Flowy improvement using MapReduce

Peter Nemeth

Computer Science
International University Bremen
Campus Ring 1
28759 Bremen
Germany

Type: Guided Research
Date: May 9, 2010
Supervisor: Prof. Dr. Jürgen Schönwälder

Abstract

Network flow data represents aggregated information of packets and frames with common attributes that traverse the network. Analyzing this data can be crucial for companies and institutions to optimize network performance, secure the network and increase response time to security related problems, thus decrementing expenditures. Having reliable and fast analyzer applications with expert personnel stabilizes the network to a great extent for any organization.

The Computer Networks and Distributed Systems research group (CNDS) at Jacobs University has designed a stream-based query language that searches for patterns in network flow data records. The first implementation - called flowy - has just been completed, but it experiences high execution time compared to other flow data analyzers (e.g. flow-tools). The goal of the thesis is to investigate whether Google's popular MapReduce algorithm can be used to overcome the bottleneck that is present in the current version of flowy, and attempt to create a technical implementation of it.

Contents

1 Introduction 3
1.1 Flow data processing . . . 3
1.2 MapReduce . . . 4

2 Flowy 5
2.1 Query language overview . . . 5
2.2 Flowy implementation . . . 6

3 MapReduce Frameworks 7
3.1 Apache Hadoop . . . 7
3.2 Disco . . . 8

4 Slicing of flow data for flow queries 9
4.1 Filters only . . . 10
4.2 Groupers . . . 10
4.3 Merger . . . 13

5 Practical advancements 13
5.1 Disco installation . . . 14
5.2 Running flowy as a map function . . . 14
5.3 Slicing input . . . 15
5.4 Modifications on flowy . . . 15

6 Conclusions and further required work 15

1 Introduction

Analyzing network traffic, especially Internet traffic is an emerging area of research. The network protocol Netflow/IPFIX was initially designed by Cisco to allow the collection of IP traffic information that traverses a network. Network flows have many definition, but the traditional Cisco definition [1] is: a unidirectional sequence of packets sharing a 7-tuple, values being source IP address, destination IP address, source port, destination port, IP protocol, ingress interface and IP type of service. Netflow version 5 and above contains several additional fields, such as timestamps of the start and end of the flow. If collected, these records can grow to vast numbers and contain lots of valuable network traffic information and patterns amongst them. The analysis of these records and extraction of relevant information from them requires certain flow analysis tools.

Flowy [2] is a flow-data analysing tool which is based on the query language defined in Vladislav Marinov's M.Sc. thesis [3]. It takes a query and an HDF (Hierarchical Data Format) [4] file containing the flow-data as input, and as output returns an HDF file with the flows matching the pattern set by the query. HDF is a technology suite that makes the management of extremely large and complex data collections possible. Flowy comes forth from the line of existing analyzers with it's ability to conduct searches of even intricate patterns in the flows. Unfortunately, the implementation with its current slow execution time is not competitive enough. There are several distinct ways to try to tackle this problem; in this thesis, we attempt to use the advantages of distributed computing on flowy, applying a MapReduce algorithm on it through a framework, and adjusting the program itself to the needs of this new environment.

1.1 Flow data processing

Flow records are being exported from routers that have the NetFlow feature enabled and are collected by a flow collector, as shown on Fig. 1. The number of these flow records can be significantly high, for instance at UNINETT - a Norwegian company that maintains a national research based computer network - with 27 routers, 207 interfaces and a sampling rate 1:100 more than 30GB of flow records are collected every day [5]. Extracting relevant information calls for the usage of network flow analyzers.

Flow analysis has several applications like, gathering network statistics and usage patterns for billing and network planning, or intrusion and attack detection. There are already existing applications for it, such as flow-tools [6] and nfdump [7], and these are ample if one needs to run simple filters on the input. For detecting any more sophisticated patterns, such as checking whether W32/Blaster worm is present on the network based on the recorded flows, these tools are insufficient.

