Masaryk University
Faculty of Informatics

Performance Measurement of Stream Data Processing in Apache Spark

Master's Thesis

Bc. Filip Halas

Brno, Spring 2017

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Filip Halas

Advisor: RNDr. Daniel Tovarňák

Acknowledgement

I would first like to thank RNDr. Daniel Tovarňák, RNDr. Filip Nguyen, RNDr. Milan Čermák and RNDr. Tomáš Jirsík for the guidance and supervision of this thesis. I would also like to thank and give credit to Mgr. Martin Jelínek for his cooperation on the core practical part of this work.

Abstract

The primary goal of this thesis was to implement application for measuring stream processing performance of Apache Spark on network traffic data and to perform performance benchmark in a distributed environment. The results of the benchmark are presented in the theoretical part of this work along with the description of the benchmark and the introduction of relevant technologies.

Keywords

security, stream processing, event processing, performance testing, streaming, network monitoring, NetFlow

Contents

1 Introduction **1**

2 Network Traffic Monitoring **3**
2.1 *Flow Analysis* . . . 3
2.1.1 NetFlow . . . 3
2.2 *Packet Analysis* . . . 5
2.2.1 Port Mirroring . . . 5
2.2.2 Deep packet inspection . . . 6

3 Stream Data Processing **7**
3.1 *Event Stream Processing* . . . 7
3.2 *Streaming Architecture* . . . 8
3.3 *Streaming Systems* . . . 9
3.3.1 Apache Kafka . . . 9
3.4 *Stream Processing Engines* . . . 12
3.4.1 Runtime Model . . . 13
3.4.2 Apache Storm . . . 13
3.4.3 Apache Samza . . . 14
3.4.4 Apache Flink . . . 14
3.4.5 Apache Apex . . . 15
3.4.6 Apache Beam and Cloud Dataflow . . . 16

4 Apache Spark **17**
4.1 *Concepts* . . . 17
4.1.1 Resilient Distributed Datasets . . . 17
4.1.2 Shared Variables . . . 19
4.2 *Runtime Architecture* . . . 20
4.2.1 Cluster Management . . . 21
4.2.2 Application Submission . . . 22
4.3 *Components* . . . 22
4.3.1 Spark Core . . . 22
4.3.2 Spark SQL . . . 23
4.3.3 Spark Streaming . . . 23
4.3.4 Spark Structured Streaming . . . 24
4.3.5 MLlib . . . 26
4.3.6 GraphX . . . 26

5 Stream Processing Benchmark **27**
5.1 *Specification* . . . 27
5.1.1 Selected Testing Scenarios . . . 27
5.2 *Testing Environment* . . . 28
5.2.1 Dataset . . . 28
5.2.2 Testbed Configuration . . . 29
5.2.3 Testing Methodology . . . 30
5.3 *Application Architecture* . . . 31
5.4 *Implementation in Apache Spark* . . . 33
5.4.1 Application Initialization . . . 33
5.4.2 Task Execution . . . 35
5.4.3 Flow Processing . . . 36
5.4.4 Helpers . . . 38
5.4.5 Configuration and Deployment . . . 38
5.4.6 Development . . . 40
5.4.7 Migration to Spark 2.1.0 . . . 41
5.4.8 Self-Contained Mode . . . 42
5.5 *Results* . . . 42
5.5.1 Framework Comparison . . . 46
5.6 *Potential Improvements* . . . 49

6 Conclusion **51**

Bibliography **53**

A Attachments **57**

1 Introduction

Flow aggregation and export, which is often used for network traffic monitoring and analysis, only allows for analysis on delayed batches of data collected over a certain interval since the previous batch. This approach does not allow for detection of potential threats or other anomalies in real-time. Over the past few years, several open source stream processing engines have emerged which can process BigData in real-time or near real-time, the newly emerged engines could potentially be deployed in distributed environment and used as network data analysis platforms.

The primary aim of this work is to determine whether distributed stream processing engine Apache Spark can be applied to real time network flow processing, which requires processing a large volume of small structured messages. The suitability of Apache Spark for flow processing is determined by implementing performance benchmark [1] based on common security analysis algorithms of NetFlow data and by experience obtained throughout the development. The implemented Spark benchmark and obtained data from testing in distributed environment form the practical part of this work.

The following chapter briefly describes the most commonly used method of network traffic monitoring and analysis using the NetFlow/IPFIX protocol. Chapter 3 introduces stream data processing paradigm and describes some of the most commonly used technologies in streaming applications which are able to process large amounts of data in parallel by distributing the processing of data into several computation units. Apache Spark is introduced and described in its own dedicated chapter in more detail with focus on its core concepts and its *micro-batch* streaming model used to implement the Spark benchmark application.

The last chapter describes the practical part of this work, the chapter consists of the benchmark specification and testing environment description, followed with a detailed overview of the implemented application and its architecture. The results obtained from the Apache Spark benchmark are presented in section 5.5 along with their comparison with results obtained from the benchmark implemented in Apache Storm and Apache Samza. Section 5.6 proposes several ideas

which could lead to improvements of the benchmark implementation in the future when technologies such as Apache Beam evolve.

2 Network Traffic Monitoring

Network traffic monitoring is an important task that is needed in many modern applications where network behavior needs to be understood or where even small downtime could cause severe problems. With network traffic analysis we can detect slow or failing components, occurring security breaches, learn which actions to take to improve application performance and more.

Network monitoring functionalities can be hard-coded directly into the router or require additional hardware and software, but offering greater flexibility. There are two main types of network monitoring tools based either on analysis of aggregated flow data or on individual packet analysis.

2.1 Flow Analysis

Flow represents a traffic stream in your network, typically it can be identified by a common set of attributes, such as source and destination IP addresses and port numbers in combination with information about the used protocol. If we observe network communication where any of the flow identifying attributes is different from the previous flows a new flow is created. Most common example is NetFlow (particularly v5, v9 and its variant IPFIX).

2.1.1 NetFlow

NetFlow is a network protocol developed by Cisco for the collection and monitoring of network traffic flow data generated by devices, such as routers and switches, that support NetFlow. It provides statistics on packets flowing through the NetFlow-enabled device to characterize network operation. NetFlow consists of three components:

- **Flow exporter (cache):** analyzes and collects IP packets and exports them as flow records to flow collectors
- **Flow collector:** collects, filters and stores data received from a flow exporter

- **Flow analyzer:** analyzes and presents processed flow data in some context (e.g., intrusion detection, bandwidth utilization) and in a more user-friendly way

Flow Record

Flow records are the basic output unit of flow exporter. They are created and updated through analysis of each packet forwarded within router or switch for a certain certain set of attributes that uniquely identify the direction and type of communication. Subsequent packets with the same IP headers update previously created flow records. Various information can be stored in flow records, such as:

- **Packet headers:** original information transmitted in each packet of the flow. Packet headers used for the identification of the flow include:
  - source and destination IP addresses
  - source and destination port number
  - input and output interface numbers
  - used protocol
  - type of service
- **Timestamps:** timestamps of the flow communication start and end
- **Duration:** total duration of the communication
- **Number of packets and bytes:** total number of packets and bytes transmitted in the flow
- **TCP flags:** union of all TCP flags observed over the duration of the communication

Flow records are ready for export when they are either inactive for a certain period of time (15 seconds by default) or active for too long (30 minutes by default) or when there was a packet with TCP FIN or RST flag signaling the communication end. Terminated or expired records are periodically exported to the flow collector which is responsible

for collection, analysis and storage of the flows and the production of reports used for traffic and security analysis. Flow collector can combine timed-out flow records of communications which took longer than the default 30 minutes. Data processed by flow collector are then presented in analysis application (flow analyzer).

