

In order to find the parts of flowy that are most detrimental to performance, a profiling test was done. The standard Python profiler does not support profiling of multiple threads, so a profiler function written by Maciej Obarski, was used to achieve multi-threaded profile of the program [20]. The profiler, gives the total number of calls to each function and the total wall-clock time spent between function call and function return.

The total run time as measured by the time it took the *main()* to finish was 643 seconds. Note that the total sum of functions execution times is more than the execution of *main,* because execution times are wall-clock times between call and return which do not exclude the time during, which the CPU was executing something else.

In order to find out which flowy functions are the worst performers, function with less than 1000 calls were filtered out from the statistics. This was needed because function call time does not exclude the time during which a function was waiting for another function to return, so functions with the longest call times, were actually functions called just a few times like flowy's main() function, whose time is dependent on its internally called functions. Number of function calls alone is not a good metric, because their cumulative time may be much less than the time spent on some function with less calls. Execution time is also not very indicative because as mentioned above, functions with small number of calls, which take long time usually call other functions, which are the real bottleneck. The metric used for finding worst performing functions was time spent per functions call, which is just the division of the time spent in function call divided by the number of calls.

From the sorted by execution time per call functions list, several draw attention. The top badly performing function which does not call other functions within flowy was the Queue get() method which took 468 seconds. This method is called by a grouper reading records for its branch. Since the Queue class is actually a blocking queue, the get method blocks when a branch is record starved. This shows that for this particular query large part of the records get filtered out, and consequently groupers spend a lot of time waiting for a record. It also suggests that the most important thing to be done in order to speed up the grouper stage and consequently the whole pipeline is to optimise the filtering stage.

The next function on the worst performers list is the read_rows_list. It mostly called by the merger to read lists of group records. Although this function needs to do random reads from group record files, in this particular case the number of groups for each branch was rather small (about a thousand per branch), so there is probably another reason for the bad performance. One possible reason is each read operation for groups involves reading from both a PyTables *Table* and *VLArray* because of the limitation of PyTables which does not allow both columns with different data types and columns which hold variable length values in the same table (see Subsection 5.3.2 for more details), and concatenating the result into a tuple which is used to create a *Record* instance. The problem with reading from two or more tables, is that the results have to be concatenated, which due to the fact that flowy uses immutable tuples as return

type for the read_rows_list means that the contents have to be copied, which as the next paragraph explains may be a really serious problem.

The last of the worst performing functions turns out to be *reset()* function used in the filtering stage to reset a record branch mask into its initial state for the next record. This function took total of 142s, and the only thing it does is to call deepcopy[1] on a dictionary. Since this function runs in a single thread and as mentioned above, the other processing stages threads wait significant amount of time for the filter to provide a new record, this function seems to be causing the most significant slow down of all other functions.

In order to test what impact deepcopying has, flowy was changed to work with a shallow copy in the reset() function. The result was that the simple filter query (dstport = 12121), took 16:08 minutes instead of 19:30 minutes. Working with a shallow copy results in incorrect results, but the number of function calls is the same, with both deep and shallow copies, so the difference in performance comes from the less time it takes for a shallow copy compared to deep copy. It seems that lowering the amount of copying in flowy can have significant performance impact.

The other functions on the worst performing list are either indirectly called by flowy, or are flowy functions which call other already discussed flowy functions.

6.3 Possible improvement

6.3.1 Lowering the amount of copying

As discussed in the previous section, one of the possible reasons for bad performance is the deepcopy of some objects. Lowering the amount of copying may result in performance improvement. One of the functions which use copy the most is reset() function of filter module's BranchMask class. One way to lower the need for this operation is to remove the branch mask mechanism, or remove it only for simple queries. Another way to improve branch masking is to use a module written in lower-level language such as C, which may achieve more efficient memory management.

Another way to reduce internal copying of objects is to avoid immutable types for containers (tuples in particular) as any modification on them result into creation of new object, which needs some of its state copied.

6.3.2 Using PyTables in-kernel searches

PyTables offers the so called in-kernel mechanism to filter records from tables, while reading them. PyTables in-kernel searches provided arithmetic comparison rules, and the PyTables' author claims they result in 10-fold increase of speed of record retrieval [17]. In order for this to be implemented, arithmetic comparison filter rules will have to be transformed into PyTables in-kernel search queries, and flowy's PyTables access module will have to be extended with the capability to read from PyTables filtered records. Unfortunately in-kernel searches are available only for *Table* instances and not for *VLArray* objects so they can only be used for fixed width-columns.

[1]Deepcopy is a Python function, which copies data structures such as dictionaries by making copies of each element in the dictionary instead of just creating new references to the old objects.

6.3.3 Multithreaded Merger

Although flowy uses separate threads for different processing stages, the most work-intensive stage - the merger is single-threaded (separate mergers run in separate threads though). It is possible to split the outermost branch loop among several threads. If each thread stores its results in a separate location, there won't be need for blocking on shared resources as all other operations on shared resources are reads. This should provide a performance increase proportional to the number of threads if the storage can keep up with the increased number of reading threads.