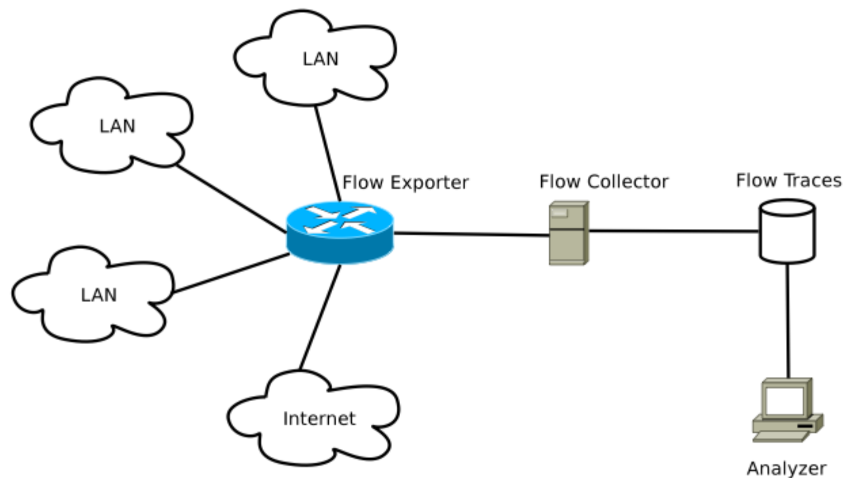

Figure 1: Network flow collection

Flowy, on the other hand, is capable of handling such complicated pattern that can be derived from the fields that the recorded flows have.

1.2 MapReduce

MapReduce is a software framework introduced by Google to support distributed computing on large data sets on clusters of computers [8]. Its main benefit is the parallel execution of independent subparts, gaining a considerable advantage to centralized computing, if the actual distribution is feasible. Another description of it by Google is that it is an abstraction that allows software engineers to perform "simple computations while hiding the details of parallelization, data distribution, load balancing and fault tolerance"[9]. They claim that this technology can manipulate more data faster than any conventional database.

MapReduce contains two main functions: Map and Reduce:

1. The Map function (user defined) takes an input and produces key/value pairs based on the algorithm it contains. The values with the same keys are then grouped and passed to the Reduce function.
2. The Reduce function takes an intermediate key and the set of its values, then merges the values to form a possibly smaller set of values, and return them to the user.

MapReduce is used in a wide range of applications and architectures, such as distributed grep, distributed sort, document clustering, machine learning and many others. Its inputs and outputs are usually stored in a distributed file system.

Fig. 2 shows the overview of the planned use of MapReduce for our system, with the original input being sliced and passed to distinct nodes running flowy via the Map function, then their output being combined by the Reduce function and returned to the user.

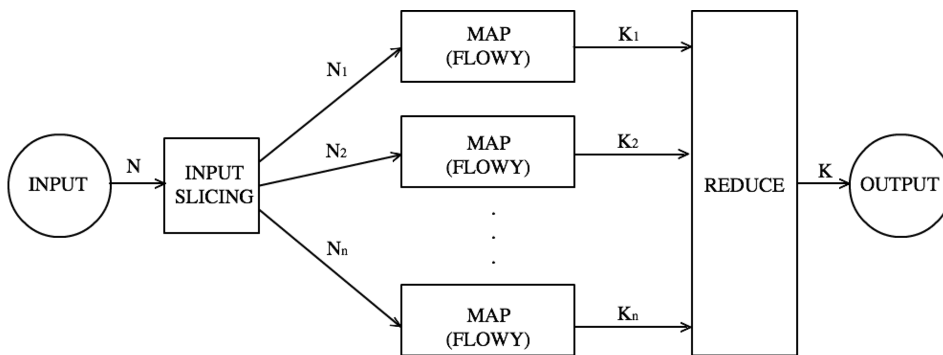

Figure 2: MapReduce applied on flowy

2 Flowy

Flowy is a flow-data analyzer written in Python, implementing the IP flow record query language proposed by the aforementioned thesis. It is capable of handling queries written in the given markup, and utilizes HDF as storage for the flows - a set of file formats and libraries designed to store and organize large amounts of numerical data. Unfortunately, even though flowy can identify patterns in flows that other flow analyzers such as flow-tools or nfdump can not, its execution time is significantly higher than theirs in similar queries. A sample query (get records with a specific destination port) run on a smaller records set (single day with 250MB worth of records) takes 45 seconds in flow-tools and 19:30 minutes on flowy.