There are several different flow record formats depending on the used version of NetFlow. The most commonly used versions are:

- **NetFlow v5:** available on wide range of devices, does not support IPv6, fixed flow record format
- **NetFlow v9:** template based (flexible flow record format)
- **IPFIX:** extension of NetFlow v9 with added support for variable length fields

2.2 Packet Analysis

Flow analysis is not sufficient in every area because in most cases the flow data do not provide information about the content of analyzed packets, flow analysis is suitable to determine overall network traffic statistics, but not when it is required to analyze packet content in depth. In some applications it might be necessary to inspect the data of each packet and then decide what to do with it. One of the common ways to obtain or capture packets for analysis is using *port mirroring*.

2.2.1 Port Mirroring

Port mirroring, sometimes refereed to by Cisco as SPAN (Switched Port Analyzer), is a packet capturing technique used on a network switch that is based on sending a copy of packets seen on the desired ports or even an entire VLAN (depending on what was selected for analysis) to another port for data collection and analysis. Port mirroring is supported on most network switches. An alternative to port mirroring is network tap (test access point) which is an external device used to monitor traffic between two network nodes. The device has two ports, one for each node it is setup to monitor traffic between, and additional ports to connect the analyzing device that receives the mirrored packets.

2.2.2 Deep packet inspection

Deep packet inspection techniques are based on the processing of packets in detail which allows for a more powerful analysis on extracted data than flow analysis. Packets can be obtained with port mirroring or network tap. In applications with high traffic the raw processing of packets can be very expensive in terms of hardware requirements. One of the uses of deep packet inspection is recognition of the applications that are being used in your network (content-based application recognition), in addition to flow analysis, application recognition using deep packet inspection can handle unusual or dynamic port numbers [2, 3, 4].

3 Stream Data Processing

With increasing demand for real-time data processing in scalable, modern applications in many fields (e.g., social networks, financial services) large streams of events, often from various data sources, need to be processed real-time and acted upon immediately, such as when we need to detect a potential attack from network activity logs or react on a system failure. A number of open source stream processing frameworks has emerged to address this problem, each with their advantages and disadvantages but often sharing a similar conceptual foundation.

3.1 Event Stream Processing

Structuring data as a stream of events is approach applicable in many different areas. The term event stream processing can be broken down into the following parts:

- **Event:** event is a record (collection of fields) of some fact which happened at a specific time (e.g., page visit or some error)

```
{
  "eventType": "pageView",
  "timestamp": "1413215518111",
  "ipAddress": "12.34.56.78",
  "pageUrl": "/index.html",
  "browser": "Chrome 56"
}
```

Figure 3.1: Sample Page View Event

- **Stream:** stream is a continuous flow of events relevant to your application (events from connected devices to your system)
- **Processing:** stream processing is making sense of the flow of events through analysis, it is about the identification of significant events (such as network attacks) to which we need to react immediately

Event stream processing focuses on real-time or near real-time analysis of the incoming time-based data as they are passing through the processing engine. There are two basic approaches to event processing:

- **Store every event:** storing every processed event as raw data for example in HDFS[1] and running a query on the stored data (or some subset) every time you need to analyze the data. This maintains a history of everything that has happened in your application, but as the data grows the cost of analysis increases as well.

- **Store a derived data:** computing some summary (derived data) from each of the processed events. Every time a new event is being processed, the relevant data to your application is updated accordingly and the event discarded afterwards which maintains a state in your application with only the data you want and results in significantly faster data analysis. This approach is also applicable to near real-time batch processing of events, where several events are processed and discarded at a time.

There are advantages and disadvantages to each approach, however, they can be used together which will lead to storing both raw and derived data and having the advantages of both approaches.

3.2 Streaming Architecture

There are many tools to chose from when designing a streaming architecture and each application might have different particular requirements, but a majority of the existing architectures use components that can be roughly classified into:

- **Producer:** producer is a system that collects data from datasources and produces event data in configured format to streaming system such as Apache Kafka. An example is Apache Flume[2] which has a source and sink architecture. Flume source collects

1. `https://hortonworks.com/apache/hdfs/`
2. `https://flume.apache.org/`

event data from data sources and Flume sinks puts the collected data into a streaming system.

- **Streaming system:** Streaming systems read and persist data from producers and send the data reliably to consumers. An example is Apache Kafka which is designed to handle large volumes of data and can linearly scale to deliver large quantities of events per second.
- **Consumer:** consumers are often stream processing engines which can manipulate or analyze data consumed from a streaming system. It is up to the processing engines to do actual work with the collected data. Examples are Kafka Streams[3], Spark Streaming, Apache Storm, Apache Flink and Apache Apex.

Systems such as Apache Kafka can be used both as a *streaming system* and also as a *consumer* (Kafka Streams), therefore the provided examples could possibly perform role of a different component in the streaming architecture than the component as which they are classified [5, 6, 14, 19].

3.3 Streaming Systems

The primary objective of a streaming system is to handle large volumes of data which requires distribution of work across cluster, system availability, fault-tolerance and optimized resource usage. There are many open-source streaming and messaging systems available, however, not many scale as well as Apache Kafka when it is needed to process and deliver large number of messages per second (100k+/sec) and fulfill the usual requirements put on a streaming system.

3.3.1 Apache Kafka

Apache Kafka is a distributed streaming platform which is run on one or more servers as a cluster. *Producers* send records to Kafka and Kafka stores the records and passes them to subscribed *consumers*. Processed messages are immediately stored to a filesystem when they

3. `https://kafka.apache.org/documentation/streams`

are received and retained for a configured period of time (retention period), the amount of data stored does not affect performance.

Use Cases

Although Kafka is classified here as streaming (or messaging) system, that is only one of its uses. Kafka can also act as a stream processing engine. Trivial message processing can be done using Kafka producer and consumer APIs, however, Kafka also disposes with its very own integrated streams API for complex stream transformations.

Kafka also comes with a tool called Kafka Connect and its very own connector API which allows to reliably connect existing Kafka topics to other data systems. For example, you can define a connector that moves data between Kafka and relational database.

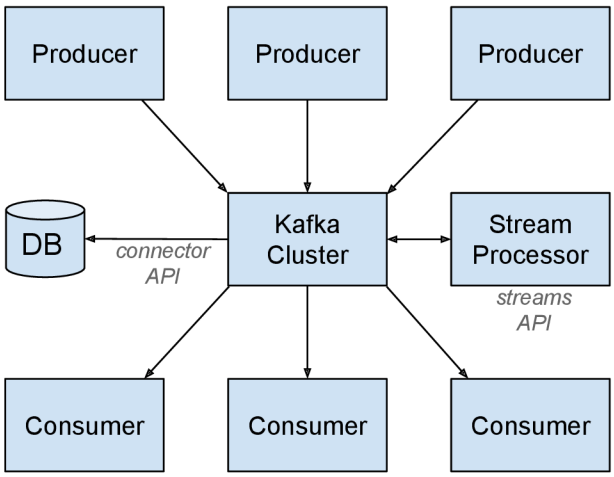

Figure 3.2: Kafka Architecture

Topics and Logs

Kafka uses *topics* as message streams of similar type or category. Producers publish messages to topics and consumers can subscribe and begin consuming messages from one or more topics. For each topic Kafka maintains a partitioned log to which producers append records

and consumers subscribe to the changes. Each partition contains an ordered, immutable sequence of records and every record has an unique id called *offset*. When producer publishes a message, it receives confirmation message with the offset of the newly published message. Usually, the first message in each partition gets the offset 0 followed by a message with the offset 1 and so on.

