NTNU
Norwegian University of
Science and Technology

# Distributed NetFlow Processing Using the Map-Reduce Model

**Jan Tore Morken**

Master of Science in Computer Science

| | |
|---|---|
| Submission date: | June 2010 |
| Supervisor: | Svein-Olaf Hvasshovd, IDI |
| Co-supervisor: | Morten Knutsen, UNINETT AS |
| | Arne Øslebø, UNINETT AS |

Norwegian University of Science and Technology
Department of Computer and Information Science

## Problem Description

Examine how the map-reduce programming model can be applied to NetFlow processing in order to efficiently analyze very large sets of NetFlow data. Implement a number of common NetFlow data processing operations 1) using a map-reduce framework, and 2) using an optimal approach without any framework constraints. Execute relevant benchmarks and compare the efficiency and scalability of the implemented systems. Based on this, evaluate the viability of using general frameworks for distributed NetFlow processing.

Assignment given: 15. January 2010
Supervisor: Svein-Olaf Hvasshovd, IDI

## Abstract

In this Master's thesis we study the viability of using the map-reduce model and frameworks for NetFlow data processing. The map-reduce model is an approach to distributed processing that simplifies implementation work, and it can also help in adding fault tolerance to large processing jobs.

We design and implement two prototypes of a NetFlow processing tool. One prototype is based on a design where we freely choose an approach that we consider optimal with regard to performance. This prototype functions as a reference design. The other prototype is based on and makes use of the supporting features of a map-reduce framework.

The performance of both prototypes is benchmarked, and we evaluate the performance of the framework based prototype against the reference design. Based on the benchmarks we analyse and comment the differences in performance, and make a conclusion about the suitability of the map-reduce model and frameworks for the problem at hand.

# Preface

This Master's thesis is the result of my participation in the course *TDT4900 Computer and Information Science, Master Thesis* at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) during the spring semester of 2010.

The assignment was formulated in cooperation with UNINETT AS in Trondheim, and follows up on previous work done during my specialisation project during the autumn semester of 2009.

I wish to thank Morten Knutsen and Arne Øslebø at UNINETT AS for providing me with an interesting assignment and for putting their resources at my disposal. Great thanks also go to my supervisor at NTNU, Professor Svein-Olaf Hvasshovd, for providing me with invaluable guidance and always useful feedback.

*Trondheim, June 11, 2010*

Jan Tore Morken

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

NetFlow data can provide important information about the traffic that passes through a network. However, as the network load grows, the amount of collected flow data also increases. The result is that NetFlow data collections grow too large for conventional, single-threaded processing tools to process efficiently.

In previous research we have studied common NetFlow processing techniques and found that the map-reduce model accommodates well for the kind of processing operations we want to perform [1]. As map-reduce frameworks eliminate much of the complexity of writing parallel, fault-tolerant data processing programs, we wish to study how our processing operations can be implemented in the map-reduce model and evaluate the efficiency of such implementations.

## 1.2 Requirements

This section describes the general requirements that form the basis for the evaluations done in this thesis.

### 1.2.1 Scalability

Ideally, a processing tool should be able to operate efficiently on any number of CPU cores on any number of machines. If this is the case, the tool should also scale to any volume of data given that enough hardware is available to store and process it. We refer to this as scalability: The ability of the tool to continue to perform well as more data and more computers are added to the processing chain.

The general goal is to eliminate both disks, CPUs and individual machines as performance bottlenecks. Removing CPU as a bottleneck is particularly important, as disk performance is still continuing to increase dramatically, especially with the introduction of solid state drives (SSDs). The fact that multiple hard drives can be combined in RAID setups for increased speed, without making any modifications to software, also puts emphasis on CPU parallellism. The introduction of solid state drives (SSDs) is also an indication that the performance of storage devices will continue to evolve.

### 1.2.2 Query Response Time

Somewhat related to scalability, the query response time is the time it takes from the moment the user submits a query to the system, until the system returns a useful result to the user. It is worth keeping in mind that "useful results" in this context may also include partial results that only include a subset of the data, e.g. returning results in increments of higher precision.

Existing tools generally provide good response times, much thanks to their simplicity. Earlier studies have shown, however, that general processing frameworks tend to introduce a runtime overhead that makes doing ad hoc queries impractical. This is much due to the extra overhead added by job scheduling.

### 1.2.3 Operations

Based on our previous research we have identified three operations that are essential in a NetFlow processing tool. The first is *filtering*, which involves selectively choosing what data to include in the processing and not. The second operation is *aggregation*, which involves grouping and counting records that share given attributes. Finally there is *sorting*, which involves reordering the output data in an ascending or descending fashion based on one or more attributes.

## 1.3 Structure of Thesis

In Chapter 2 we introduce the NetFlow technology and the map-reduce model. We also discuss various aspects concerning the efficiency of the map-reduce model.

In Chapter 3 we describe the methodology we use throughout our research.

In Chapter 4 we present two processing tool designs. One describes a baseline architecture that is independent of any map-reduce frameworks but is designed for high performance. The other design describes an architecture that can be implemented using a general map-reduce framework.

In Chapter 5 we evaluate two existing map-reduce frameworks in order to determine which is the most suitable one for our requirements.

In Chapter 6 we explain the details of how we have implemented both architectures presented in Chapter 4.

In Chapter 7 the performance of our two implementations is evaluated and compared in order to determine how well a map-reduce based system performs compared to the baseline system.

Finally, in Chapter 8 we draw our conclusions about the viability of map-reduce and general map-reduce frameworks as a basis for a NetFlow processing tool.

# Chapter 2

# Background

This chapter introduces the most important technologies we are working with in the course of this project. Section 2.1 introduces the NetFlow technology, as well as typical areas and methods where NetFlow is utilised. The map-reduce programming model is explained in Section 2.2, along with a discussion of a number of issues that affect the efficiency, scalability and error resiliency of map-reduce implementations.

## 2.1 NetFlow and IPFIX

The NetFlow technology was originally introduced by Cisco Systems [2]. It provides a mechanism for exporting summaries of traffic that is observed in networking equipment such as routers and switches.

The Internet Engineering Task Force (IETF) has launched an effort to standardise the export of flow data [3]. Termed Internet Protocol Flow Information Export (IPFIX), this new standard builds on – and is, in many respects, identical to – Cisco's NetFlow version 9 [4, 5].

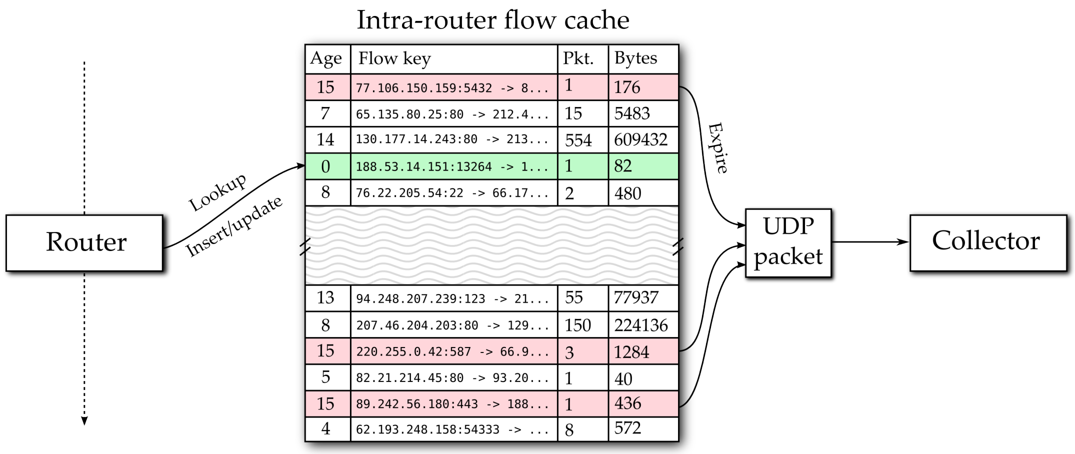
### 2.1.1 Architecture

As the name of the technology implies, NetFlow revolves around flows. A flow is defined as a set of packets within a time frame that share a certain set of attributes. These attributes have been defined to be the following [6]:

- source IP address
- destination IP address
- source port number

- destination port number
- IP protocol type
- IP type of service
- input interface

The first six attributes in the list above are simply IP packet header fields. The last attribute – "input interface" – is the device interface number on which the IP packet was observed.



| Age | Flow key | Pkt. | Bytes |
|---|---|---|---|
| 15 | 77.106.150.159:5432 -> 8... | 1 | 176 |
| 7 | 65.135.80.25:80 -> 212.4... | 15 | 5483 |
| 14 | 130.177.14.243:80 -> 213... | 554 | 609432 |
| 0 | 188.53.14.151:13264 -> 1... | 1 | 82 |
| 8 | 76.22.205.54:22 -> 66.17... | 2 | 480 |
| 13 | 94.248.207.239:123 -> 21... | 55 | 77937 |
| 8 | 207.46.204.203:80 -> 129... | 150 | 224136 |
| 15 | 220.255.0.42:587 -> 66.9... | 3 | 1284 |
| 5 | 82.21.214.45:80 -> 93.20... | 1 | 40 |
| 15 | 89.242.56.180:443 -> 188... | 1 | 436 |
| 4 | 62.193.248.158:54333 -> ... | 8 | 572 |

Figure 2.1: NetFlow router cache

Figure 2.1 illustrates a typical approach to capturing NetFlow data from network routers. The system that is capturing packets – such as a router, switch or network probe – keeps a cache of current flows, named intra-router flow cache in Figure 2.1. Each row listed in the cache represents a flow. Each flow has a unique flow key, as well as counters for the total number of packets, bytes, as well as the age of the flow.

When a packet is captured, a lookup is done in the cache to find an existing flow with the same attributes. If no matching flow is found in the cache, a cache entry is created to represent the flow. If one is found, the flow counters – such as byte and packet counts – are incremented based on the current packet.

All flows in the cache have a limited life span, and will eventually expire from the cache and be sent to the collector. There are a number of events that can cause a flow to expire. An obvious case is when a TCP connection is closed. Flows will also expire after a certain amount of idle time, and long-lasting flows that are not idle may still be forced to expire after a given amount of time.

Collection of NetFlow data can be done in a number of ways. It is common to capture and export flow data from routers or switches to a central collector,

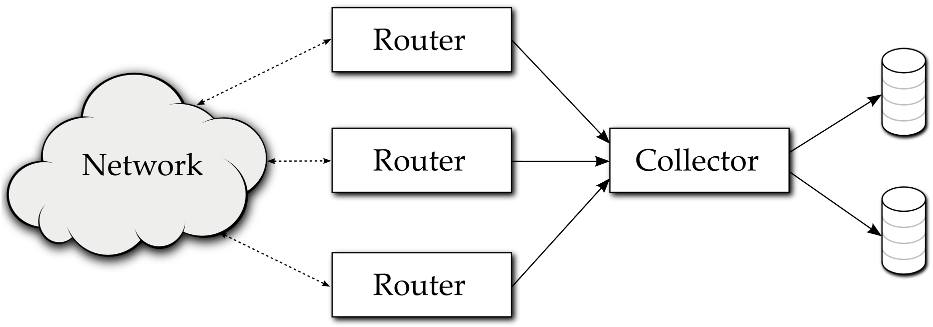


Figure 2.2: NetFlow export architecture

as shown in Figure 2.2. In networks with high traffic volumes, one may choose to sample the traffic (i.e. only capture every $N$th packet) in order to reduce the load on the CPU in the network equipment.

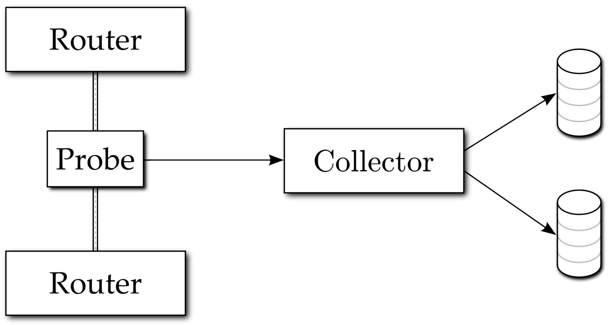


Figure 2.3: Network with traffic probe

As illustrated in Figure 2.3, another option is to attach monitoring probes to the network in some manner. Such probes – often consisting of high-end server systems with dedicated hardware for capturing traffic – are capable of processing greater traffic volumes, eliminating the need for sampling and thus enabling the collected traffic data to be more precise. This puts a correspondingly higher load on all subsequent parts of the processing chain.