2.1 Query language overview

The query language follows a stream-oriented approach. It consists of six processing elements that are connected to each others with pipes. Each of these receive an input stream, perform some operations on it and pass the result to the next element. After receiving the search query, flowy goes through these elements (they take place in ascending order, shown on Fig. 3):

1. Splitter: takes a stream of flows as input and clones them on more output branches
2. Filter: takes a stream of flows and returns the ones matching the filtering rules
3. Grouper: takes a stream of flows and groups them based on grouping rules
4. Group-filter: takes groups of flows and returns the ones matching the group-filtering rules
5. Merger: merges seperate groups from the branches based on its module's rules
6. Un-grouper: ungroups the groupped flow records and returns them as the output.

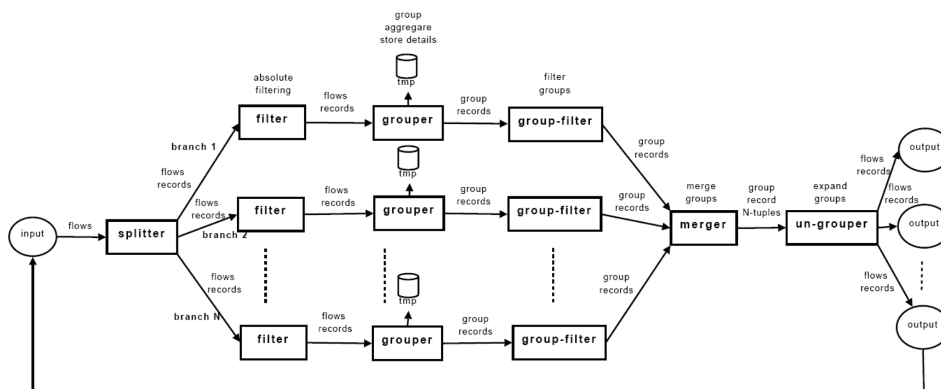

Figure 3: Query language structure

2.2 Flowy implementation

Flowy takes the query file as its mandatory command line parameter. This query file is a plain text file, where all the elements of the query language have to be defined, in addition to the input and the output files - in the latter case allowing more than one. All these files have to be in HDF format.

When flowy receives the query, it first parses it and then applies the extracted rules on the input sequentially: each rule is applied on the input (or the already processed input after the filter stage) at its own stage. The elements of this sequence are strictly established by the query language, and in the current version of flowy, all of them have to be defined. With the current default settings, in case of a successful run, the results matching the pattern set by the query will be saved in the output

file “ungrouped.h5”, which can be printed for an overview using the printhdf tool from flowy’s package.

3 MapReduce Frameworks

Several MapReduce implementations are available. Some of these lack scalability and are not distributed. As we are looking for a scalable, distributed, open-source solution, the feasible list of candidates was narrowed down to two: Apache Hadoop and Disco.

3.1 Apache Hadoop

The Apache Hadoop project [10] develops open-source software for reliable, scalable, distributed computing since 2007. It includes several subprojects, from which the relevant one for our research is called “Hadoop MapReduce”. Briefly, it is a framework with the main idea of utilizing distributed computing on any number of machines, sharing the input and the computations to each based on manually set configuration, then combining the output to the final result with a responsible node checking for correctness and consistency.

Apache Hadoop is a Java software framework that enables applications to work with thousands of nodes and petabytes of data [11]. It uses a distributed filesystem (HDFS - Hadoop Distributed Filesystem), and above that comes the MapReduce engine consisting of a Job Tracker and a Task Tracker. The Job Tracker gets the jobs from client applications and pushes them to the Task Tracker, for execution, while trying to keep the work as close to the data as possible. The Job Tracker is aware of the locations of the data at all times, and organizes the data chunks belonging together to the same racks, thus reducing network traffic on the main backbone network. Amongst several other companies using it [12], the Yahoo! Search Webmap is the leading Hadoop application as of now that runs on a more than 10,000 core Linux cluster and produces data that is now used in every Yahoo! Web search query.