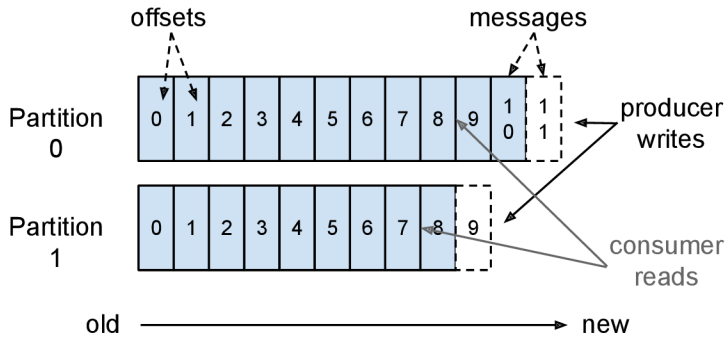

Figure 3.3: Anatomy of a Topic

As consumers read messages from Kafka they store their position in each topic as the offset of the last message read, in case their instance crashes and needs to resume consumption of the messages from the last known position.

Distribution

The partitions of each topic are distributed across the servers in the Kafka cluster, each server is assigned balanced amount of partitions and is responsible for handling their data and requests. Additionally, each partition can be replicated across multiple servers to ensure fault-tolerance.

Producers and Consumers

Producers publish messages to topics. Producer needs to determine which message will go to which partition inside the target topic. This

can be done with *round-robin* distribution or by implementing your own partitioning function (e.g., partition by message key).

Consumers belong to consumer groups. If a change happens to a subscribed topic, only one consumer instance in each subscribed consumer group will receive the new message. Consumer instances are either in separate processes or on different machines.

Kafka as a Messaging System

Kafka combines two traditional messaging approaches: *queuing* and *publish-subscribe*. The consumer groups generalize these two approaches. They allow you to divide up message processing over a collection of processes in each consumer group and as with *publish-subscribe* model, you are allowed to broadcast messages from Kafka to multiple consumer groups.

Kafka provides message ordering guarantees and load balancing over a pool of consumer processes. By default, Kafka guarantees *at-least-once* message delivery between consumer and producer, but consumers can also be configured for *at-most-once* and *exactly-once* message delivery guarantees [20, 21].

3.4 Stream Processing Engines

Stream processing engines are responsible for computations over consumed data as they come. A typical stream processing engine should have a *one-at-a-time processing model*, it should define operations that will be applied to each processed event individually and independently upon arrival with low latency.

There are many requirements a stream processing engine should fulfill such as automatic processing distribution with load-balancing, handling of stream imperfections (e.g., delays, out-of-order data), maintaining data integrity and offering mechanisms to recover from failures.

This section introduces some of the most actively developed open source stream processing engines currently available. Spark Streaming and Spark Structured Streaming also belong here, but are introduced separately in the Apache Spark chapter.

3.4.1 Runtime Model

Runtime model defines the capabilities of the system and its use cases, there are two main distinctive approaches of streaming engines: *native streaming* and *micro-batching*. Native streaming is when each consumed recored is processed immediately, one by one. Micro-batching is when small batches of records are created (typically over a constant period of time) before they are processed by the engine.

Micro-batching systems have higher latency and throughput, but they have cheaper cost of fault-tolerance and load-balancing than native streaming. Examples are Spark Streaming (page 23), Spark Structured Streaming (page 24) and Trident.

3.4.2 Apache Storm

Apache Storm is a native (event-by-event) streaming engine with very low latency, making it a fitting solution for systems where each event needs to be processed instantaneously.

Storm uses a high-level abstraction called *topology* to describe the steps that each consumed event will pass through in a form of computation graph. Topology consists of two types of nodes and connecting edges (streams):

- **Streams:** streams are unbounded sequences of data that are continuously arriving at the system
- **Spouts:** spouts are data sources at the edge of a topology
- **Bolts:** bolts are operations which are applied to their input stream as a processing step and return a new output stream as a result, they can consume any number of input streams

Using the components above you build a topology that consists of small, modular, discrete components. Storm uses Thrift[4] for topology definitions which means that topologies can be created using a large number of programming languages. A topology runs until you kill it. Storm has no order processing guarantee and *at-least-once* message processing guarantee, Storm ensures that each message is processed at least once but in some scenarios there can be duplicates.

4. `https://thrift.apache.org/`

Trident

Trident is a micro-batch processing alteration to the native streaming model of Apache Storm. Trident significantly increases processing latency due to its step-away from the event-by-event processing of Storm. Trident has some additional higher level operations like window, aggregations and state management which are not supported in Apache Storm. Trident, unlike Storm, has *exactly-once* message processing guarantee, making it more suitable for systems that cannot handle duplicate messages [22].

3.4.3 Apache Samza

Apache Samza is a stream processing engine, closely tied to Apache Kafka streaming system, designed to take the most out of Kafka. By default, Samza uses Kafka for messaging and Apache Hadoop YARN[5] for resource negotiation.

Samza works with streams (by default Kafka topics) broken down into partitions where each partition contains an ordered, immutable sequence of messages. Each message in the partition has an unique identification offset.

The stream processing logic (processing of input streams and sending messages to output streams) is put into Samza job which is then broken down into smaller execution units called tasks. Tasks are the units of parallelism of the job and partitions are the units of parallelism of the stream (Kafka topic). Each task consumes messages form one or more partitions of each of the job's input streams (there cannot be more tasks than input partitions). The YARN scheduler takes charge of distributing each task to a cluster node.

Apache Samza supports state management, uses YARN to ensure fault-tolerance and offers *at-least-once* delivery guarantee [23].

3.4.4 Apache Flink

Apache Flink is native streaming engine that also supports batch processing. Flink handles batches (bounded data) as a special case

5. `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`

of data streams with finite boundaries. Flink has DataStream API for working with unbounded data streams (native streaming) and DataSet API for working with bounded data collections (batch processing).

The DataStream API allows to process unbounded data streams in event-by-event fashion. Applications in Flink are composed of the following components:

- **Streams:** streams are unbounded, immutable datasets flowing through the system
- **Transformations:** transformations take one or more DataStreams and return a new DataStream
- **Sources:** sources of the DataStreams entering the system
- **Sinks:** data sinks are outputs for DataStreams to files, sockets and other external systems

The DataSet API for working with batches is similar to the DataStream API. Data sources can be files in some persistent storage such as HDFS, data sinks support the same built-in output formats for DataSets as for DataStreams. Transformations work on DataSets as well as on DataStreams, both APIs share a similar set of operations (e.g., map, flatmap, filter, reduce) but differ in some operations that do not make sense either for stream or batch processing.

Apache Flink supports execution of applications written for Apache Storm, offers *exactly-once* message processing guarantee, state management, low latency and high throughput rates with little configuration needed [24].

3.4.5 Apache Apex

Apache Apex is a native stream processing engine which also supports batch processing through an unified interface. Apex is bounded closely to the Hadoop ecosystem, using YARN for scaling and HDFS for fault-tolerance. It offers high throughput, low latency, state management and *exactly-once* message processing guarantee.

Apache Apex core platform is supplemented by a library of connector and logic functions called Malhar. Malhar includes connectors

for integration with great variety of commonly used file, database and messaging systems.

Streaming applications in Apex are represented by a Directed Acyclic Graph (similarly as topologies in Storm) composed of computation units called *Operators* connected with graph edges called *Streams*. Operators take in input streams and transform them to output streams which allows for separation of concerns into smaller, reusable, functional components [25].

3.4.6 Apache Beam and Cloud Dataflow

Cloud Dataflow is a fully-managed service for execution of both batch and stream processing jobs implemented with Apache Beam SDK. Dataflow is distributed as part of Google Cloud Platform (GCP)[6] allowing it to closely integrate with other Google cloud services.