When flows have expired on the capture device, they are exported to a flow collector. If the NetFlow protocol [6] is used, this is done by bundling a number of flows in a UDP packet and sending this packet to a collector host. A collector process on this host listens for such packets and stores incoming flows in some format. The storage format is largely application specific, since usage patterns can vary a lot.

### 2.1.2 Data Storage

The collected flows must be sent to a collector process, and the collector process stores the flow data in some format. The specifics of this format are determined by the program. Some collectors [7, 8] will process and perform aggregation on data as it is received, and will often store only aggregated data. This ensures quick access to a predefined set of statistics, such as aggregated counters per IP address, port numbers and such, but limits the user to that predefined set of queries.

Other collectors [9, 10] will pass flows directly to storage without any processing. In this case flow records are typically stored sequentially in plain files, often in an unordered fashion.

Much due to the fact that so many different flow storage formats exist, the IPFIX standard also includes a standardised file format for flow record storage [11]. This format is simply a serialised stream of binary IPFIX messages written to plain files. This maintains full flexibility with regard to how the data can be used later, and allows the use of any IPFIX features in the stored data.

### 2.1.3 Use Cases and Existing Tools

The possible uses of flow data span a wide area. Among the most common usage areas is collecting network link statistics for use in capacity planning or accounting. For instance, one may wish to determine the total volume of data each IP address or subnet in a customer network has transported to or from an external network (e.g. the Internet). To do this you would typically start by filtering the flow records to exclude any internal traffic, then aggregate records by IP address and summarise byte and packet counts for each address.

Another possible use of flow data is anomaly detection [12, 13]. By studying the change in flow data over time and looking at deviations in the data from expected values, it is possible to detect anomalous traffic patterns in a network. In the case of a worm outbreak, for instance, you may see a sudden increase in traffic on a given port number or to/from IP addresses that are normally quiet. In this case you may want to both aggregate on some field as well as well as sort the resulting aggregates to extract the most active hosts or the busiest port numbers.

As can be seen from the examples we have just mentioned, there are three operations that are very commonly performed on flow data: filtering, aggregation and sorting. All these operations are commonly supported by even simple flow processing tools such as Nfdump [9] and SiLK [10].

## 2.2 The Map-Reduce Model

The map-reduce programming model was introduced by Google in 2004, in a paper describing the architecture and some limited aspects of their non-public implementation [14]. Since then, the map-reduce model has attracted a lot of attention, and a number of publicly available implementations have appeared [15, 16, 17].

One of the primary advantages of programming in the map-reduce model is that the complexity of writing a concurrent program is abstracted away, and to a certain degree eliminated. The programmer implements two functions: map and reduce. The framework takes care of invoking these functions on the input data and scheduling parallel execution of them across any number of computation nodes.

This simplification of a program into only two relatively simple functions may come at a cost, however. Some computational tasks are simply not suitable for processing with map-reduce. One example is joining two large data sets, as well as any problem that requires interdependent computations to be performed.

### 2.2.1 Concept and Design

One of the aspects that make writing programs in the map-reduce model so simple is the structure in which programs must be written. As previously mentioned, all map-reduce jobs must be expressed as two functions: `map` and `reduce`. It is not necessary to pay attention to issues like threading, synchronisation, locking, and other aspects related to distributed or threaded execution.

The classic example used to illustrate how a map-reduce program can be implemented, and how the map and reduce functions work, is a word counting program. This program accepts a set of documents as input and emits an index of words and the number of occurrences of each word.

```
function map(String key, String document) {
  for word in split_words(document) {
    emit_intermediate(word, "1");
  }
}

function reduce(String word, Iterator values) {
  int count = 0;
  for v in values {
    count += to_integer(v);
  }
  emit(word, to_string(count));
}
```

Listing 2.1: Word count map-reduce program

Listing 2.1 shows the pseudocode of such a word counting program. As can be seen, the `map` function accepts a key and a value as input, and emits a number of intermediate key-value pairs as output. The `reduce` function accepts a key and a *list of values* as input. In this particular case it simply calculates the sum of the values in the input list and emits a final key-value pair for each input key.

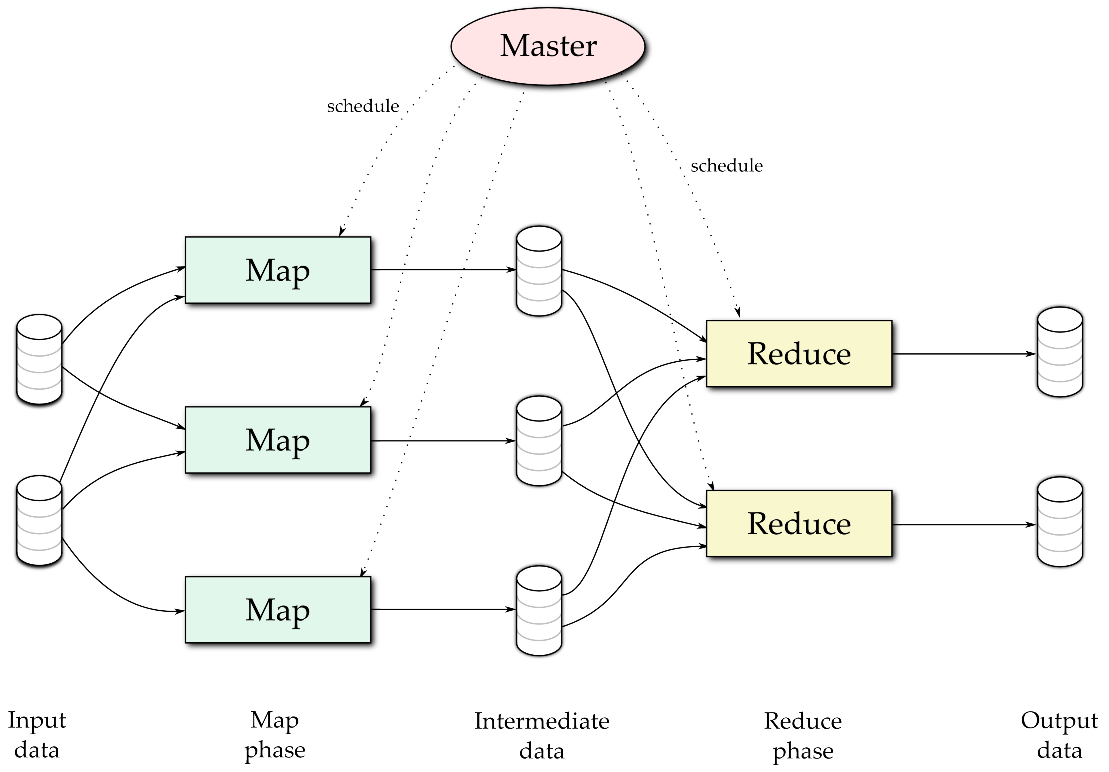


Figure 2.4: The map-reduce model [14]

Figure 2.4 illustrates the execution and flow of a map-reduce job. After submitting a job – such as the word counting program described above – the first step in a map-reduce process is to split the input data into suitably sized chunks that can be used as input for each map worker. For each input record in the chunk, the worker calls the map function, which emits an output record in the form of an intermediate key-value pair. These key-value pairs are then distributed to reduce workers based on their keys. All intermediate records that share the same key, regardless of which map process they originate from, are sent to the same reduce worker, and ultimately the same `reduce` function call.

### 2.2.2 Implementation Details

The map-reduce model only describes the general structure of a distributed program. How a map-reduce job is executed depends on the framework implementation. These aspects vary between systems.

Depending on what the desired error resiliency and performance is, a number of design choices can be made. One issue is whether to store map or reduce outputs to disk before passing them down the execution chain. Depending both on how I/O intensive a job is, as well as the hardware the job is running on, a better choice may be to use a streaming-based approach, where intermediate data is rarely written to and read from disk.

#### Out-of-Band Data

Out-of-band data is data that is emitted from the map and reduce functions through an auxiliary channel, and is not part of the regular output data. This data is passed back to the programmer in some manner and can for instance be made available in the subsequent phases.

One use of out-of-band data is to collect statistics about the processed data. Having information about the distribution of values is essential when performing a bucket sort, for instance. Bucket sorting is a common approach to sorting in the map-reduce model.

#### Intermediate Disk Storage

After data has been read from disk and processed by the map function, one must choose whether or not the intermediate key-value pairs should be stored to disk. The same question arises for every map or reduce phase involved.

Storing intermediate data to disk ensures that the size of the map output is not limited by the amount of memory in the cluster. Instead it is limited by the amount of available disk space, which may be many orders of magnitude more than the amount of available memory.

Also, the more frequently processed data is stored to disk, the more quickly the system can recover from node failure where the contents of memory, but not disk, are lost. If using a distributed, replicated file system, one may not even have to wait for the failed node to return. At the same time, it is obvious that an additional overhead is associated with storing all data to disk before or after it is transferred to the correct node for further processing.

If intermediate records are not stored to disk, there are two options:

- Keep map output data in main memory until the reduce phase begins. While theoretically possible, this will likely not be possible in the general case because of the constraints in memory size.
- Execute the map and reduce phases simultaneously, passing intermediate records directly from the map function to the reduce function. We discuss the implications of this in the upcoming section on reducer execution.

### Incremental Processing

The concept of incremental processing in this context is similar to storing intermediate records to disk. Instead of storing intermediate map output records, however, we store the states of reducer processes, similar to checkpoints in file systems and database management systems.

Such reducer states would consist of two primary parts: the current reducer data and a set describing which parts of the original input data set are included in the reducer data. If a reducer node fails, the framework may then be able to resume processing from the point where the reducer last saved its state, without reading data that has already been processed again.

### Serial or Parallel Reducer Execution

At this point we are down to rather specific details, but seemingly small details can often have a big effect on the performance and scalability of a system. One such detail is how entries are read and processed by the reducer processes. This is also related to whether intermediate records are stored to disk or not, as will be explained shortly.

As described in Section 2.2.1, the programmer expresses the reduce operation as a function that takes an intermediate key and the associated values as

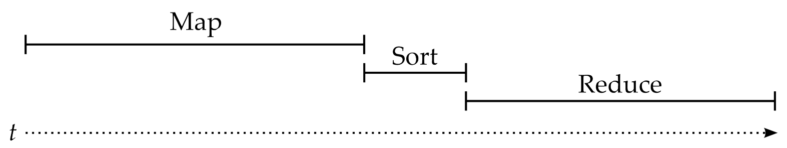


Figure 2.5: Serial execution of phases

parameters. This helps in keeping the complexity down, but there are some issues that should be kept in mind. This approach means that there is one reduce function call per intermediate key. It is largely up to the framework and the implementation language in which fashion these function calls should be executed. We can divide the possible execution approaches into two general strategies: parallel and serial.

Serial execution – illustrated in Figure 2.5 – is the most commonly used approach, being the method described in most papers and most often used by general frameworks. For each reduce worker, the intermediate data is sorted and grouped by keys before being passed to the reduce function. All intermediate records must be collected before sorting can begin, meaning that the map phase has to complete before the reduce phase can begin (as discussed in Section 2.2.2 on page 11).

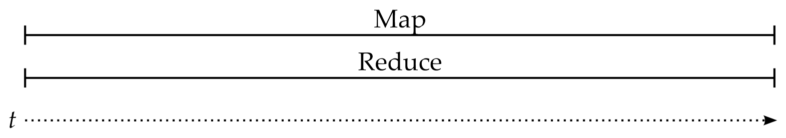


Figure 2.6: Concurrent execution of phases

In the case of parallel execution – as shown in Figure 2.6 – the reduce worker may execute multiple invocations of the reduce function simultaneously, for instance by using threads or a more specialised solution. Another, possibly more efficient approach is to change the structure of the reduce function to make it accept key-value pairs in a similar fashion to the map function. No parallel execution would be required, and the programmer could perform per-key calculations or aggregation in any desired way.