Dependencies for Hadoop are only an ssh server (e.g. openssh-server), ssh client and SUN java 6 sdk. All input data needs to be loaded into HDFS and staged (shared amongst the nodes) first.

Apache Hadoop is used by a considerable number of companies for various projects. It is robust, and guarantees a very high chance of error-free computation. However, many issues are raised having to implement flowy through Hadoop, the main being that it is written in JAVA. To be able to run any Map/Reduce jobs written in other languages (in our case, Python), we have 3 options:

1. Hadoop Streaming, a utility that allows creation and execution of external Map/Reduce scripts. For simple commands and scripts, it is sufficient, however, running flowy through this Streaming utility poses several problems. As flowy uses HDF as it's input and output, if it's streamed through Hadoop the input file would have to be already copied to the HDFS. As the dependencies needed to process HDF cannot be streamed, the HDF cannot be handled properly. In addition, there are no other options to slice the file to share it amongst the nodes from there due to the same reason. This makes flowy incapable of working, rendering this option unsuitable for us.

2. Pydoop is a Python MapReduce and HDFS API for Hadoop. Built as a wrapper around the C++ API, pydoop allows one to build python Map/Reduce functions. However, staging the HDF input file properly in the HDFS would still be a problem, just as the wrapping of modules. Even though it would be feasible to use it, it would need to much modifications on the original flowy, and many tedious workarounds Hadoop. Thus, this would not be a suitable solution either.

3. Translating Python scripts to JAVA through Jython. This is the official solution suggested on the Apache Hadoop page, yet it only supports the standard Python library and most modules (such as the ones flowy utilizes) do not work with it.

In conclusion, all these options either change the language of flowy, or need serious modifications to be applied on it for file-handling and basic working. Thus, even though the framework seems excellent for many other applications, for flowy it would not be optimal.

3.2 Disco

Disco [13] grew out of a practical need at Nokia Research in 2008, as they kept accumulating more and more data and did not have good tools to handle it. They examined Hadoop first, but realized that it cannot be integrated well with the existing infrastructure without taking a deep dive into the Hadoop codebase, and they also - just as in our case - were reluctant to give up Python for the data analysis tasks. Consequently, Nokia started developing its own MapReduce framework, called Disco. The system has been written in Erlang, a language designed for building highly concurrent, fault tolerant, heterogenous distributed systems. MapReduce is a prime example of a system with the aforementioned characteristics.

Disco is an implementation of the Map-Reduce framework for distributed computing. As the original framework, which was publicized by Google, Disco supports parallel computations over large data sets on unreliable cluster of computers. This makes it a useful tool for analyzing and processing large datasets without having to bother about difficult technical questions related to distributed computing, such

as communication protocols, load balancing, locking, job scheduling or fault tolerance, which are taken care of by Disco [14].

Technically, Disco works by extracting bytecode from the Python functions, and distributing them to the nodes by the Disco master node. Map and reduce functions are treated as different units, they get packaged up and sent to the available worker nodes, where they are processed and executed. Erlang is used to orchestrate the work of these independent Python processes.

The Map and Reduce jobs can be written easily in Python as functions that take a few fixed parameters and return a tuple or a list of tuples. A python jobfile must have a map and a reduce function defined, a disco module imported and a new job started, which encompasses the definition of a map function, reduce function, input file and several other optional attributes.

Dependencies are an ssh server (e.g. openssh-server), ssh client, Erlang/OTP R12B or newer, Lighttpd 1.4.17 or newer and the cJSON module for Python. Input data can be anywhere on all the nodes, just the individual absolute paths have to be set up to reach them.

Disco has been successfully set up on a Ubuntu 9.10 test node to run flowy as the map function without any difficulties, so any consequent input/output passing to flowy through Disco should be completely straightforward from here. No extensions of Disco or the installation of additional frameworks were needed.