At the beginning of 2016, the Dataflow model and SDKs were bundled into an incubator project called Apache Beam. The project has graduated from incubation with the Apache Software Foundation over the year and it now follows the community-driven development processes of the foundation with new contributors joining the original developers from Google.

The Apache Beam SDK which is now considered the Cloud Dataflow SDK provides an unified programming model for writing of both batch and stream data processing jobs that can run across a diversity of processing engine using runners. Currently, there are runners for Apex, Spark, Flink and Cloud Dataflow. There is also direct runner for testing and development purposes which can run jobs on your machine [26, 27, 28].

6. `https://cloud.google.com/`

4 Apache Spark

Apache Spark is an open source general-purpose cluster computing system designed to process big datasets fast, offering high-level APIs for Scala, Java, Python and R (since Spark 1.4).

With the increased need for parallel data analysis, Spark has emerged as an extension to Hadoop MapReduce[1] as a more user friendly solution, offering fast computation due to its complex distributed algorithm. Being a general-purpose system, Spark is suitable for various types of applications which can roughly be classified into data science tasks and data processing applications.

4.1 Concepts

4.1.1 Resilient Distributed Datasets

Spark resolves around the concept of *resilient distributed datasets* (RDDs), an RDD represents an immutable collection of objects that is partitioned and distributed across computation nodes in cluster. When writing applications in Apache Spark the elementary operations consist of creating a new RDD, transforming or manipulating an existing RDD or iterating through an existing RDD to compute a result. Apache Spark's distributed algorithm handles automatically the parallelization and distribution of data in your application RDDs across cluster.

RDDs can be created from any Hadoop supported storage source as well as from the local file system, RDDs can also be created by distributing an existing collection (e.g., list) by calling *parallelize* method in the driver program.

There are two basic types of operations that RDDs support: *transformations* and *actions*.

Transformations

Transformations create a new RDD by running operation on an existing RDD (Fig. 4.1), for example *filter(func)* is a transformation that uses a predicate function to match elements that it will return and

1. `https://www.tutorialspoint.com/hadoop/hadoop_mapreduce.htm`

map(func) is a transformation that applies a function to each element in the dataset, forming a new dataset from the values returned by the function.

Spark transformations are lazy evaluated (computed on demand), which means they are only computed when they are required to compute a result by an action, until then, only the order of transformations for the dataset is stored.

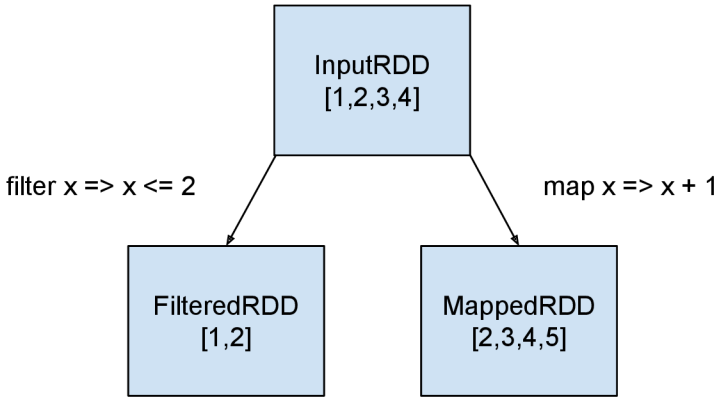

Figure 4.1: InputRDD transformation with *map* and *filter*

Actions

Actions are operations that return result to the driver program after performing computation on the input RDD. For example *reduce(func)* is an action that reduces all the elements of the input dataset using given function, operating on two elements and returning one element (each of the dataset type), eventually reducing the dataset down to a single value, *count()* is an action that returns the number of elements in the dataset and *first()* is an action that simply returns the first element in the dataset. An action returns a value to the driver program or it can write data to an external storage.

Key-Value Pair Datasets

There are certain operations and actions which are only available on RDDs with key-value pair data (for example *reduceByKey()*). For these operations to work, the structure of RDDs has to be stored in tuples.

Persistence

By default RDDs are lazy evaluated which might not be always desired if the plan is to reuse a computed RDD many times. However, Spark offers the ability to persist (or cache) an RDD for fast access in either memory, disk or both (depending on a chosen storage level). The stored data can also be serialized to make the objects take up less space at the potential cost of increased CPU time.

4.1.2 Shared Variables

Spark supports two types of shared variables: *Accumulators* and *Broadcast Variables*. First, we take a look at the concept of closure and why we need special mechanisms for shared data distribution in Spark.

Closures

In cluster mode, Spark driver will automatically divide RDD operations into tasks and distribute their computations across cluster nodes. Before each task execution a serialized closure is computed containing variables and methods that the *about to be computed* task needs access to. The computed closure normally contains copies of the driver variables, making it impossible to alter the original variable on the driver node by changing its passed value on a cluster node. This is demonstrated in the program sample shown in Figure 4.2, if a similar program runs in cluster mode, the printed value of *counter* will always result to its initial value 0, as no changes made to *counter* will be propagated back to the driver node.

If running Spark in local mode, the program in Figure 4.2 might behave differently, allowing to alter the original value of *counter* because the task could be executed in the same JVM as the one of the driver.

```
int counter = 0;
rdd.foreach(x -> counter += x); // Wrong
println("Counter value: " + counter);
```

Figure 4.2: Wrong Approach to Shared Variables [7]

Accumulators

Accumulators are variables designed to be safe and efficient. An accumulator is created in the driver node and can be passed down to cluster nodes and its changes get propagated back to the driver node. The cluster nodes can *add-only* to the passed accumulator through an associative and commutative *add* operation and the driver node is the only node that can access the accumulator's value.

Spark has native support for accumulators of numeric types and collections, accumulators of additional types can be implemented as well. Since Spark version 2.0 named accumulators can be created which makes their values displayed in Spark's web UI.

Broadcast Variables

Broadcast variables can be used to pass efficiently large data to cluster nodes for *read-only* access, serving as a cache to avoid passing data each time when a computation is requested. Spark driver initiates the broadcast variable, which can be any serializable type, and each cluster node will once receive its copy. Running tasks will then have fast, read-only access to the broadcast data on any cluster node.

4.2 Runtime Architecture

Spark uses master/slave model of communication, applications are designed to have one central coordinating (driver) node and several worker (cluster) nodes to which the master node distributes work. The driver splits Spark application into tasks and schedules their task execution to executors in worker nodes. An executor is a process responsible for task execution.

Spark applications can be launched either in *local mode* on a single machine or in *cluster mode* sharing the cluster resources. In local mode everything runs on the same JVM (driver, executors and tasks) in the same Java process, in cluster mode Spark application needs to connect to a cluster manager responsible for resource allocation across applications.