The clear advantage of a parallel or semi-parallel approach is that there would be no need to sort the intermediate data in advance. Instead a single pass can be made through the data, and records passed to the reduce function as they are encountered. This can work well in cases where the key space is small compared to the number of reduce nodes.

Both approaches have their disadvantages. The parallel solution demonstrates a larger demand for memory, as each reduce worker must maintain a reduce state for all intermediate keys assigned to it. It also removes the possibility of passing out-of-band results from mappers to reducers (however reducer to reducer is still possible, in the case of chained map-reduce jobs). The serial approach is not as memory intensive, but is potentially very I/O intensive as intermediate records must visit disk. Serial execution can also be slower than the parallel approach if the available compute resources allow efficient parallel execution.

### Choice of Programming Language

The choice of an implementation language for a map-reduce framework or program is very much a matter of speed versus implementation. With the introduction of frameworks for distributed processing, a general tendency is to choose implementation languages that allow quick implementation. Rather than spending more man hours on producing optimised program code, it can be less expensive to simply add more hardware to achieve the same performance gain.

# Chapter 3

# Methodology

This chapter describes the methodology we use to evaluate possible solutions to the problem at hand.

## 3.1 Overview

Our goal is to determine the viability of using map-reduce frameworks for flow processing. To do this, we start by designing a baseline reference system called Pagneda. Pagneda is designed to be as fast as possible. We also design a solution that is based on the supporting features of map-reduce frameworks. By doing this we can compare the framework based system against the reference system with regard to performance and scalability.

Having finished the two designs, they should both be implemented. We need to first determine what framework to use for the framework based design. Then we can implement both designs, followed by benchmarks to evaluate them. We should test how well both systems scale and what their throughput is, and then compare them to see how well the framework based design performs compared to what we consider an optimal solution.

## 3.2 Design

In Chapters 1 and 2 we have introduced a set of requirements for a distributed flow processing tool. Based on these requirements we must design and implement two architectures:

1. A baseline design that describes an optimised, stand-alone solution to the problem. This design should not be restricted to the functionality of any

map-reduce framework, but it should still abide by the basic principles of the map-reduce model. As such this is a *reference design* that represents a *best case* in terms of performance. It should provide a standard against which we can measure and evaluate a design that is based on the foundations of more general map-reduce frameworks.

Implementing a fully functional version of this design can take months. It is not the intention to turn this design into a full-fledged implementation, however. Instead, we must choose parts of this design that can be implemented within our very limited time frame, but still provide a proper reference point for a performance comparison of our two designs.

Accompanying the optimised design should be a theoretical estimation of what performance can be expected.

2. A design that makes use of the supporting features provided by map-reduce frameworks, and thus provides a tradeoff between performance and implementation complexity. This includes defining the map and reduce functions as per the requirements that are common among frameworks.

   An implementation of the framework-oriented design should take require little effort (in the range of days or weeks) to implement.

## 3.3 Framework Evaluation

In order to implement the framework-based design mentioned in Section 3.2, we first need to determine exactly which existing framework we should base our implementation on. Based on our requirements – both with regard to performance, as well as of functional and practical nature – we should determine by what criteria the frameworks should be evaluated. We must then identify which frameworks are available and suitable for our needs, and proceed to study and test these frameworks based on the criteria we have defined. Finally, a decision must be made as to which framework we will base our implementation on.

## 3.4 Implementation

We proceed by implementing both designs. For the framework solution we use whatever language and methodology that the framework dictates. The optimised solution, however, is implemented in C for the sake of performance. Only basic functionality is added, but enough to evaluate it's performance and create a point of reference for evaluating the efficiency of the map-reduce framework.

## 3.5 Benchmarks

The next step is to benchmark both implementations. The goal is to compare the performance of the framework implementation to that of the optimal solution.

The benchmarks are based on the requirements described in Section 1.2, and measure how well the implementations scale as more data and nodes are added, as well as what query response times can be achieved. The benchmarks are as follows:

- Minimum response time
  Execute a query that takes no or a minimal amount of data as input and measure the time taken until a response is returned.
- Node/data scalability
  Repeatedly execute a query on an increasing number of nodes, with each node processing the same, fixed amount of data.
- Node scalability
  Repeatedly execute a query on an increasing number of nodes, distributing a fixed total amount of data uniformly across all nodes.
- Data scalability
  Repeatedly execute a query on a fixed number of nodes while varying the amount of data processed by each node.

## 3.6 Analysis and Conclusion

Having finished the benchmarks we can study the results. We identify the areas where the systems perform better or worse than expected, and attempt to find the reasons for this.

Based on our analysis we should be able to make a number of educated conclusions about the viability of map-reduce – both frameworks and the model in general – for flow data processing.

# Chapter 4

# Design

This chapter describes two possible ways the map-reduce model can be employed. In Section 4.1, we start by listing some assumptions that our designs are based on. This is followed up by a description of the common data format we will use for our implementations and benchmarks in Section 4.2. In Section 4.3 we describe a baseline design, which takes a basic but efficient approach to the processing challenge. Then, in Section 4.4, we present a design based on general map-reduce frameworks.

## 4.1 Assumptions

This section describes some assumptions we make about the input data and user queries. These assumptions are based on the required operations, as well as what operations and data formats existing tools use.

### 4.1.1 Input Data

Our designs are quite flexible with regard to how data is stored, but we generally assume that flow data is stored as plain files in a file system.

To formalise this, all the flow data stored on a node or on a storage device – or any subset of the flow data that is read by a single entity (such as a map process) – can be seen as block of $N$ flow records. These files can be seen as $m$ chunks of arbitrary size, and $F_{chunks} = \{f_1, ..., f_m\}$ would be a set describing the size of each chunk.

In Section 4.2 we describe a simple data format that is very similar to the simple formats used by tools such as Nfdump and SiLK, and that is compatible with our assumptions.

### 4.1.2 User Input

As previously explained, the user will typically express the wanted computation as a query consisting of one of more of the following:

- A filter expression defining what flow records should be included in the processing.
- A set $\mathcal{Q} = \{Q_1, ..., Q_m\}$ of aggregation queries, each defining a set of flow record fields to aggregate on.
- A sort condition to indicate how the end result should be sorted, such as sorting by the total number of bytes.
- A limit statement to restrict the number of returned rows, typically to only the most interesting ones.

## 4.2 Data Format

Both designs are based on a common, binary data format. The format is intentionally simple. This keeps the overhead caused by parsing to a minimum, and allows for a more rapid implementation.

Table 4.1 shows the simplified flow format. It is very similar to the one used in NetFlow v7, as it has the same record length. It differs in that we store more precise time stamps, and that we omit some routing information (the IP address of the next hop and the prefix lengths of the source and destination networks). Offsets and field lengths are provided in bytes, and we can see that the total length for a single flow record is a constant 52 bytes. This format is all binary and can easily and efficiently be read and parsed by a C program.

It is worth noting that this format does not accommodate for IPv6 records, similar to how NetFlow versions that predate NetFlow v9 do not support IPv6. The format used in a production system should be extended to support IPv6, thus adding at least $2 \times 12 = 24$ bytes to each record. This brings the record length up to 76 bytes.

Another aspect worth noting is that at offset 19, a padding byte is inserted. This ensures memory alignment [18], so that accessing a single 4-byte field in the structure will only require reading a single 4-byte word from memory. Put shortly, an extra byte is wasted for a considerable increase in field access speed. Many C compilers (such as GNU) will perform this optimisation automatically [19]. By adding this padding ourselves, however, we achieve the same optimisation, but can fully predict the offsets of all elements in the structure.

| **Offset (B)** | **Length (B)** | **Field** |
|---|---|---|
| 0 | 4 | Start time (UNIX timestamp) |
| 4 | 4 | Start time (microseconds) |
| 8 | 4 | End time (UNIX timestamp) |
| 12 | 4 | End time (microseconds) |
| 16 | 1 | IP protocol |
| 17 | 1 | IP type of service (ToS) |
| 18 | 1 | Flags |
| 19 | 1 | *(padding)* |
| 20 | 4 | Source IPv4 address |
| 24 | 4 | Destination IPv4 address |
| 28 | 2 | Source port |
| 30 | 2 | Destination port |
| 32 | 4 | Source AS number |
| 36 | 4 | Destination AS number |
| 40 | 2 | Inbound interface number |
| 42 | 2 | Outbound interface number |
| 44 | 4 | Number of packets |
| 48 | 4 | Number of bytes |

Table 4.1: Simple flow format

### 4.2.1 Compression

If the available CPU resources allow it, compressing flow data may be advantageous. This is particularly in two respects: saving hard drive space and increasing the read performance. The disadvantage is the extra load decompression puts on the CPU and memory.

There are a number of compression algorithms that are near ubiquitous on UNIX systems today.

- The very commonly available *GNU Gzip* program [20] implements a variation of the LZ77 compression scheme called DEFLATE, known for being a fast, general-purpose algorithm.

- The *bzip2* program and algorithm [21] is also in common use. It demonstrates better compression ratios than gzip, but decompression performance is generally considerably worse.

- The *LZMA* algorithm [22] provides even better compression ratios than bzip2. Compression is slower and more memory intensive than bzip2, but decompression is more efficient.

In this section we shed some light on the efficiency of compression of flow data in the previously described data format. We do this by testing the performance of compression and decompression, as well as studying what compression ratios we can achieve.

### Performance Gain

Data compression can provide a gain in processing speed given one simple condition: decompression must be faster than the read speed of the disk. That is, the output data rate of the decompression program must be greater than the input rate when the input rate is equal to the disk read speed.

| | Compression (MB/s) | | | Decompression (MB/s) | | |
|---|---|---|---|---|---|---|
| | `gzip` | `bzip2` | `lzma` | `gzip` | `bzip2` | `lzma` |
| Fast (`-1`) | 31.61 | 6.14 | 8.42 | 87.44 | 21.81 | 34.27 |
| Medium (`-5`) | 15.75 | 5.31 | 1.07 | 94.27 | 15.45 | 37.50 |
| Best (`-9`) | 1.95 | 5.02 | 0.78 | 97.79 | 12.49 | 42.54 |

Table 4.2: Compression and decompression performance

Table 4.2 shows the performance of the Gzip, bzip2 and LZMA algorithms, in megabytes per second, when input data is stored in RAM. We use actual flow data in the previously described format as benchmark data. The compression values show how quickly the original file is read when using the given compression level (fast, medium or best). For decompression we list the speeds at which uncompressed data is output from the program.

From this table we observe that only Gzip is able to decompress data at a speed which somewhat matches or exceeds the performance of modern, mechanical hard drives. LZMA demonstrates a speed that is less than half that of Gzip, and bzip2 performs even worse.

It is also important to note the compression performance. The performance of the compression algorithm must at least match the rate at which data is collected.

### Compression Ratio

A lower compression ratio means that less data has to be read from disk. If decompression and processing is able to keep up, such as by performing them in parallel, this may ultimately lead to a $\frac{1}{r}$ times increase in processing speed, $r$ being the compression ratio.

Table 4.3 shows the compression ratios that the different algorithms can demonstrate, again using flow data for our tests. All the Gzip variants give

| | `gzip` | `bzip2` | `lzma` |
|---|---|---|---|
| Fast (`-1`) | 0.39 | 0.32 | 0.28 |
| Medium (`-5`) | 0.35 | 0.30 | 0.24 |
| Best (`-9`) | 0.33 | 0.29 | 0.19 |

Table 4.3: Compression ratios

less compression than even the fastest bzip2 variant, and all variants of bzip2 compress worse than any variant of LZMA.

## 4.3 Baseline Architecture

This section describes an architecture that is not constrained by the use of a map-reduce framework, but rather acts as a stand-alone application. The structure of this architecture is still based on the fundamentals of the map-reduce model, however, and maintains the same line of thought and simplicity. Other than a passing data down a directed acyclic graph, there is no inter-process communication or synchronisation between processes.

The architecture presented in this section functions as a baseline architecture. This means that we consider this a reference design, against which we will measure and evaluate the performance of the framework-based design presented in Section 4.4.