In conclusion, out of the two MapReduce frameworks, Disco suits our project more than Hadoop, so we chose to use it for the actual implementation.

4 Slicing of flow data for flow queries

One of the main problems with running multiple instances of flowy on several nodes is the distribution of the input amongst these instances. The whole essence of flowy resides in the fact that it can get the specific flows that match the given pattern from a vast amount of interconnected and independent flows. Thus, most traditional slicing methods fail, but to prove this we should go from the least complex scenario (query only utilizes filters) to the most complicated scenario (groupers and merger). For better understanding of the problem, Fig. 4 depicts the packets forming flows that traverse a network over time. The squares are packets, the groups are the flows, and the dashed lines show possible slices, pointing out the issue of separating packets that belong together, and flows that could be connected together via grouping rules.

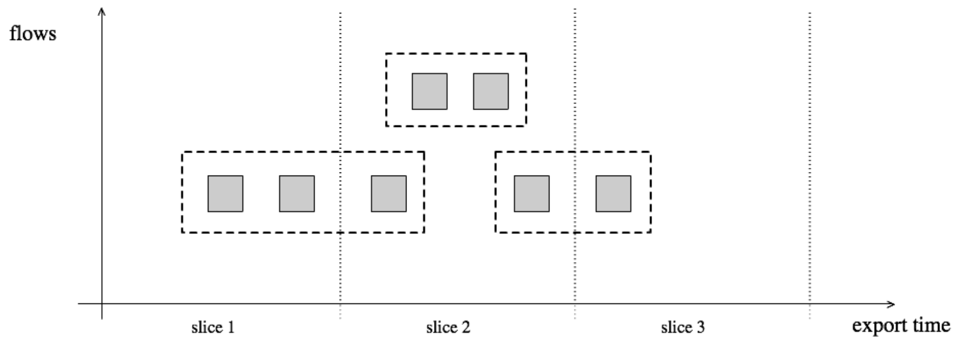

Figure 4: The slicing problem

4.1 Filters only

Theoretically, the simplest query we can have only contains one (or multiple) filters applied in a chain. Practically, this is not allowed at the moment, as only the Splitter and the Ungrouper are allowed to be undefined in flowy's queryfile. However, assuming a correction of flowy to allow empty query parts as well, at this stage we could apply two different intuitive slicing mechanisms:

1. Rewriting the query, adding some boundaries such as between two given export timestamps (`unix_secs` attribute). This method would also benefit from flowy being able to skip records (and recognize that it can skip) without reading in all values.
2. Generating separate input files for each node. As copying is an expensive operation, this method is only efficient if copying is unavoidable in order to distribute the data amongst the nodes.

An additional desirable feature would be to keep the query the same as much as possible for all the nodes. The more nodes we have the more time it would take to create the specialized queries for each node.

4.2 Groupers

Adding groupers and group filters to a query poses a higher complexity than just having filters. This also includes the ability to write queries that make slicing data very problematic, as the groupers can group flows from any segment of the flow trace. We can observe that the slice boundaries can not be static, but have to depend strictly on the input data due to the high flexibility of these operations. As none of the former two methods can deal with non-fixed overlaps amongst the flows, we need new methods.

3. Extending flowy to make it aware of slice boundaries, and allow processing of post-slice records. One logic could be for flowy to take two more command line parameters (two unix timestamps as the boundaries, mentioned in point 1) to define the slice to be processed, then extract the groups found in it based on the query. Having done this, for each slice, skim the rest of the input, but do not add any new groups to the initial set of groups collected, just gather any additional relevant data for the existing ones.

Fig. 5 illustrates the concept of preslice, slice and postslice for the respective map instances. The pre-slices are skipped for each node, the actual slices and post-slices read and processed, from the slice the relevant groups created. The post-slices which will not spawn new groups, just be added to the existing ones if matching, or discarded if not.