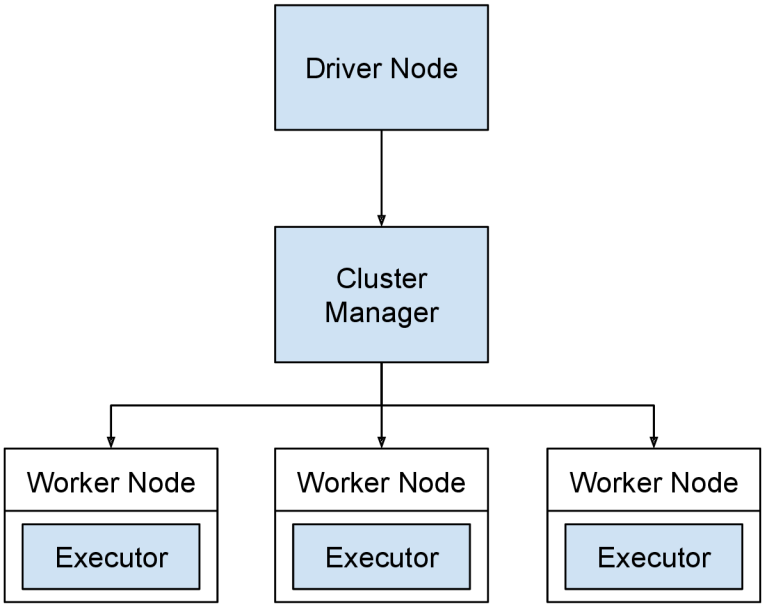

Figure 4.3: Spark Cluster Mode Architecture [8]

4.2.1 Cluster Management

Spark is aimed to scale efficiently up to large number of computation nodes which requires a cluster manager. Spark can use its own integrated cluster manager in Standalone Mode or run over an existing cluster manager system like Apache Mesos or Hadoop YARN.

Standalone Mode is a simple deploy mode included in the official distribution of Spark for launching applications in clustered environment. The cluster can be started with scripts already provided in Spark.

4.2.2 Application Submission

Spark provides a unified way to run programs on it through a script *spark-submit*. The script can submit application in either local mode or connect to any of the supported cluster managers, depending on the passed arguments. The script *spark-submit* launches the driver program which requests resources to launch executors from the cluster manager. Throughout the application invocation the driver sends tasks to executors depending on the RDD actions and transformations invoked in the program.

4.3 Components

Spark consists of several closely integrated components (shown in Fig. 4.4), Spark Core being the primary component, allowing the user to use other components as libraries (or modules) that each rely on the lower level Spark Core component.

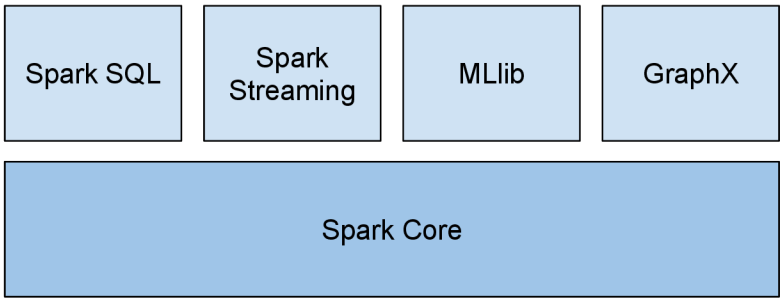

Figure 4.4: Apache Spark Components

4.3.1 Spark Core

Basic Spark functionality including task scheduling, memory management, fault recovery, interaction with storage systems and more. Exposes Spark's primary concept of *resilient distributed datasets* for working with data via Spark's RDD API.

4.3.2 Spark SQL

Spark SQL is a module for structured data processing through SQL interface. Data can be accessed from various data sources (e.g., JSON, Parquet, Hive Tables and JDBC). Spark SQL provides functionality to manipulate data through SQL, HQL[2] or a custom high-level Dataset API and to intermix SQL with data manipulations provided by Spark RDD API.

The primary concept of Spark SQL is *Dataset* which represents a distributed collection of data (objects with a defined schema). Dataset offers the convenience of RDDs with additional performance optimizations done by Spark SQL. Dataset can be organized into named columns forming a DataFrame, which is simply a Dataset of type *Row*. DataFrame is equivalent to database table (schema with rows that result from a query the DataFrame describes).

4.3.3 Spark Streaming

Spark streaming is a module for applications which need to process data as they come, it provides its own high-level abstraction *discretized streams (DStreams)* to simplify working with continuous data stream. It processes data in micro batches, depending on configured streaming interval, where each batch is internally one RDD of data processed over the duration of set interval since the last batch. In another words, DStream is internally a continuous sequence of RDDs where each RDD represents data from a certain interval.

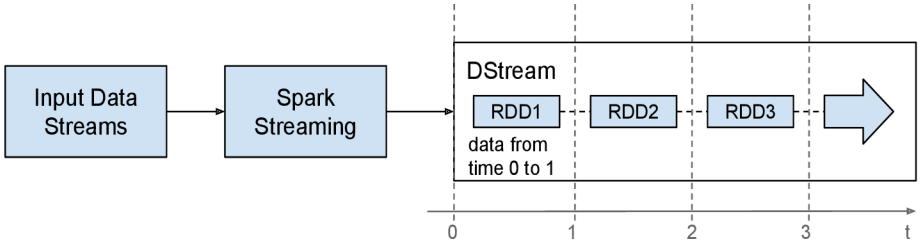

Figure 4.5: Spark Streaming Batch Processing

2. https://cwiki.apache.org/confluence/display/Hive/Home

DStreams can be transformed using transformations similar to RDD API, such as *map(func)* and *filter(func)*. DStreams also support additional time-related *windowed computation*.

Spark streaming can integrate with various real-time data sources (e.g., Apache Flume, Apache Kafka and HDFS) and supports *checkpointing*, a mechanism for fault-tolerance which works by storing sufficient amount of information in a reliable storage to serve as a recovery in case of failure.

Window Operations

In default stream processing, we are limited to operate with datasets that only contain data over a certain period of time (our set batch processing interval). If we process DStream with a *foreachRDD(func)* which applies our passed function to each RDD generated by our DStream then there is no knowledge of the already processed data in the previous RDDs.

Using window operations we have access to more than one dataset at a time, the amount of datasets we will have access to at once is set by parameter *window length* and the amount of datasets we skip after a window is processed is set by *sliding interval* parameter. This allows us to process data over a period that is a multiple of our batch processing interval [10].

4.3.4 Spark Structured Streaming

Structured streaming is a processing model introduced in Apache Spark 2.0, it is a set of additions to Spark streaming model that uses high-level DataFrame and Dataset API from Spark SQL to represent infinite datasets of structured data and the micro-batching concept of Spark streaming. Unlike in Spark streaming the batches are added up, meaning that we can execute some supported queries on all processed data up to a specific time.

The primary concept of structured streaming is to append all incoming data into an unbounded input table, where each new streamed object will be appended as a new row into the table. The user defines queries to execute on the input table and triggers (which set how often the queries should run). A query is executed repeatedly (by a trigger)

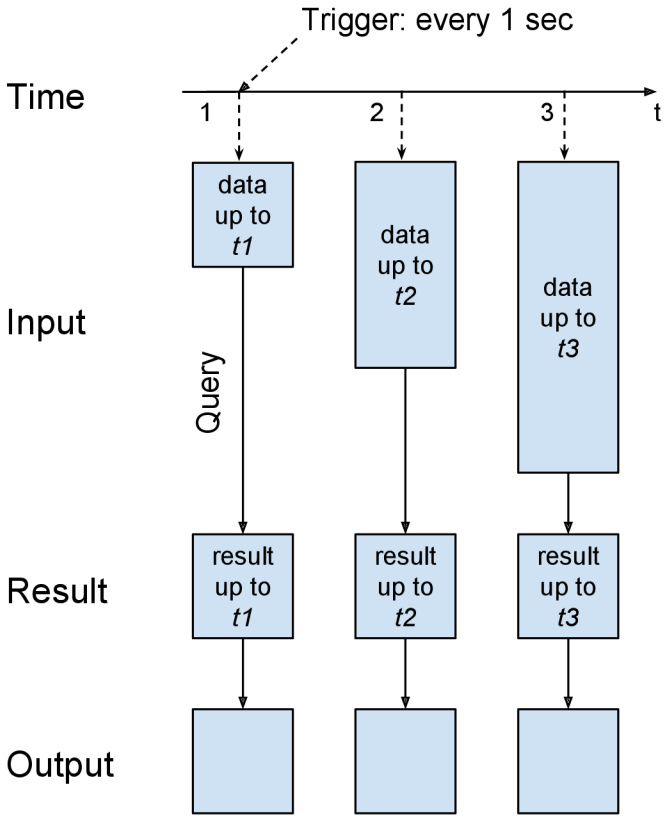

Figure 4.6: Programming Model for Structured Streaming [7]

and each time produces a *result table*, the result table is written every time to an *output sink* in a way that depends on the user specified *output mode*, the supported output modes are:

- **Append (default):** writes a result table with results only for new records since the last triggered query
- **Complete:** writes a complete result table for records up to now
- **Update:** writes a result table with results only for updated records since the last triggered query

In some output modes the support of streaming queries is limited, for example complete output mode only supports aggregation queries.