### 4.3.1 Structure

Figure 4.1 describes the general architecture of our baseline design. The general flow of data is as follows:

1. *Filter processes* (green boxes) on storage nodes read, filter and locally aggregate data from storage devices.

2. Using a list of all *aggregation processes* (yellow boxes), all filter processes map aggregates with the same aggregation key to the same aggregation processes.

3. Aggregation processes wait for and receive data from filter processes. Received aggregates that share the same key are merged.

4. Filter processes signal that reading has finished and that all records have been distributed.

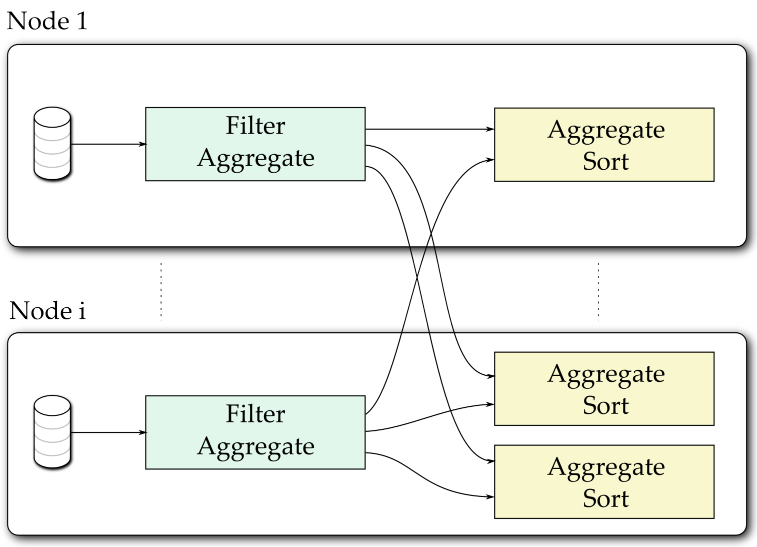


Figure 4.1: Baseline architecture

5. Each aggregation process sorts the local aggregates according to any sort field(s) supplied by the user, and limits the results if a limit has been specified.

6. Aggregation processes signal completion and the results are made available for reading by a merge process, like the user or storage agent.

In the following sections we describe the filter and aggregation processes in more detail.

### Filtering and Local Aggregation

Filtering is the initial step in the processing chain. One or more filter processes on each node reads flow records in chunks from a storage device locally on the node on which it runs. For each record that is read, any user-supplied filter expression is evaluated. If the expression evaluates to true, the record is aggregated locally in the process, based on the user's aggregation query or queries.

As mentioned, the input data is split into chunks. Given that there are $m$ chunks, $F_{chunks} = \{f_1, ..., f_m\}$ is a set describing the number of flows in each chunk $f_i$. Records are filtered and aggregated for each chunk individually. While the effectiveness depends on the size of input chunks, local aggregation exploits the fact that even in a small chunk of flow data, many flows will share aggregation keys. By performing this local aggregation, we reduce the amount of data each filter process needs to send to the aggregation processes.

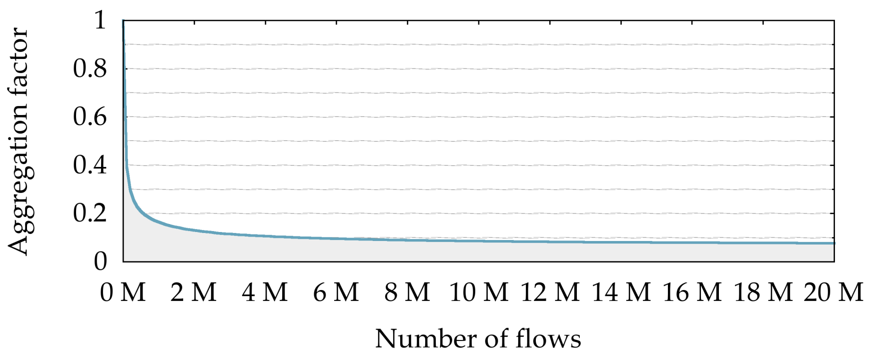


Figure 4.2: Aggregation factor for destination IP addresses

We performed a practical study of the effect of aggregating flows on destination IP addresses, and show the results in Figure 4.2. We have found the destination IP address to be the least commonly shared field between flows, and so it represents a "worst case" in single-field aggregation queries. By varying the number of time-continuous input flows from a single one to 20 million, we found that the ratio $C$ of flows with distinct addresses is subject to exponential decay. We call this ratio the *aggregation factor*.

The aggregation factor will, of course, vary between aggregation queries. There will also be a slight variation between chunks. Generally, with $m$ being the number of input chunks on a node, and $n$ being the number of aggregation queries, we can express the aggregation factors as $C = \{c_{1,1}, ..., c_{m,n}\}$. By using the aggregation factor to our advantage, we can write the number of emitted intermediate records as $\sum_{i=0}^{m}\sum_{j=0}^{n} f_i \times c_{i,j}$. It is clear that small values for $c_{i,j}$ would be of great advantage.

From the graph in Figure 4.2 we see that we can get a considerable decrease in the number of map output records even with small chunk sizes. By 2 million flows, which corresponds to roughly 100 MB of data, most of the "work" has already been done. We have only accumulated about 260,000 aggregates, producing a ratio of about 0.13. We also continue to observe a slight decline in the aggregate-flow ratio even beyond 20 million flows (about 1 GB of data), with the ratio at least dropping to 0.077 in our tests.

It is worth noting that the same aggregate-flow ratio may not hold equally well for IPv6 traffic, due to the extended address space and such features as IPv6 Privacy Extensions [23].

The disadvantages of this approach are only slight. A few issues stand out:

- Many mappers will store different aggregates with the same key, leading to less efficient utilisation of memory in the cluster as a whole.

- Aggregates are sent from the mappers to the reducers in bursts, potentially making the network connection between the nodes a bottleneck.
- Further processing of the aggregates is delayed for the time it takes a mapper to process an entire chunk and start sending the resulting aggregates to the reducer.

All of the aforementioned issues can somewhat be mitigated by adjusting the chunk size. A smaller chunk size will cause less memory to be used by the mapper, and the processing delays will be reduced. Traffic bursts will also be slightly smaller, but there will be correspondingly more of them. A larger chunk size, however, will reduce the total number of aggregates passed from the mapper to the reducers, ultimately also reducing the total amount of data transferred from the map processes to the reduce processes.

#### Global Aggregation and Sorting

As the filter processes finish processing chunks, the results are sent to aggregation processes. The aggregation process simply merges all incoming aggregates, maintaining a hash table for near-constant time lookup of existing aggregates.

As soon as all aggregates have been transferred from the filter workers to the aggregators, the aggregation workers can sort and limit the resulting aggregates if the user has requested it.

#### Merging and Completion

If sorting was requested by the user, it is also necessary to properly merge the results from all aggregation workers. As all the workers have already sorted their individual results, it is trivial to merge them into a complete, sorted result set. This is done by simultaneously reading the sorted results from all workers, always retrieving the largest or smallest value depending on the sort order.

## 4.4 Framework Based Architecture

This section presents an approach to NetFlow processing in the map-reduce model, adapted to fit the typical abstraction provided by map-reduce frameworks such as Hadoop and Disco. We also discuss some problems with the basic approach, and introduce some refinements to alleviate these issues.

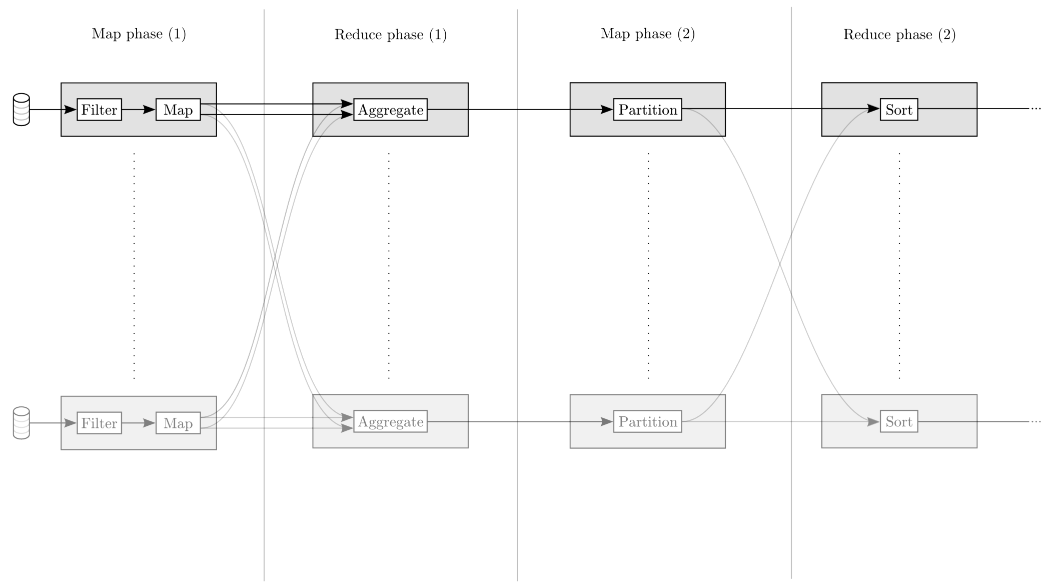


Figure 4.3: Basic map-reduce chain

Figure 4.3 gives an overview of a basic map-reduce chain that can provide filtering, aggregation and sorting of records. The first and leftmost map-reduce iteration is concerned with filtering and aggregation. The second and rightmost iteration provides sorting of the aggregates.

In order to minimise I/O and network traffic, filtering should be done as early in the execution chain as possible. It is therefore placed in the initial mapper. The mapper reads records directly from the assigned input chunk. For each record, the filter expression is evaluated, and the record is immediately skipped if the expression evaluates to false.

After verifying that the record should be included, the mapper extracts the aggregation key fields from the record based on the aggregation queries. For each aggregation query $Q_i$, an intermediate key-value pair is emitted. The key part is the aggregation key, and the value part is whatever information from the record that is needed to calculate the aggregate (such as the number of packets and total number of bytes in the flow). In some cases, this simple filtering may be all the user wants, so at this point the result set may already be returned to the user without passing through subsequent reducers.

If the user has requested aggregation as part of the query, the intermediate key-value pairs are then passed on to reducers. All key-value pairs which share the same key are sent to the same reducer. Upon reception, the reducer aggregates records based on the key.

The greatest challenge lies in sorting the final result in an efficient manner.

The general map-reduce model does not accommodate well for sorting, unless you have advance knowledge of the distribution of values you are sorting on. A general method which performs fairly well if the distribution is known – or can at least be quite accurately estimated – is bucket sorting.

A bucket sort involves assigning non-overlapping value ranges to a number of buckets (typically one bucket per node or CPU). Records are then distributed to these buckets based on the value ranges, and each bucket is sorted individually. Due to there being no overlap in the value ranges of the buckets, the final result can be obtained simply by concatenating all buckets.

The bucket sort can be implemented in the map-reduce model by executing a new map-reduce iteration on the data that is to be sorted. The output from the first reduce phase is input into a second map. The map assigns aggregate records to buckets by emitting key-value pairs where the key identifies the bucket and the value represents the aggregate record. The reducers then receive the buckets, sort them each individually, and finally emit the sorted list of records.

### 4.4.1 Optimised Execution

The basic approach described in the previous section has a number of problems.

- Given $F$ input flows and a set $\mathcal{Q} = \{Q_1, ..., Q_i\}$ of aggregation queries, the mapper function will emit $F \times i$ intermediate records. This may lead to consumption of more network bandwidth than is desirable, as described in Section 4.3.1.

- The sorting method is very inefficient, as it involves redistributing aggregates based on the sort key. In the worst case, all aggregates must be transferred to another node.

Given a bit of flexibility in the map-reduce framework, these two issues can be resolved by subjecting the input and output data to pre- and post-processing, respectively.

#### Pre-Processing

Pre-processing will in practice do the same thing as the filter process in the baseline architecture. Records are read from storage, filtered and then locally aggregated. The local aggregates are then used as input for the map-reduce job, mapping aggregates to reduce workers based on their keys, and redistributing them. Thus, instead of passing $F$ intermediate records down the processing chain, the map worker emits the same $\sum_{i=0}^{m}\sum_{j=0}^{n} f_i \times c_{i,j}$ records as the baseline filter process does, $f_i$ again being individual chunk sizes and $c_{i,j}$ being aggregation factors for all chunks and aggregation queries.