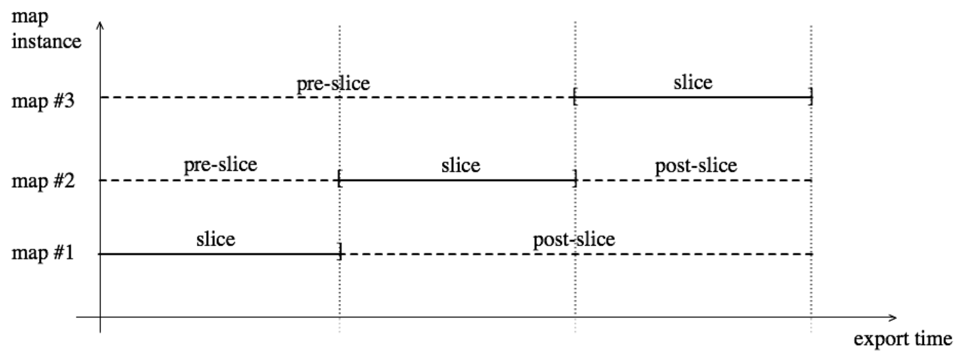

Figure 5: Pre-slice, Slice, Post-slice for 3 map instances

Currently, this slicing mechanism is used, the input being evenly distributed amongst the nodes. Based on the network's characteristics, it might also be beneficial to try looking for low traffic spots for the actual cut, such as the nighttime on Fig. 6 [15]. In this case the risk of cutting records belonging to the same groups could significantly drop for most queries. However, this would require more information on the local minima of the traffic/time relation, and is highly network specific.

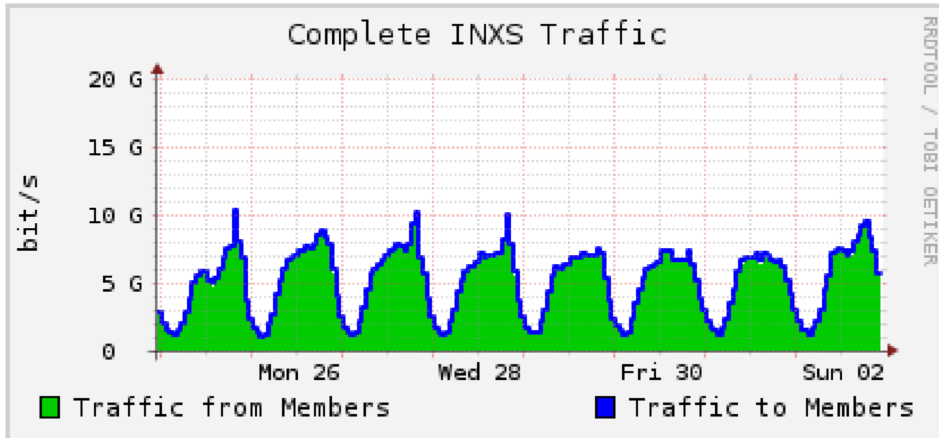

Figure 6: Network traffic distribution over time (8 days) - INXS Munich

The pre-slice neglecting and post-slice sweeping logic can cause an issue of many redundant fragment groups being created that need to be removed at the reduce step. There are 3 characteristics of these not needed groups: they have the same end time as the original; the records contained in the "fake" group are pure subset of the original group; the delta records must all start pre-slice as seen from the slice producing a "fake" group.

The problem with the superfluous groups is illustrated on Fig. 7 (the numbered squares are records in the flow trace), where the groups would look like the following on the respective nodes: slice 1 would create group $g_{1,1}$=1,2,3,4,5, slice 2 group $g_{2,1}$=3,4,5 and slice 3 group $g_{3,1}$=5 and group $g_{3,2}$=6. In this example, we have to check the group's records in the reduce function whether they are the pure subset of a group in the respective pre-slice. $g_{1,1}$ is a proper group, as there exists no pre-slice for slice 1. Upon checking $g_{2,1}$ in slice 2 we see that it is the pure subset of $g_{1,1}$ so we can discard it. Same procedure takes place with $g_{3,1}$ in slice 3, but we keep $g_{3,2}$ as the record of the group does not match any records present in previous groups.