Structured streaming is still undergoing development and is in alpha version at the time of writing, so a lot might change in the upcoming releases of Apache Spark [7, 11, 12, 13].

4.3.5 MLlib

MLlib is scalable machine learning library, included in Apache Spark, which contains machine learning functionality. MLlib contains common machine learning algorithms (classification, regression, clustering and collaborative filtering) and offers functionality such as model evaluation and data import.

4.3.6 GraphX

GraphX is a component for graph manipulation and graph-parallel computations. GraphX extends the Spark RDD API and uses the abstraction *directed multigraph* with properties on each vertex and edge to represent data. GraphX provides structural and join operators (subgraph, joinVertices) for graph manipulation and includes common graph algorithms (PageRank, connected components and triangle counting) to make data analytics easier.

5 Stream Processing Benchmark

The primary aim of this work was to develop a configurable tool for automatic performance measurement of distributed event stream processing in Apache Spark and to perform measurements on common security analysis tasks of NetFlow data in distributed environment. Similar performance measurements were taken in Apache Storm and Apache Samza and the results of their comparison with Apache Spark were published in article *A Performance Benchmark of NetFlow Data Analysis on Distributed Stream Processing Systems* [1] to determine the suitability of distributed Stream Processing Engines to real time network flow processing.

5.1 Specification

To determine the suitability of a particular stream processing framework a set of basic operations included in the majority of security analysis tasks was to be implemented and used to measure performance of each stream processing engine.

The tests had to be executed in various cluster setups (different number of virtual machines or CPU cores) on prepared dataset of flow data to find out how a different cluster setup can alter the performance of the framework. This put requirements on the benchmarking tool which needed to be configurable and usable in various environments and also automatically executable so it would be possible to take multiple measurements without the need to execute each test manually.

5.1.1 Selected Testing Scenarios

The selected testing scenarios correspond to operations commonly used in security analysis tasks, supplemented by an *Identity* operation without any data processing done to determine the base throughput of the processing engine. There is also a scenario *Syn DoS* which simulates a real network attack detection method and requires all the previous operations to be applied on the input stream.

- **Identity:** used to determine the base throughput of the engine without any data processing done on the input stream (originally each input message was sent to Kafka output)
- **Filter:** each message in the input stream is tested to match a specific *destination IP* and the matching messages are counted (originally each filtered message was sent to Kafka output)
- **Count:** each message in the input stream is filtered by a specific *destination IP* and the filtered messages are counted, only one result message with the counted *destination IP* and number of all flows containing the IP is sent to output
- **Aggregation:** for each processed message we update an aggregated summary to contain data with processed packet count for each *destination IP*, only one result message containing a *Map* of destination IP addresses as keys and their packet counts as values is forwarded to output
- **Top N:** additional post processing for the aggregated packet count summary which computes result as *Top N destination IP* addresses with the largest amount of transferred packets
- **Syn DoS:** each flow is tested to contain only *TCP SYN* packet type, matching flows are counted and aggregated by *source IP* and at the end *Top N* elements are selected and sent to output

5.2 Testing Environment

5.2.1 Dataset

The performance measurements were done on a dataset based on real network traffic. The basis of this dataset was a network traffic sample taken from *CAIDA*[29] dataset. The original dataset stored in *PCAP* format was transformed to flow data represented in *JSON* which is commonly used format to represent structured data in distributed systems. The first one million flows for one selected destination IP (50.224.90.224) were taken as the initial data of the dataset and then repetitively inserted into the final dataset with modified IP address in

each repetition. The resulting size of each message was 270 Bytes on average. An example of a single message from the resulting dataset is shown below:


```
{
  "date_first_seen":"2014-01-16T12:49:46.591+01:00",
  "date_last_seen":"2014-01-16T12:49:46.591+01:00",
  "duration":0.000,
  "src_ip_addr":"146.36.139.66",
  "dst_ip_addr":"62.148.241.49",
  "src_port":42614,
  "dst_port":25,
  "protocol":6,
  "flags":"....S.",
  "tos":0,
  "packets":1,
  "bytes":48
}
```


Figure 5.1: Sample Dataset Message

5.2.2 Testbed Configuration

The performance measurements were done on a dedicated cluster of 7 VMware vSphere 6.0 nodes. The tested processing engines were deployed on the cluster along with Apache Kafka and Apache ZooKeeper[1] which Kafka requires. Each of the VMware cluster nodes had the following specification:

- 2 x Intel®Xeon®E5-2670 (16/32HT cores in total)
- 192GB 1600M MHz RDIMM ECC RAM
- 2 x HDD 600GB SAS 10k RPM, 2,5" (RAID1)
- 10 Gbit/s network connection, 1 Gbit/s virtual NICs

There were four different configurations for each VMware virtual machine available in the cluster which allowed for convenient testing

1. `https://zookeeper.apache.org/`

in environments which differed in the number of virtual CPUs and available memory. The VMware machine configurations differing in the amount of available memory and number of CPU cores are shown in Table 5.1.

Type	vCPUs	Memory	Drive
vm_large	32	128GB	300GB
vm_normal	16	64GB	300GB
vm_medium	8	32GB	300GB
vm_small	4	16GB	300GB

Table 5.1: VMware virtual machine configurations [1]

Two nodes were dedicated to Apache Kafka which produced stream of flow data to the tested systems and consumed computed results. The Kafka nodes were always running on *vm_large* virtual machines to ensure Kafka would produce flows at maximum speed and not bottleneck the processing performance of the tested engines. The remaining nodes were fully available for utilization by each tested streaming engine.

The following software was installed on the cluster nodes:

- Debian Linux 8.1.0 x64
- Oracle Java 1.8.0
- Scala 2.9.2
- Apache Hadoop 2.7.1
- Apache Zookeeper 3.4.5
- Apache Kafka 0.8.2.1
- Apache Spark 1.4.1
- Apache Storm 0.9.4
- Apache Samza 0.8.0

5.2.3 Testing Methodology

The tested streaming systems were processing input stream produced by Kafka consisting of 100 million flows (27 GB total size). The number of partitions of the dataset was set to the number of available CPU cores to the tested system in the benchmark or their multiples, this was necessary for full utilization of all CPU cores by Spark and Samza.

The input Kafka topic was being filled with a separate tool *ekaf-sender*[2] (implemented by Daniel Tovarňák and Martin Laštovička) which would fill the topic during each test run. This tool was also used to measure the time of each test, which was done by logging the time when filling of the Kafka topic started (at the beginning of each test) and then waiting for each test to finish processing the whole dataset and sending a special result message to a designated Kafka topic which was used to determine the test finish time. It was ensured that the performance of filling the Kafka topic with the dataset was significantly faster than the processing speed of each tested system, as otherwise it would possibly bottleneck the measured performance of the streaming systems [1].