### Post-Processing

The introduction of a post-processing phase is meant as a replacement of the initially suggested sorting method where aggregate records are redistributed into buckets and sorted with an additional map-reduce iteration. Instead we allow the aggregation reducers to sort their individual results before passing them down the chain. When the results have been sorted, they can be on to a post-processor that merges the final results.

# Chapter 5

# Map-Reduce Frameworks

This chapter introduces and compares a number of existing map-reduce frameworks. We begin by describing the evaluation criteria in Section 5.1. Section 5.2 provides an evaluation of Apache Hadoop [16] and introduces some useful features of that framework. In Section 5.3 we consider Disco [15], a more lightweight but less feature-rich alternative to Hadoop. We make a conclusion in Section 5.4, deciding which framework we will use for our benchmark implementation.

## 5.1 Evaluation Criteria

The job of the framework is to accept jobs of some format, and schedule these jobs to run on a cluster of nodes. When the job finishes, the result should be returned to the caller. The framework may also be involved in tasks such as formatting the input and output data and managing an underlying storage platform.

The implementations differ in many regards, ranging from programming language to job format and data format. In order to determine which map-reduce framework is likely to be the one most suitable for the job at hand, we evaluate the frameworks based on a number of criteria.

**Processing speed** How efficient is the pipeline that records pass through?

**Framework overhead** Compared to executing a processing job on a single computer, executing it on a cluster of machines is likely to add some overhead.

**Feature completeness** Rapid implementation of distributed jobs is at the core of the map-reduce model. What features does the framework provide that can relieve the programmer from writing too much code?

**Ease of use** Is the documentation of good quality? How difficult is it to develop and run jobs? While it is difficult to quantitatively measure documentation quality or ease of use, we provide a general assessment.

## 5.2 Apache Hadoop

Apache Hadoop is a Java-based framework with very extensive functionality. Hadoop does not only provide a map-reduce implementation, but an entire software suite for performing distributed computations. The following section covers Hadoop with regard to features, ease of use and performance.

### 5.2.1 Features

As Hadoop is a large and complex system, an exhausting list of features would be too long to present here. However, we can highlight the functionality which is of relevance to us.

#### Hadoop Distributed File System

HDFS provides reliable storage of both input and output data for map-reduce jobs. It automatically distributes data across any number of nodes, optionally also adding replication. Data is represented as files, but files can be split into blocks that are scattered across nodes. Map-reduce programs then process the data block-wise, and given that the data format allows it, many nodes can operate on the same file simultaneously.

#### Pig and Hive

Pig and Hive are frameworks that provide a non-programmatic interface to some common operations on large data sets. Instead they provide interfaces that expose the data sets in a standardised way. They accept queries from the user, which are compiled into Hadoop jobs and executed on a cluster.

All Pig queries form a directed acyclic graph through which data flows. Each node in this graph represents some operation on the data set. Pig allows the user to construct these graphs in an incremental fashion through a fairly simple query language called *Pig Latin*.

```
dump = LOAD <input> USING FlowStorage() AS (start_time, end_time,  ...);
project = FOREACH dump GENERATE src_addr, packets, bytes;
ag = GROUP project ALL PARALLEL 2;
```

```
s = FOREACH ag GENERATE group, SUM(project.packets), SUM(project.bytes);
DUMP s;
```

Listing 5.1: Example Pig Latin script

Listing 5.1 demonstrates how a query can be constructed using Pig Latin. This query groups flow records by source IP address and calculates a sum of the number of packets and bytes for each flow. Pig also supports filtering and sorting of data, as well as limiting the result set to a specified number of records.

Hive provides a query language, *HiveQL*, that is similar to SQL and more general than the Pig query language. All data sets processed by Hive are exposed as tables. From these tables one can select, insert, perform joins and other operations often associated with relational algebra. Listing 5.2 shows an example Hive query that performs a similar query as the Pig Latin example in Listing 5.1.

```
SELECT flows.src_addr, SUM(flows.packets), SUM(flows.bytes)
FROM flows GROUP BY flows.src_addr;
```

Listing 5.2: Example HiveQL select query with aggregation

Similarly to Pig and SQL, Hive supports grouping of records and calculating sums and record counts for the aggregated data, a feature we have utilised in the example above. The listing above also illustrates the similarity between SQL and HiveQL, and shows that HiveQL can be considerably less verbose than Pig Latin. Although it is not shown in the example, Hive also allows filtering, sorting and limiting of the result set.

### 5.2.2 Usage and Documentation

As previously mentioned, Hadoop is a complex system. Setting up a Hadoop cluster can be a daunting task, as there is a large number of configuration options and settings to adjust. Writing jobs, and efficient ones at that, is also likely to require the programmer to familiarise themselves with a very extensive API.

A strong point of Hadoop, however, is the very comprehensive documentation that is available online. Full Java API documentation is available, as well as a range of tutorials, papers and articles that cover many of the possible uses of Hadoop.

### 5.2.3 Performance

We have previously studied and evaluated the performance of Hadoop, as well as the integrated Pig framework [1]. A number of relevant queries were adapted to the Pig framework and executed on an 8 CPU cluster.

Our findings with regard to the performance of Hadoop include:

- The minimum turnaround time for Hadoop jobs is more than 20 seconds. The Pig framework adds an additional overhead of roughly 20 seconds, meaning a submitted job will take a minimum of 40 seconds to complete, regardless of the amount of data that is to be processed.

- All intermediate data goes to files. Hadoop does not support streaming of data between nodes. While this adds error resiliency, it means that throughput is limited to whatever the file system can accommodate for.

- The Pig framework does not do a very good job at performing query optimisation, leading to an unnecessarily high number of Hadoop jobs being executed for some queries.

## 5.3 Disco

Disco was originally developed by Nokia Research. It has been developed using a combination of Erlang and Python. Erlang is at the core, and is typically not exposed to the user, while the development API is written in Python. When compared to Hadoop, it stands out as a more lightweight but less feature rich framework.

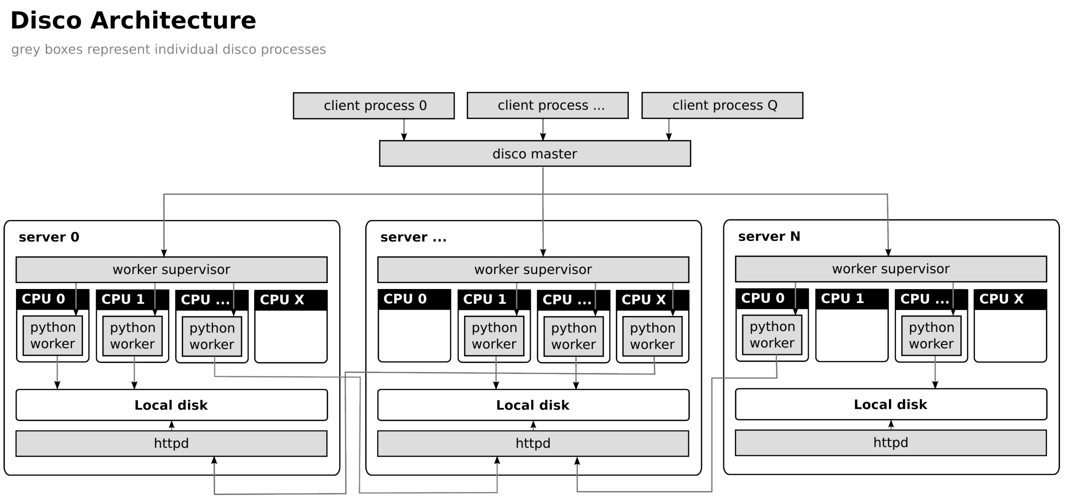


Figure 5.1: Disco architecture [24]

Figure 5.1 shows the general architecture of a Disco cluster.

### 5.3.1 Features

Apart from the minimal set of features required to write and run map-reduce jobs, the Disco framework does not provide a big selection of features.

The main map-reduce interface is exposed as a Python API. This means that writing a map-reduce job can be as simple as writing a Python script implementing two functions, and calling an API function with references to these functions.

Disco also allows external programs written in any language to function as map and reduce functions. CPU intensive tasks, for instance, can be written in C.

The Disco Project has also introduced the Disco Distributed File System (DDFS), a file system similar to Hadoop's HDFS, and Discodex, a key-value storage system. These systems are in their early infancy. At the time of writing they were only just published, giving us little opportunity to study them in detail.

### 5.3.2 Usage and Documentation

As jobs are typically written in Python using a very compact API, Disco provides an excellent environment for rapid implementation of processing tasks.

Documentation is scarce, however, and does not cover much more than the basics.

### 5.3.3 Performance

The performance profile of Disco is slightly different from that of Hadoop. The following are the results of our analysis:

- Disco demonstrates an excellent minimal turnaround time of less than one second, making it suitable for ad hoc queries.
- Similarly to Hadoop, Disco also stores all intermediate and output records to disk.
- The performance of the record pipeline is severely limited. Each individual record is read, processed and written by Python code. For each record there is a slight overhead, which causes Disco to perform poorly when there is a large number of small records to process.

## 5.4 Conclusion

Many of the features Hadoop provides can prove very useful in our problem context. For instance, data must be distributed among nodes. HDFS can assist in doing this, as well as add replication. On the other hand, the data format in question is one that requires little effort to split, distribute and index "manually". Also, if distribution is done manually, full control is kept as to how the data is distributed (such as splitting by time intervals, flow fields, etc.).

Hadoop's Hive and Pig tools both provide useful query languages that fit our needs quite well. We have previously found, however, that the Pig framework has performance issues that tend to cause prohibitively long query times. We have not tested the performance of Hive, but a published benchmark shows that it performs better than Pig in most cases, but in many situations also considerably slower than writing map-reduce functions directly [25].

While feature-wise an excellent platform, Hadoop has generally proven to be somewhat slow and heavy. It introduces a query time overhead that is very impractical for ad hoc queries.

Disco's low turnaround time for jobs, combined with the ability to embed external programs, for instance written in C, makes it an attractive alternative to Hadoop. We do not consider the lack of an integrated, distributed file system a big issue. As previously explained, it is trivial to manually partition and distribute the data. Since we are concentrating on UNIX-like environments, it is also possible to make use of any POSIX-compatible distributed file system, transparent to the map-reduce framework.

Disco has no equivalent of Hive or Pig, which means that a query language would have to be implemented from scratch. This would take considerable work, but a query language is not essential in testing the performance of the framework. Therefore this is something we choose to put aside for now.

In conclusion, Disco seems to be the most viable option for a performance evaluation. The low response time puts it quite far ahead of Hadoop with regard to ad hoc queries. Also, the supporting features of Hadoop (HDFS, Pig, Hive, etc.) are not critical for our needs, and may even get in the way of reliably evaluating the map-reduce model itself.

# Chapter 6

# Implementation

In this chapter we provide some details on two different implementations of a very basic processing tool. We begin by describing the internals of *Pagneda*, our implementation of the previously described baseline architecture, in Section 6.1. Then, in Section 6.2, we describe an implementation based on the Disco map-reduce framework.

## 6.1 Pagneda

Pagneda is a simple implementation of certain parts of the baseline architecture described in Section 4.3. It is implemented in pure C for the sake of performance. We have chosen to limit our implementation to filtering and global aggregation, and do not implement sorting or merging of results.

Pagneda has no error handling or resiliency, as it is meant to be run in a very limited and controlled environment.

### 6.1.1 Structure

Pagneda implements the baseline system very much as described in Section 4.3. It consists of two programs – `filter` and `aggregate` – which communicate via TCP streams.

The `filter` program accepts a list of aggregation fields as parameters, and reads input records from standard input. It maintains a hash table (via the Judy C library [26]) that is used for performing local aggregation. For each flow record that is read, the aggregation key is extracted and looked up in the hash table. If an existing aggregate record is found, the byte, packet and flow counters of that aggregate record are incremented based on the current flow record. If an aggregate is not found, a new one is allocated and initialised with the values from the current flow record.