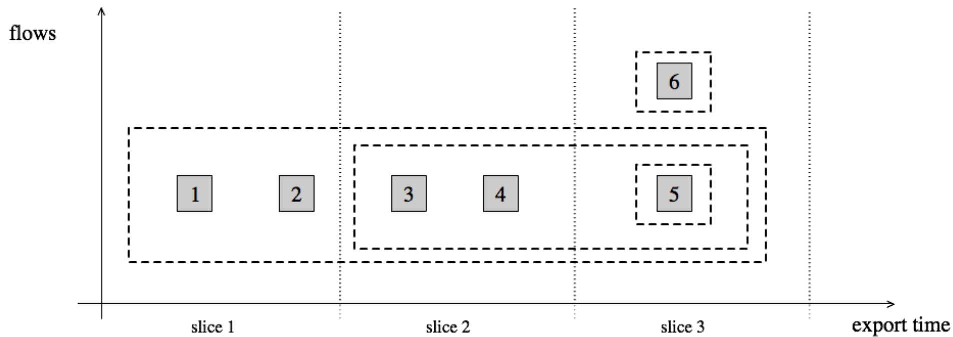

Figure 7: Multiple occurrances of a group and its subsets

4.3 Merger

The Merger is the part with the highest complexity and represents a considerable component of the poor execution time of flowy. It takes streams of group records, applies the merging rules on them, then passes them on as tuples of group records. The simplest method would be to split flowy into two parts and have Splitter, Filter, Grouper and Groupfilter in the map function, and the Merger in one single reduce instance. Unfortunately, this would not make flowy faster, as the merger has the highest algorithmic complexity in flowy, and it should use distributed computing's advantages to the fullest potential.

As an alteration of the previous idea, we could already try to merge what can be merged in each map instance, and merge the rest in the reduce instance via a stripped version of flowy which accepts group records and merges them.

For each node, the slice boundaries (first / last) can be passed to the local instance of flowy by command line arguments, so for each node, we can have the same, unaltered instances.

5 Practical advancements

Firstly, experiments with the MapReduce frameworks have been conducted, and after Hadoop failed to satisfy the needs of our research, Disco has been installed and tested. Flowy was run as its map function on a single node and it succeeded, and the work progressed in the direction of the slicing problem. Having the input sliced, modifications on flowy were made to have its functionality adjusted to the concept of pre-slice skipping and post-slice sweeping presented before. The upcoming sections discuss the actual modifications and extensions made.

5.1 Disco installation

Disco installation is relatively simple for only one node: after installing the dependencies, run make in the folder of the latest (up to date it is version 0.2.4) extracted Disco package, enable passwordless login via ssh to localhost and start Disco with “disco master start” (on non system-wide installation this needs to be run from the bin folder). Flowy needs to be installed on the same system with all its dependencies for Disco to be able to run it in the map function.

Theoretically, in case of multiple nodes, for each node, the exact same is needed: Disco, flowy and their dependencies (and the passwordless ssh login to all nodes). We have to designate a master node that will orchestrate the work amongst the other ones, and start disco as “disco master start”. On all worker nodes we have to run “disco worker start”. If everything is set up, we need to create a job on the master node and run it. Actual testing on more than one node has not yet taken place.

5.2 Running flowy as a map function

We have to create a Disco job function, which requires the definition of map and reduce functions (both can be done as external modules as well), and an input file (also can handle empty input by setting it to the string “raw://none”). A so called `map_reader` function can be manually set up, but not necessary as the default is set to be a `line_reader`, which reads each line of the input file and spawns new sub-jobs from them.

Several different approaches can be used to tackle how the input is being passed to the map function, we give the job function’s input field a text file containing slice boundary times for each node per line. When Disco reads this file, it spawns a new map function for each line read, passing the contents to the map function in parameter “e”.

The map function uses Python’s `subprocess` method to start an instance of flowy with the “e” parameter concatenated to its command line arguments containing the HDF’s slice boundaries for the given node, thus all nodes will have unique HDF slices to work with.