5.3 Application Architecture

The Spark benchmark was developed to use Kafka for both input and output data streams, but should work on any Spark supported input source with minor adjustments. Due to the Runtime Architecture of Apache Spark, in distributed mode it is required to have one central driver node to coordinate and distribute work across other worker nodes. Local mode had to be used for execution of the Spark benchmark on a single machine. Running Spark applications on two machines in distributed mode is not supported, as this results in one node coordinating the other node and either the driver or the single worker would be performing computations.

Abstraction of the Spark benchmark deployed on four cluster nodes is shown in Figure 5.2. The driver is responsible for the input stream distribution across the worker nodes, processing of results computed by the worker nodes and producing output back to Kafka. Each worker has assigned its own Kafka input stream receiver, so the actual input stream processing is done in parallel. The application uses accumulators as shared variables for computation of the test results. Each worker node updates the accumulators passed by the driver at the end of processing of each micro-batch partition with results computed over its assigned data. After the distributed processing of the entire

2. `https://github.com/xdanos/ekafsender`

dataset is finished, the driver performs additional post processing and sends result to the designated Kafka topic.

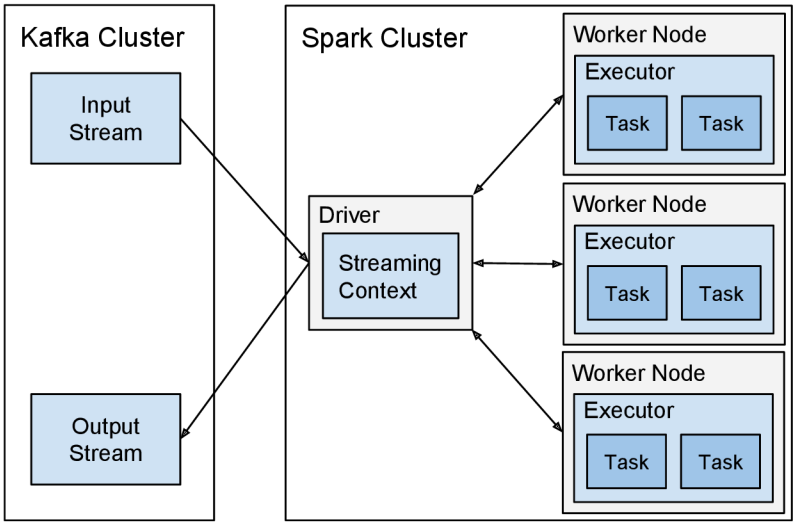

Figure 5.2: Spark Benchmark Architecture in Distributed Mode

To execute the application the user is required to configure the properties in *pom.xml* file, variables in *bin/setenv.sh* shell script and to specify the desired testing scenarios which should be executed in the *bin/all-tests-read.sh* script with the amount of repetitions of each test before the script execution. The benchmarking application is then automatically sequentially executed by the *bin/all-tests-read.sh* script with different configuration parameters in each run. After the benchmark is submitted to Spark the shells script executes Kafka console consumer configured to only consume one new message from the result topic. After the consumer consumes the expected single result message (produced by the application after the result computation) the script prints the result to standard output and continues with the execution of the benchmark application with the next testing scenario. The script does not require any user input and after it finishes with the benchmark execution of all the configured testing scenarios the results will remain available in the result Kafka topic.

5.4 Implementation in Apache Spark

The benchmark of distributed stream processing in Apache Spark was implemented using Spark's Java API (in Java version 1.8). The core functionality was implemented using Git in cooperation with Martin Jelínek in Apache Spark 1.4.1. The benchmark uses primarily Spark Streaming module for stream processing, Maven for dependency management and build automation, and shell scripts for the distribution and automatic execution of the benchmark in various setups.

5.4.1 Application Initialization

The main Java application is configured using *pom.xml* properties, assembled into a jar with all dependencies and submitted to Spark via Maven exec plugin which executes the *spark-submit* script. The script takes first the Java virtual machine arguments, followed by the path to the compiled jar of the application. The last arguments are passed to the user program. The following arguments are used to set Spark configuration properties at runtime:

- **class**: the application entry point
- **master**: the cluster master node URL or a value that sets the amount of worker threads in local mode
- **deploy-mode**: sets whether the application will be executed on a running driver (cluster mode) or locally
- **driver-memory**: allocates memory to the Spark driver node, the amount of driver memory cannot be set in the application after it has already started and the memory has been allocated to the Java virtual machine

The remaining used configuration properties replace the variables used in **.properties* files during the Maven process-resources phase and are parsed into `java.util.Properties` objects during the application initialization using the functions provided by the utility class `PropertiesParser.java`. The parsed Spark properties are set directly on the `SparkConf` object used to initialize the `StreamingContext` (the

main entry point of the Spark streaming functionality). The configured Spark properties are listed and available for checking under the *Environment* tab in the Spark's web UI of a running driver.

The application main class is `App.java` which contains the function *main* executed immediately after the application is submitted and initialized. Two arguments are required, configured as properties in *pom.xml* and overridden with Maven command-line arguments if the application is executed during automated testing:

- **Machine count**: the total amount of cluster nodes the application will be executed on (used for the computation of Kafka receivers which can run on the cluster)

- **Test class**: defines the class which will be applied to the micro-batches formed from the input data stream and the type of additional post processing done after the whole input dataset will be processed

After the processing of arguments, the application initializes Spark `StreamingContext` with a predefined micro-batch interval of one second and `SparkConf` object containing several configuration parameters initially set in *pom.xml*.

In the next step, the application initializes Accumulators which are used in computation of the result data. The core part of the application uses default numeric accumulators provided in Spark and custom class `MapAccumulator` implemented as accumulator of type `Map<String, Integer>`. The custom accumulator class implements `AccumulatorParam` interface with method to obtain the initial value of a new accumulator and method for adding additional data to the accumulator value which corresponds to merging of two `Map` objects together.

Spark streaming is configured to receive data from Kafka with the use of receivers implemented using the Kafka high-level consumer API. Kafka streams are created and configured with the use of `KafkaUtils` object which contains the factory method for discretized stream creation. The amount of initiated Kafka streams depends on the number of machines available in the cluster, however the total amount needs to be lower or equal to the number of input topic partitions. Final

input `JavaPairDStream` object is created as an union of every initiated Kafka Stream with the use of the *union* DStream transformation.

5.4.2 Task Execution

When the application will be executed it will start streaming until the whole input data stream is processed and depending on the passed *test class* argument, Spark will be set to concurrently execute jobs defined by a specific class (which is basically a function) for each micro-batch RDD formed from the union of all input streams at a time.

Spark driver will create a streaming job every batch duration to process data pulled from the input stream in that particular batch interval, the job will be divided into tasks and the task execution scheduled to executors running indefinitely on worker nodes (or on local threads in local mode). Kafka receivers will run on worker nodes and store external data in executors responsible for the data processing in assigned tasks from the driver process. Executors have their own in-memory storage for RDDs cached in the application.

The class that will be executed for each RDD implements an interface which represents a function in Spark's Java API. Although both anonymous classes and lambda expressions can be passed to *foreachRDD* instead of regular classes, the use of external classes has lead to separation of concerns and modularity at a sustainable cost of code readability which could have been gained with the use of lambda expressions. The function *foreachRDD* used for the RDD processing logic of each test is always executed on the driver node with a partitioned micro-batch RDD passed as an argument from the input stream. RDD partition is a logical chunk of the whole micro-batch, by default, RDDs are partitioned automatically and their amount corresponds to the amount of available CPU cores.