When no more records are available on standard input, the `filter` program connects to all aggregation processes (instances of the `aggregate` program) via TCP/IP. It then iterates through all the local aggregates. Each aggregate key is hashed, and the modulo of the hash is used to determine which aggregation process the aggregate record should be sent to. The aggregate record is then sent to the correct process. Finally, after having transferred all aggregates, the `filter` program exits.

After launching, the `aggregate` program initialises a hash table and proceeds to wait for incoming connections from filter processes. Aggregate records are read from connecting processes and used to update a hash table of merged aggregate records, similarly to how the `filter` program operates.

### 6.1.2 Execution

Execution of an aggregation job begins by launching the `aggregate` program on a number of machines. It is possible to run more than one instance on a single machine to utilise multiple CPUs. After launching, the `aggregate` processes will listen on a given TCP port for incoming connections from `filter` processes.

When the aggregation processes are up and running, `filter` processes can be started. They typically run on each node that has data stored, and can also be executed in parallel to utilise multiple CPUs or disks.

Using the same aggregation logic as the `filter` program, the `aggregate` program will keep a hash table of aggregates, and “merge” any aggregates from different filters that have the same key.

## 6.2 Disco

This section describes how we have adapted the framework based design presented in Section 4.4 to be executed using the Disco framework. We presented Disco in Section 5.3.

### 6.2.1 Architecture

A machine in a Disco cluster can be either a master or a node. There is typically one Disco master in a cluster, possibly more for added redundancy. Tasks are submitted to the master, and the master keeps track of what nodes are present in the cluster and schedules jobs to run on available nodes.

Any computation that should be executed in Disco is expressed as a job. A job is submitted to the system as a Python script that describes the map and reduce functions, where to read input data from, in what format and so on.

#### Patches to Disco

Disco is still somewhat experimental software. New versions are not subject to extensive testing before being released, but they rather rely on the community to report bugs and defects. Bugs do indeed exist, some of which would prove to get in our way and needed fixing.

We created and applied a set of patches to version 0.2.4 of the Disco code [27] to fix many of the bugs we came across. None of our patches made any changes to the functionality of the system, but merely fixed programming errors that prevented normal operation. As such we will not go into any more detail here, but we rather refer to Appendix B for more details on what changes had to be made to make Disco work as expected.

### 6.2.2 Job Format

Disco has support for using external programs as map and reduce functions. It also allows the programmer to specify how data should be read from disk. We have attempted to employ this by reusing some of the C code written for Pagneda. By doing this we rest on Disco's ability to schedule and execute jobs across a cluster, transfer data and handle errors, but gain performance by processing the data in C rather than Python.

We replace Disco's default file reader – which simply treats each line in the input file as an input record – with a wrapper around the `filter` program from Pagneda. Thus the local aggregation is done efficiently, reducing the load on the framework.

The `filter` outputs key-value pairs that function as intermediate records. This makes the map function very plain and simple, but it must still be provided, in the form of a function that return the exact same record as was passed to it.

The reduce function is a bit more specialised. It uses the programming interface provided by Disco for writing external programs. We implement the function as a C program that uses the same aggregation code as Pagneda, but reads and outputs data in the format used by Disco's external interface.

### 6.2.3 Execution

The user executes a job by invoking the Python script. The script communicates with the master to schedule the job for execution, and waits for results to become available.

# Chapter 7

# Benchmarks

This chapter describes the benchmarks we have executed in order to evaluate the performance of the implemented systems.

In Section 7.1 we start with describing the details of the hardware and software environments in which we have executed the benchmarks. We proceed by presenting the benchmarks and results for Pagneda in Section 7.2 and Disco in Section 7.3. An analysis of the results and a conclusion follows in Section 7.4.

## 7.1 Benchmark Execution

In this section we describe the environment the benchmarks are executed in, as well as the procedures and software used to execute the benchmarks.

### 7.1.1 Environment

All benchmarks were executed on a cluster of 15 server-grade machines of varying make, age and specifications. Table 7.1 lists all the nodes along with their CPU and memory specifications.

All machines in the cluster were connected through gigabit Ethernet to a single switch.

### 7.1.2 Software

All the machines in the cluster were installed from scratch with the current stable version of Debian GNU/Linux, 5.0 (Lenny). We used Disco version 0.2.4, modified with patches as described in Section 6.2.1.

| Model | CPU speed (GHz) | Cores | RAM (GB) |
|---|---|---|---|
| IBM xSeries 306m (Pentium D) | 2.80 | 2 | 4 |
| IBM xSeries 306m (Pentium D) | 2.80 | 2 | 2 |
| IBM xSeries 306m (Pentium 4) | 3.20 | 2 | 2 |
| IBM xSeries 335 (Xeon) | 3.06 | 1 | 2 |
| IBM xSeries 335 (Xeon) | 3.06 | 1 | 1.5 |
| IBM xSeries 335 (Xeon) | 2.40 | 1 | 1.5 |
| IBM xSeries 335 (Xeon) | 2.40 | 2 | 2 |
| IBM xSeries 335 (Xeon) | 2.40 | 1 | 1 |
| IBM xSeries 335 (Xeon) | 2.40 | 2 | 3 |
| IBM xSeries 335 (Xeon) | 2.40 | 2 | 3 |
| IBM xSeries 335 (Xeon) | 3.06 | 2 | 4 |
| IBM xSeries 335 (Xeon) | 3.06 | 1 | 2 |
| IBM xSeries 335 (Xeon) | 3.06 | 1 | 2 |
| Sun Fire X2200 (Opteron) | 2.60 | 4 | 8 |
| Sun Fire X2200 (Opteron) | 2.60 | 4 | 8 |

Table 7.1: Cluster nodes

## 7.2 Pagneda

In order to get an idea of the baseline efficiency we have implemented a small subset of the desired features in C. This implementation is functionality-wise similar to the one used by Disco, but manually implements many of the functions needed for distributed processing.

### 7.2.1 Procedure

We executed the Pagneda benchmarks using a set of shell scripts. The scripts were executed on a master node, which did not function as a compute node and did not hold any data. All it did was schedule jobs to run on the other 14 nodes.

One benchmark iteration has two main steps: execution and cleanup. The cleanup step destroys the aggregation processes and deletes log files on the computation nodes. The execution step is where the "real work" is done, and is the only one that is measured. We use the GNU `time` program [28] to measure the wallclock time taken until execution finishes.

The execution script – shown in Listing C.1 in Appendix C – begins by launching an `aggregate` process on each compute node that waits for incoming connections. It then launches a `filter` process on each compute node. Using the job control features of the shell, the script waits for all `filter` processes to finish, and exits as soon as the last one does. The processing time is then logged to a file by the `time` program.

To the extent possible, all tests were run a minimum of five times. Any exceptions to this are explicitly noted. The presented results are based on the average of all successful runs.

### 7.2.2 Results

In this section we present the results of four Pagneda benchmarks. We evaluate the minimal response time of the system, as well as how well it scales when we add more nodes, data or a combination of the two.

#### Minimal Response Time

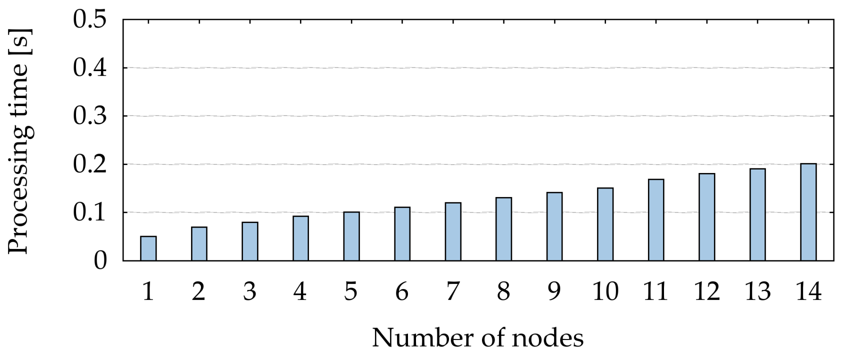


Figure 7.1: Minimal turnaround time with Pagneda

Figure 7.1 shows the minimal response time when we process a single flow record on each involved node. What this chart shows is basically the time a shell script on the master node takes to launch `aggregate` and "no-op" `filter` processes on all compute nodes via SSH and wait for all the `filter` processes to finish. With only one node involved, the response time is 50 ms. By 14 nodes, the response time was 201 ms.

While we do observe some slight variations in the response time, it is monotonically increasing, mostly proportional to the number of nodes. On average we observe a 10.8 ms increase in response time per additional node. This increase can mostly be attributed to the lack of concurrency in the benchmark scripts, as pointed out in Section 7.2.1.

### Node/Data Scalability

In order to examine the efficiency of Pagneda as more nodes are added to the cluster, we executed an aggregation query on an increasing number of nodes and distributed an identical 100 MB chunk of data to each node.

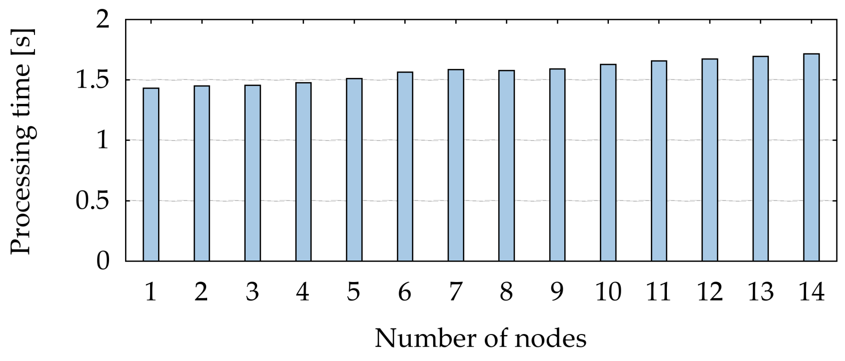


Figure 7.2: Pagneda processing times for 100 MB of data

Figure 7.2 shows the development in execution time for this benchmark. We observe a slight increase in run time as nodes are added. With a single node and 100 MB of data the run time is 1.43 seconds, and it is 1.71 seconds with 14 nodes and 1400 MB of data. This gives an average increase of 20 ms per added node. As seen in the previous section on minimal turnaround time, about half of this increase can be attributed to the benchmark scripts. The rest is likely due to a slight overhead added by the increased network traffic.

The execution time is not monotonically increasing, however. We see a slight drop in run time when going from 7 to 8 nodes, after seeing a slightly steeper increase from 4 to 6 nodes. We attribute this to the variations in the cluster node specifications.

### Node Scalability

In this benchmark we also vary the number of nodes from 1 to 14, but we run all tests on an aggregated total of 300 MB of data. In the case of one node there is a single 300 MB file. In the case of two nodes we have a 150 MB chunk on each node, and so on.

As can be seen from the chart in Figure 7.3, Pagneda displays an exponential decay in execution time when distributing 300 MB of data across nodes. The decrease in run time is slightly more dramatic for the first four nodes added. From 1 to 2 nodes we see a 43% decrease. From 2 to 4 nodes the reduction is as much as 53%. From 4 to 8 nodes the change is once again 43%. While

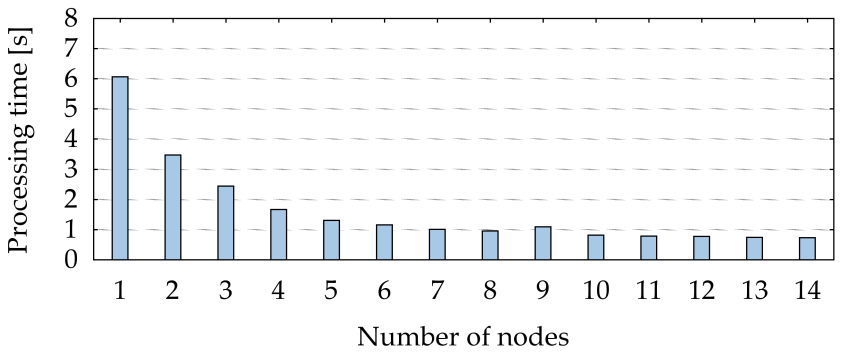


Figure 7.3: Pagneda processing times for a total of 300 MB of data

doubling the number of nodes does not reduce processing time by exactly 50%, the general trend is in that area. As with earlier tests we see slight variations, again likely due to hardware variations as well as a slight overhead added by additional nodes.