Although by default the reduce function expects a tuple or list of tuples from the map function as a result to work with, it is possible to give any arbitrary tuple back and instead of working with that, modifying the reduce function to work with the output files (ungrouped.h5) produced by flowy on the nodes.

5.3 Slicing input

In the previous section the input file passed to Disco was already mentioned; the creation of the file itself is done right before the job function is initialized. An external module's (sliceFileCreator) `sliceIt` function is called with two parameters: the HDF to be checked for boundaries, and the number of nodes used as workers. `sliceIt` extracts the earliest and latest export timestamps from the HDF and divides it equally by the number of nodes to be used, writing the boundary times of the slices in a textfile called "slices.txt"; each line containing the first and last times bounding a single slice.

5.4 Modifications on flowy

The executable of flowy has been modified to accept three command line arguments instead of one, so now it takes a query file as the first argument, then the first and last export timestamps for a given slice. These two additional arguments have been added as extra parameters in the flowy_exec module that deals with executing the subcomponents, and also have been added to the Filter and Grouper files (and their validators).

The first occasion when the records are actually read within flowy is at the Filter step, so here the skipping of the pre-slice has to be done, so that the records passed to the following components only contain the given slice for the node and the post-slice. This is made in a naive way, stepping through the records till the start time of the boundary is found, then applying the filter rules only on the records onwards. The filter rules must be applied on the post-slice records as well to speed up the grouping phase by eliminating non-relevant records.

The Grouper module loops through the records and checks whether they match the grouping rules. Once a feasible candidate is found, it is matched with the already existing groups, and in case it does not match with any of them, a new group is created. At this point, a condition has been altered, restricting new group creation for records with maximum export timestamp of the second boundary of the slice, thus allowing new group creation for only the records that are in the slice. All records from the post-slice are still matched and added to already created groups.

6 Conclusions and further required work

This research paves the path to be taken for the improvement of flowy by using distributed computing, yet there is still work to be done to fully finish the implementation. The steps to be taken to accomplish the project are the setting up of a reduce function aggregating the outputs by a modified flowy merging the groups

that could not be merged on the individual nodes and eliminating duplicate records; the creation of this stripped version of flowy; deployment on a cluster and exhaustive testings and comparisons.

References

[1] B. Claise - Cisco Systems NetFlow Services Export Version 9. RFC 3954, Cisco Systems, October 2004

[2] Kaloyan Kanev, Jürgen Schönwälder - Flowy - Network Flow Analysis Application, Master's thesis, Jacobs University Bremen, August 2009.

[3] Vladislav Marinov, Jürgen Schönwälder - Design of a Stream-based IP Flow Record Query Language, Master's thesis, Jacobs University Bremen, May 2009.

[4] HDF5 Documentation. Also available online at http://www.hdfgroup.org/HDF5/doc/doc-info.html

[5] Arne Øslebø - An overview of traffic analysis using NetFlow. Available online at http://www.ist-lobster.org/events/tutorial-2006/netflow.pdf

[6] flow-tools. available online http://www.splintered.net/sw/flow-tools/ , Last accessed 15th January, 2010

[7] NFDUMP. available online at http://nfdump.sourceforge.net/ , Last accessed 15th January, 2010

[8] Jeffrey Dean, Sanjay Ghemawat - MapReduce: Simplified Data Processing on Large Clusters, Google Inc., December 2004

[9] Introduction to Parallel Programming and MapReduce, Google Code University, http://code.google.com/edu/parallel/mapreduce-tutorial.html, Last accessed 9th May, 2010

[10] Apache Hadoop Official site - http://hadoop.apache.org/ , Last accessed 8th May, 2010

[11] Wikipedia. Hadoop — Wikipedia, The Free Encyclopedia, 2010, Last accessed 15th January, 2010

[12] Apache Hadoop - Powered by http://wiki.apache.org/hadoop/PoweredBy, Last accessed 20th April, 2010

[13] Disco official site - http://discoproject.org/ , Last accessed 8th May, 2010

[14] Personal communication with Ville H. Tuulos, creator of Disco

[15] Internet Exchange Point in Munich (INXS MUC), http://www.inxs.de, Last accessed 9th May, 2010