6.4 Conclusion

At this stage flowy may be slower than other tools, but it still has some features, which make it more useful for some tasks than them. It has merger operation, allowing grouping of aggregated records based on time relations. Such processing is not available in other NetFlow analysis tools. Another advantage is that flowy is easy to extend with user defined functions for filtering and aggregation. Despite its performance issues there is room for improvement, so there is potential that flowy may become a viable network flow analysis tool

Bibliography

[1] Common Data Format (CDF). available online at http://cdf.gsfc.nasa.gov/.

[2] flow-tools. available online http://www.splintered.net/sw/flow-tools/.

[3] FLOWD. available online at http://www.mindrot.org/projects/flowd/.

[4] Hierarchical Data Format. available online at http://www.hdfgroup.org/.

[5] Indexing in HDF. available online at http://www.hdfgroup.uiuc.edu/RFC/HDF5/hdf5Indexing/.

[6] NFDUMP. available online at http://nfdump.sourceforge.net/.

[7] NfSen. available online at http://nfsen.sourceforge.net/.

[8] Stager. available online at http://software.uninett.no/stager/.

[9] The SiLK Reference Guide. available online at . http://tools.netsa.cert.org/silk/reference-guide.htm.

[10] *NetFlow Services Solutions Guide*, 4th Edition edition, January 2007.

[11] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, November 1983.

[12] Francesc Altet and Ivan Vilata. OPSI: The indexing system of PyTables 2 Professional Edition. Exhaustive description and benchmarks about the indexing engine that comes with PyTables Pro. available online at http://www.pytables.com/docs/OPSI-indexes.pdf.

[13] David M. Beazley. PLY (Python Lex-Yacc). available online at http://www.dabeaz.com/ply/ply.html.

[14] B. Claise. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information, jan 2008.

[15] P. Deutsch. GZIP file format specification version 4.3, may 1996.

[16] Andy Dustman. MySQLdb: a Python interface for MySQL. available online at http://mysql-python.sourceforge.net/MySQLdb.html.

[17] Scott Prater Vicent Mas Tom Hedley Antonio Valentino Jeffrey Whitaker Francesc Altet, Ivan Vilata. PyTables User's Guide: Hierarchical datasets in Python - Release 2.0.4. available online at http://www.pytables.org/docs/manual/.

[18] Vladislav Marinov. *Design of an IP Flow Record Query Language*. Campus Ring 1, 28759 Bremen, Germany, January 2009.

[19] Bill Nickless. Combining Cisco NetFlow Exports with Relational Database Technology for Usage Statistics, Intrusion Detection, and Network Forensics. In *LISA '00: Proceedings of the 14th USENIX conference on System administration*, pages 285–290, Berkeley, CA, USA, 2000. USENIX Association.

[20] Maciej Obarski. Profiling Python Threads. available online at http://code.activestate.com/recipes/465831/, 2006.

[21] Markus F.X.J. Oberhumer. LZO Real-time compression library. available online at http://www.oberhumer.com/opensource/lzo/.

[22] Tobi Oetiker. The rrdtool manual,. available online at http://ee-staff.ethz.ch/oetiker/webtools/rrdtool/manual/index.html.

[23] J. Quittek, T. Zseby, B. Claise, and S. Zander. Requirements for IP Flow Information Export (IPFIX), October 2004.

[24] Anna Sperotto. Using SQL databases for flow processing. Joint EMANICS/IRTF-NMRG Workshop on Netflow/IPFIX Usage in Network Management, October 2008.

[25] B. Trammell, E. Boschi, L. Mark, T. Zseby, and A. Wagner. Specification of the IPFIX File Format, October 2008.

[26] Brian Trammell. fixbuf. available online at http://tools.netsa.cert.org/fixbuf/.

[27] G. Van Rossum. *The Python Language Reference Manual*. Network Theory Ltd., September 2003.

[28] Ward van Wanrooij and Aiko Pras. Data on Retention. In *Ambient Networks, 16th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, pages 60–71, 2005.

[29] Wikipedia. Bitmap index –- Wikipedia, The Free Encyclopedia, 2008. [Online; accessed 16-December-2008].

List of Figures

2.1 NetFlow v.9 Export Packet Example [10] 8

3.1 Network Flow Query Language Architecture 10
3.2 Flow level breakdown of a simple FTP transfer 10
3.3 Capturing FTP download of Big Files with the IP Flow Filtering Framework 14

5.1 Application Architecture 26

List of Tables

2.1 NetFlow v.5 record format[10] 7

4.1 Storage size and increase ratio compared to flow-tools format, time to insert data into database/HDF file, and time to do a query 23

6.1 Execution time against length of NetFlow trace 46