For seven nodes and upwards, the change gets smaller, with a 28% reduction from 7 to 14 nodes. The reduced effect of adding more nodes is partly due to the increased setup time. It is likely also because the time spent doing global aggregation (as described in Section 4.3) becomes greater relative to the time spent performing local aggregation. The aggregation factor of 14 small blocks is likely considerably higher than for one large block, resulting in a greater total number of aggregates across all nodes.

One thing worth noticing is the increase in run time at nine nodes, followed by a drop at ten nodes. Unfortunately it has proven difficult to pinpoint the exact reason for this.

### Data Scalability

In this benchmark we keep the number of nodes constant at 14. Instead we add data blocks in increments of 50 MB on each node. Based on this we run tests on a total of 700 MB to 14 GB of data.

Figure 7.4 shows how Pagneda performs with an increasing data volume. The development in execution time is quite clear and predictable, and the overall trend is linear scalability. Processing 250 MB of data per node takes 4.8 seconds, 500 MB takes 9.5 seconds, and 1000 MB per node takes 19.5 seconds.

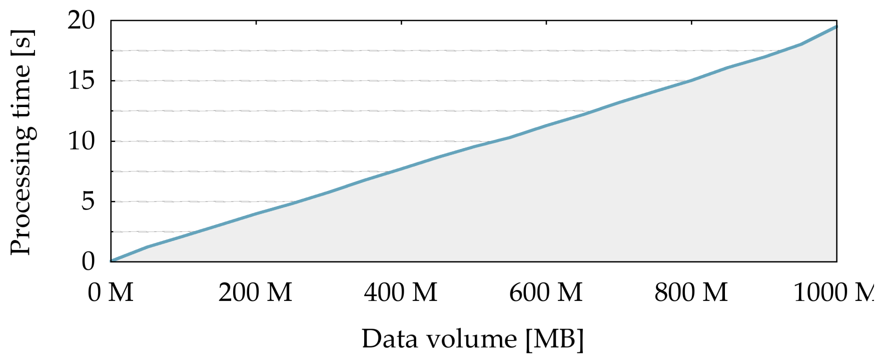


Figure 7.4: Pagneda processing times for 0–1000 MB of data on 14 nodes

## 7.3 The Disco Framework

We subjected Disco to the same benchmarks that we ran Pagneda through. Like in the previous section about Pagneda, we first introduce the procedure and measurement techniques, and then proceed to present the results of our tests.

### 7.3.1 Procedure

The Disco benchmarks perform the same operations as the Pagneda benchmarks. We also use the exact same input dataset, and put aside one machine to act as a master. The differences lie in how the jobs are executed. Instead of shell scripts, we use Python scripts that communicate with the Disco API. Instead of the workers communicating directly using simple TCP streams, Disco uses temporary files and HTTP to transfer data.

All Disco jobs are submitted via the master node, using a script like the one shown in Appendix C. We perform our benchmarks by repeatedly invoking the Python scripts that describe the jobs and measuring the wall clock time elapsed before the script finishes. Time measurement is once again done using the GNU `time` program.

### 7.3.2 Results

This section presents the results of executing the same benchmarks on Disco that we did for Pagneda.

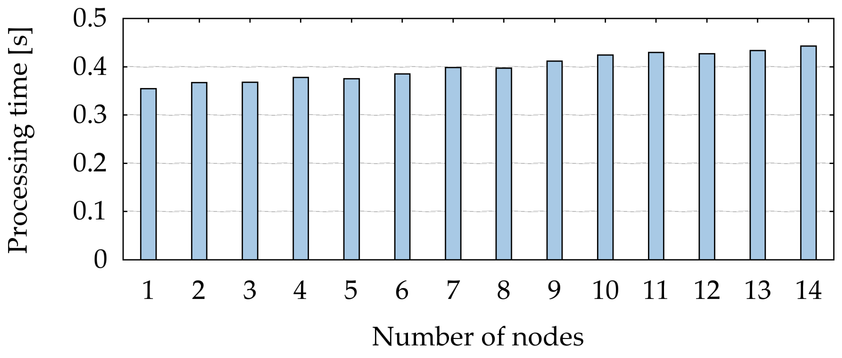


Figure 7.5: Minimal turnaround time with Disco

### Minimal Response Time

Figure 7.5 shows the response time of the processing script when the only input data on each node is a local file containing a single flow. This isolates the run time to be the overhead added when the cluster is expanded with more nodes. It also shows the impact scheduling and job control in Disco has on run time. The response times are averages calculated from 200 runs.

With one computation node, the response time is about 350 ms, which is about 7 times that of Pagneda. With the addition of more nodes we observe a slight increase in response time, although not monotonic. The lack of monotonicity is mainly caused by outliers in the benchmark results. Despite the lack of monotonicity, however, the general trend is an increase in response time of 6.3 milliseconds for each added node, which is 4.5 ms less than our Pagneda benchmark can demonstrate.

The lower increase in response time is likely due to Disco being more efficient at exploiting concurrency when scheduling remote workers. Pagneda's steeper increase is likely an artifact of executing remote commands in a sequential fashion. Based on our benchmark results we can estimate that with more than about 70 nodes in the cluster, we will most likely find Disco to be more efficient at scheduling than our Pagneda setup.

### Node/Data Scalability

Like with Pagneda, we put 100 MB of data on each node, and run an aggregation query with as many concurrent map and reduce processes as there were nodes.

Figure 7.6 shows the effect on wall clock processing time (average calculated from three runs) as more nodes are added to the processing. In essence this

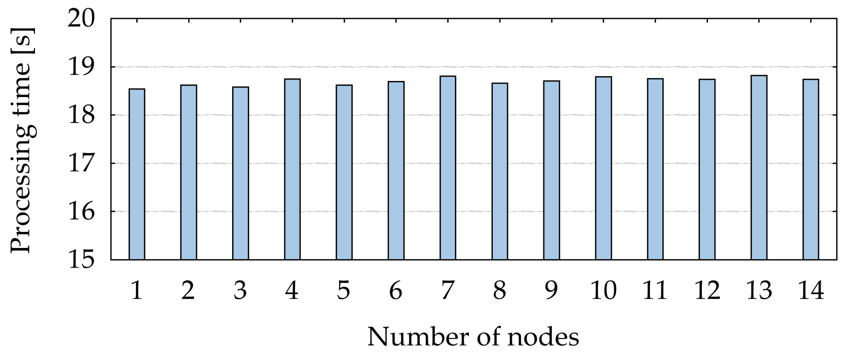


Figure 7.6: Disco processing times for 100 MB of data

reflects the time taken until the slowest reducer finishes.

It is difficult to extract any trends from this data, as the execution time fluctuates as nodes are added. We assume that the fluctuations can be explained by the diverse specifications of the cluster machines. Another factor that may have a certain effect on the final processing time is the quality of the hash function used by Disco to distribute intermediate records among reducers. This, however, has not been examined in detail.

Despite the fluctuations, the overall trend is that there is a slight increase in execution time from 1 to 14 nodes. The increase is only in the area of 0.2 seconds, however, which is slightly lower than Pagneda.

**Node Scalability**

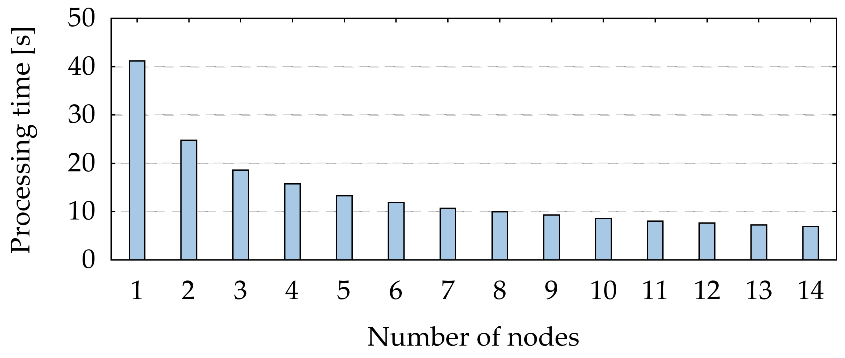


Figure 7.7: Disco processing times for a total of 300 MB of data

Figure 7.7 shows the development in run time as nodes are added. When using one node, the processing time is 41 seconds. Going up to 14 nodes, the processing time drops to a little less than 7 seconds.

We can see a clear exponential decay in execution time. Doubling the number of nodes does not mean run time is cut in half, however. Going from one to two nodes we see a 40% run time decrease. From two to four nodes the reduction is about 37%, and roughly 35% from 7 to 14. As such it is clear that the gain from adding more nodes decreases as the cluster gets larger and chunks get smaller. This can be due to many factors. As with Pagneda, smaller chunk sizes involves more data being transferred between nodes. Also, more time is spent doing "administrative" tasks between phases – like disk storage and network transfers – relative to the time spent processing data.

### Data Scalability

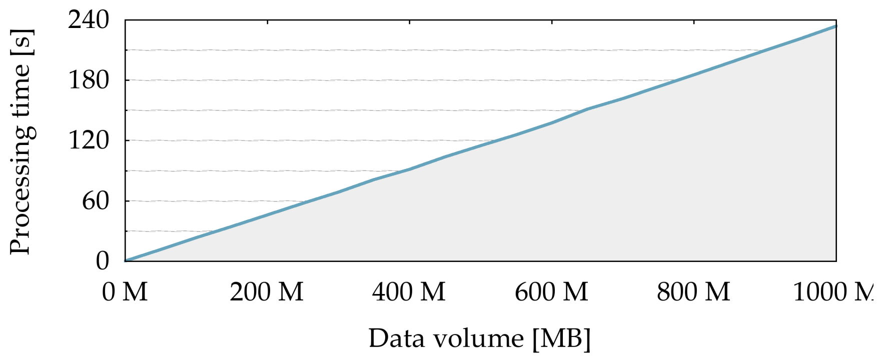


Figure 7.8: Disco processing times for 0–1000 MB of data on 14 nodes

Figure 7.8 shows that Disco scales linearly when the data volume is increased, with the run time being very near proportional to the total amount of data processed. Processing 250 MB per node takes 57.6 seconds. Processing 500 MB per node takes 115.2 seconds, and 1000 MB per node takes a total of 234.0 seconds.

This also clearly illustrates the throughput of the framework. 1000 MB in 234 seconds corresponds to 4.28 MB/s per node on average, or 59.8 MB/s across all 14 nodes. This is more than an order of magnitude slower than Pagneda's 718 MB/s across all nodes.

## 7.4 Analysis and Conclusion

Our benchmarks have shown areas where Disco and Pagneda perform both similarly and differently.

We start by considering the response times of the two systems. Disco has shown to have a response time in the area of 350 ms and upwards, which is quite acceptable for ad hoc queries. The addition of about 6 ms for each involved computation node also bodes well for Disco. Pagneda demonstrated a minimal response time as low as about 50 ms, with a 10 ms increase per additional node, which shows how low one can get with a specialised solution.

Disco has shown that it scales fairly well as more nodes are added. Processing 1400 MB of data on 14 nodes takes a little – about 200 milliseconds – longer than processing 100 MB on 1 node. In addition to this we have found that run time increases linearly, proportional to the amount of data processed. Both these findings show that Disco is very predictable with regard to run time.

We must also consider the throughput of our benchmarked systems. This is where Pagneda has a clear lead on Disco. While Pagneda is able to process 14 GB of data on 14 nodes in about 20 seconds, Disco does the same in 234 seconds.

Disco appears to suffer from the fact that large parts of the processing chain are written in Python. Python being an interpreted language, it is clear and expected that the speed of a Python based system will be lower than a corresponding implementation in C. We see the effect of this very clearly. The overhead of function calls in Python is large enough to cause issues when there are tens of millions of them, as is not uncommon when processing flow data.

Another important difference between Disco and Pagneda is that Disco will store all intermediate data to temporary files, which we also expect to have an impact on the overall throughput.

To summarise, Disco is very efficient at scheduling jobs on the cluster. Where it lacks, however, is in processing performance. While we have taken measures to make processing as efficient as possible, by embedding C programs where we could, we are still seeing that Disco's worker code functions as a bottleneck.

# Chapter 8

# Conclusion and Future Work

In this chapter we draw a conclusion about the suitability of using the map-reduce model and frameworks for NetFlow data processing, and also suggest what future work can be done in this area based on the results we have found.

## 8.1 Conclusion

We have studied the applicability of the map-reduce model for NetFlow data processing. Flow processing generally involves three different operations: filtering, aggregation and sorting. The map-reduce model has been found to be an excellent match for filtering and aggregation. Sorting can also be implemented in the map-reduce model, but not optimally using the legacy map-reduce architecture. This is because all operations must be expressed as map and reduce functions, which do not work well with the most efficient sort algorithms, such as a merge sort.

We have implemented two prototype processing tools. Pagneda was written from scratch in C and functioned as a reference system. The other is based on the Disco map-reduce framework, and allowed us to evaluate the performance of a framework-based system in a real-world situation.

Through our benchmarks we have found that Disco can provide a response time of less than half a second to simple queries. Adding more nodes to the cluster, however, causes the response time to increase slightly due to additional overhead.

Disco has proven to scale quite well. The effect of adding more data and/or nodes is similar for the two prototypes. First, we see a minimal increase in execution time when increasing the number of nodes and amount of data proportionally. Second, the drop in run time is 35–40% when doubling the number of nodes. Finally, run time increases linearly, proportional to the volume of data processed.

The greatest difference between the framework-based prototype and the reference system is the throughput. In our benchmarks, Disco demonstrated a throughput of 4.28 MB/s per node, which is not up to par with what is expected and required. The performance penalty of Disco being based on Python, combined with the very large number of records that must be processed, shows that Disco is not even close to keeping up with an optimised system that is based on C.

To summarise, the prototype based on Disco displays scalability that is satisfactory, but the throughput is too low for a Disco based solution to be practical for real-world usage.

## 8.2 Future Work

There is clearly a need to work towards increasing throughput. Continued use of Disco is not likely to provide the desired performance. Based on the work we have done for this Master's thesis, there are in particular two possible directions that stand out.

It may be possible to employ a more efficient framework than the two we have considered, Disco and Hadoop. A new framework that claims to be more efficient than Hadoop, and is written in C++, is Sector [29]. This framework – and others as well – can be evaluated using the same methods we have used.

Another option may be to continue developing Pagneda. Pagneda was developed with the intention of only being a prototype, but it has already shown some potential, particularly with regard to performance. There are a number of aspects that must be covered, however:

- The query engine must be generalised to allow the input of filter, sort and limit expressions, as well as multiple aggregate queries.
- Sorting, limiting and merging, as well as arbitrary filter expressions, must be implemented.
- Implement a proper master process that is responsible for scheduling jobs efficiently.
- Implement support for reading standardised file formats.

We do not list features related to fault tolerance or incremental processing. Depending on the nature of the queries and how controlled the execution environment is, such features may not be critical.

There is no doubt that continued development of Pagneda is likely to take a lot of time. It would, however, give the advantage of ensuring fast and efficient processing, while maintaining full control of the internal workings of the tool.

# Bibliography

[1] J.T. Morken. Parallel Processing of NetFlow Data. Specialization project report, Department of Computer and Information Science, Norwegian University of Science and Technology, 2009.

[2] Cisco Systems, Inc. Cisco IOS NetFlow. Retrieved May 27th, 2010. http://www.cisco.com/web/go/netflow.

[3] Internet Engineering Task Force. Ipfix status pages. Retrieved May 24th, 2010. http://tools.ietf.org/wg/ipfix/.

[4] Cisco Systems Inc. NetFlow Version 9 Flow-Record Format. Retrieved June 5th, 2010. http://www.cisco.com/en/US/technologies/tk648/tk362/technologies_white_paper09186a00800a3db9_ps6601_Products_White_Paper.html.

[5] B. Claise. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. RFC 5101 (Proposed Standard), January 2008.

[6] Cisco Systems, Inc. NetFlow Services Solutions Guide. Retrieved May 27th, 2010. http://www.cisco.com/en/US/products/sw/netmgtsw/ps1964/products_implementation_design_guide09186a00800d6a11.html.

[7] SolarWinds. NetFlow Traffic Analyzer from SolarWinds. Retrieved May 27th, 2010. http://www.solarwinds.com/products/orion/nta/.

[8] Plixer International, Inc. Scrutinizer NetFlow & sFlow Analyzer. Retrieved May 27th, 2010. http://www.plixer.com/products/netflow-sflow/scrutinizer-netflow-sflow.php.

[9] NFDUMP. Retrieved May 24th, 2010. http://nfdump.sourceforge.net/.

[10] SiLK. Retrieved May 24th, 2010. http://tools.netsa.cert.org/silk/.

[11] B. Trammell, E. Boschi, L. Mark, T. Zseby, and A. Wagner. Specification of the IP Flow Information Export (IPFIX) File Format. RFC 5655 (Proposed Standard), October 2009.

[12] A. Wagner and B. Plattner. Entropy Based Worm and Anomaly Detection in Fast IP Networks. In *WETICE '05: Proceedings of the 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborat ive Enterprise*, pages 172–177, Washington, DC, USA, 2005. IEEE Computer Society.

[13] A. Lakhina, K. Papagiannaki, M. Crovella, C. Diot, E. D. Kolaczyk, and N. Taft. Structural analysis of network traffic flows. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of compute r systems*, pages 61–72, New York, NY, USA, 2004. ACM.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[15] Nokia Corporation. Disco. Retrieved April 15th, 2010. `http://discoproject.org/`.

[16] Apache Software Foundation. Welcome to Hadoop! Retrieved April 15th, 2010. `http://hadoop.apache.org/`.

[17] SALSA HPC, Indiana University. Twister: Iterative MapReduce. Retrieved May 27th, 2010. `http://www.iterativemapreduce.org/`.

[18] Wikipedia. Data structure alignment. Retrieved May 20th, 2010. `http://en.wikipedia.org/wiki/Data_structure_alignment`.

[19] The GNU Project. Variable Attributes - Using the GNU Compiler Collection (GCC). Retrieved May 20th, 2010. `http://gcc.gnu.org/onlinedocs/gcc-4.3.3/gcc/Variable-Attributes.html`.

[20] The GNU Project. Gzip. Retrieved June 1st, 2010. `http://www.gnu.org/software/gzip/`.

[21] Julian Sweard. bzip2. Retrieved June 1st, 2010. `http://www.bzip.org/`.

[22] Igor Pavlov. LZMA SDK. Retrieved June 1st, 2010. `http://www.7-zip.org/sdk.html`.

[23] T. Narten, R. Draves, and S. Krishnan. Privacy Extensions for Stateless Address Autoconfiguration in IPv6. RFC 4941 (Draft Standard), September 2007.

[24] Nokia Corporation. Technical Overview – Disco v0.2.4 documentation. Retrieved April 15th, 2010. `http://discoproject.org/doc/overview.html`.

[25] Y. Jia and Z. Shao. A Benchmark for Hive, PIG and Hadoop. `http://issues.apache.org/jira/browse/HIVE-396`.

[26] The Judy Team. Judy Arrays Web page. Retrieved June 3rd, 2010. `http://judy.sourceforge.net/`.

[27] Ville Tuulos. Disco v0.2.4. Retrieved May 27th, 2010. `http://github.com/tuulos/disco/tree/0.2.4`.

[28] The GNU Project. time. Retrieved May 27th, 2010. `http://www.gnu.org/software/time/`.

[29] National Center for Data Mining. Sector/Sphere: High Performance Distributed Data Storage and Processing. Retrieved June 8th, 2010. `http://sector.sourceforge.net/`.

# Appendix A
# Enclosed source code

Enclosed with this thesis is a ZIP file containing the C source code of Pagneda, our baseline system implementation, as well as the source code for an optimised aggregation reducer used in the Disco-based implementation.

- The source code of Pagneda can be found in the `pagneda` directory in the attached ZIP file.
- The source code of the optimised reducer used by Disco can be found in the `disco` directory of the attached ZIP file.

Both programs can be compiled using GNU Make, by running `make` in the directory where the source code is located.

# Appendix B

# Disco source code patches

These are patches against version 0.2.4 of Disco to make external processing tools work.

```
--- disco-0.2.4.orig/pydisco/disco/node/external.py     fcefbdd0
+++ disco-0.2.4/pydisco/disco/node/external.py  7c8d127f
@@ -1,5 +1,6 @@
-import os, os.path, time, struct, marshal
+import os, os.path, time, struct
 from subprocess import Popen, PIPE
+from disco import util
 from disco.netstring import decode_netstring_str
 from disco.fileutils import write_files
 from disco.util import msg
@@ -72,7 +73,7 @@
                                 return

 def prepare(ext_job, params, path):
-        write_files(marshal.loads(ext_job), path)
+        write_files(util.unpack(ext_job), path)
         open_ext(path + "/op", params)

 def open_ext(fname, params):
```

Listing B.1: Switch from marshal.loads() to util.unpack()

```
--- disco-0.2.4.orig/pydisco/disco/node/worker.py       7c8d127f
+++ disco-0.2.4/pydisco/disco/node/worker.py    68709339
@@ -358,7 +358,7 @@
                 else:
                        red_params = "0\n"

-                path = Task.path("EXT_MAP")
+                path = Task.path("EXT_REDUCE")
                 external.prepare(job['ext_reduce'], red_params, path)
                 fun_reduce.func_code = external.ext_reduce.func_code
         else:
```

Listing B.2: Fix incorrect path for worker's external reduce files

# Appendix C

# Benchmark scripts

The source code listings in this appendix shows the scripts used to execute benchmark. Listing C.1 shows a shell script used for the Pagneda benchmark. We do not list the scripts for all four benchmarks, as the differences are minimal. Similarly, Listing C.2 shows a script for benchmarking a Disco job.

```
#!/bin/sh

run() {
    echo $1
    for I in $( seq 1 $(( $NODES )) ); do
        export MACHINE="10.0.0.$(( $I + 1 ))"
        ssh $MACHINE "$1"
    done
}

runbg() {
    echo $1
    for I in $( seq 1 $(( $NODES )) ); do
        export MACHINE="10.0.0.$(( $I + 1 ))"
        ssh $MACHINE "$1" &
    done
}

export NODES=$1
export FIELDS="src_ip"

run "/home/jantore/master/src/stream/aggregate 9999 $FIELDS \
    >aggregate.out 2>aggregate.err &"
runbg "/home/jantore/master/src/stream/filter -n $NODES $FIELDS \
    >filter.out 2>filter.err < /data/split/$NODES"

echo "Filters scheduled, waiting for jobs to complete"

wait
```

Listing C.1: Pagneda benchmark shell script

```
#!/usr/bin/python

import sys
import os
import disco.core
import disco.util
import disco.func
import subprocess

def flow_reader(fd, size, fname):
    p = subprocess.Popen(
        [ '/home/jantore/master/src/stream/filter', '-c', 'src_ip'],
        stdin = fd, stdout = subprocess.PIPE, stderr = None
        )

    data = ""
    while True:
        read = p.stdout.read(13312)
        if len(read) == 0:
            break

        data += read
        end = len(data)
        pos = 0
        while pos <= end - 28:
            yield data[pos + 24:pos + 28], data[pos:pos + 24]
            pos += 28

        data = data[pos:end]

d = disco.core.Disco('http://netflow-test:8989/');

if len(sys.argv) < 2:
    sys.stderr.write("Missing number of blocks\n");
    exit()
```

```
blocks = int(sys.argv[1])
nodes = 14

job = d.new_job(
    name = "flow%02i" % blocks,

    input = [
        "disco://netflow%02i/in/incremental/%02i" % (node, b)
        for node in range(2, nodes + 2)
        for b in range(0, blocks)
    ],
    map = lambda e,a: [e],
    map_reader = flow_reader,
    reduce = disco.util.external([ "/home/jantore/aggregate.sh" ]),
    nr_reduces = nodes,
    scheduler = { 'force_local' : True },
    )

res = job.wait()
```

Listing C.2: Disco benchmark Python script