Every *foreachRDD* call will create a streaming job in the driver process and the driver will then divide the job execution into tasks with the *foreachPartition* function call which will handle the actual message processing logic of every partition of the dataset. The processing of each RDD partition will be done on a worker node as a standalone task. The executor responsible for a particular task execution will increment the values of the passed accumulators with the computed data over messages in each partition via passed local references to the accumu-

lator values. After a task will finish its execution the accumulators will be updated with the computed values on the driver node.

5.4.3 Flow Processing

Simultaneously with the indefinite streaming the application will execute a separate *thread* which will check in 500ms interval whether the whole input dataset was processed. At the end of the input dataset processing, the separate thread will perform additional post processing on the driver node required by the running test and will send a single result message to a designated Kafka topic. The separate thread is also responsible for regular printing of record processing speed estimates which can be found in the Spark's web UI. The estimates are not very accurate and were not used as the benchmark results because they are measured internally and the Spark application requires some variable amount of time to initialize streaming each time its executed.

The input stream processing logic is contained in separate classes (depending on the executed test type) and applied to *micro-batches* formed from the input stream every batch duration. The RDD processing operations are executed on available worker nodes unless the application runs in local mode. Every test has access to accumulators it needs to update including a counter of all processed records and during its execution the test iterates over each flow in the assigned RDD. Deserialization of input flow data is done using the *Jackson* library which maps *JSON* data to *POJOs* (plain old Java objects), this has proven to be the most efficient way of processing *JSON* data during the development of the benchmark.

The following classes located in the *cz.muni.fi.spark.tests* package have been used to implement the specified testing scenarios:

- **ReadWriteTest (Identity):** does not perform any processing, only counts all messages of every RDD, at the end of processing the temporary counter of processed records is synchronized with the accumulator of total processed records on the driver (the name of the test class remained from the originally intended functionality of this test where each message would be forwarded to Kafka output stream).

- **FilterIPTest (Filter):** uses `ObjectMapper` to parse every input flow into `Flow.class` and tests whether the *destination IP* of the input flow matches a specific constant IP, the matching flows are counted (original intention for this test was that each filtered message would be forwarded to Kafka output).

- **CountTest (Count):** filters input flows in the same way as *FilterIPTest* and counts the total number of packets of the specific filtered IP that were transmitted in all flows of the input dataset.

- **AggregationTest (Aggregation, Top N):** uses `Map` accumulator to count the total number of packets that were transmitted for each *destination IP* instead of only counting flows for a single IP.

  Top N uses the same class as *Aggregation* scenario, only after the whole input stream is processed the computed `Map` is sorted by value and its *Top 10* elements with highest message counts are considered the test result.

- **SynScanTest (Syn DoS):** uses `Map` accumulator to count messages containing only *TCP SYN* packet for each *source IP*. After the whole input stream is processed the computed `Map` is sorted by value and its *Top 100* elements are taken as the result.

  SynScanTest uses intermediate filtering to avoid bottleneck caused by slow `Map` accumulator synchronization by discarding IP addresses that only contained one message with only *TCP SYN* packet during the processing of a single batch.

The Spark Streaming module supports high-level DStream transformations (which take an input DStream and pass each element of the stream through a function call to return a new DStream) such as *map(func)*, *filter(func)* and *reduce(func)* which can be applied to each RDD of the input stream and chained together. These transformations provide alternative solution to the test implementations with use of the *foreachRDD* and *foreachPartition* operations.

There was an attempt to use the provided transformations, but in the end they were not used because they do not provide increased performance and are not well suited for functions with side effects, such as the sending of filtered messages to a Kafka topic. With the used

approach, there is only one iteration for each message of every RDD, with the use of DStream transformations this would not be possible, because every transformation returns another DStream which would have to be processed with some kind of iteration anyway. One benefit gained from the use of *foreachRDD* and *foreachPartition* operations in every test was the provided flexibility to make minor adjustments to the test implementations without having to make significant code changes.

5.4.4 Helpers

Standalone Kafka producer was implemented in Java for filling of the input Kafka topic with a file content of the dataset. This producer was used only during the development of the benchmark, but is provided in attachments for reference. The producer is configured in *pom.xml*, where it is required to specify the input file path and the output topic name. The primary function *main* in *FromFileProducer.java* processes the input file lines sequentially and sends content of each line as a single message to the output Kafka topic. Additionally, there is an implementation of Kafka consumer intended for result message processing which consumes new messages but is set to terminate after consuming only one message from the input topic, this desired functionality was achieved with the use of the default *kafka-console-consumer* provided by Kafka with optional argument `--max-messages` set to 1.

The Spark application relies on its own custom implementation of Kafka producer implemented in the *OutputProducer.java* class for sending messages to either the default *producer.topic* (set in *pom.xml*) or to a topic passed as an optional argument to the function *send*.

5.4.5 Configuration and Deployment

For automatic test deployment and cluster management several shell scripts were implemented. These scripts can be used to prepare the cluster environment for testing by installing and executing Apache Spark in standalone cluster mode, and also to submit the compiled Java application to a running Spark cluster. Below is an explanation of the cluster management scripts functionality:

- **bin/setenv.sh:** contains configuration variables shared by all the other scripts, most importantly:
  - enumeration of cluster nodes that will be used to setup Spark driver and slave nodes
  - working directory that will be used by all other scripts on each Spark cluster node
  - Kafka input producer and output consumer nodes, including path to Kafka installation on these nodes
  - Spark master URL used for establishing connection to the Spark standalone cluster master, the URL is printed to standard output when launching spark master node via the *spark-master.sh* script and needs to be set here manually
  - Spark mirror used as a source of Spark distribution
- **bin/install-spark.sh:** downloads and extracts compiled version of Spark into the working directory on a single cluster node
- **bin/install-cluster.sh:** performs *bin/install-spark.sh* on every Spark node on the cluster
- **bin/start-cluster.sh:** initiates Spark master and slave nodes in standalone mode by executing scripts *sbin/start-master.sh* and *sbin/start-slave.sh* provided by Spark
- **bin/kill-cluster.sh:** kills all Java processes on every Spark node
- **bin/clean-cluster.sh:** cleans working directory on every Spark node
- **bin/restart-cluster.sh:** by using all the scripts above this script resets the environment on every Spark node and initiates cluster in standalone mode
- **bin/run-topic.sh:** recreates Kafka topic on a single cluster node (topic deletion needs to be enabled in Kafka config)

The following scripts are responsible for submitting the Java application to a running Spark cluster:

- **bin/deploy-to-cluster.sh:** performs the build and distribution of the Java application across every Spark node and submits the application to Spark driver for execution, the script can be used for execution of a single test until it is manually terminated
- **bin/run-test-read.sh:** extension of *bin/deploy-to-cluster.sh* which executes the Java application with passed arguments and starts a Kafka consumer awaiting a single result message which will be sent from the application to a designated topic as a signal of the test end, after the test finishes the script restarts cluster environment to prepare for execution with different arguments
- **bin/all-tests-read.sh:** script for sequential automatic testing of selected test types on various number of cluster nodes, executes each selected test one after another and each test is executed in various selected cluster setups differing in the amount of virtual machines (e.g., if set to run identity and filter test on five, four and three cluster nodes the script will execute $2 * 3 = 6$ tests in total)

5.4.6 Development

During the development of the benchmark there were several changes which had to be done as a response to either some limitation discovered in one of the tested frameworks or because of a potential bottleneck which could affect the benchmark results. This has lead to diverting from the original idea of how the benchmark would be implemented. At first, Kafka input stream was supposed to be filled only once with the whole dataset and each tested framework would reset its consumer topic offset and read data from beginning before each test. During the final testing however, Kafka was being filled with data live, simultaneously with the tested framework consuming the data produced.

The initial development and debugging of the benchmark was performed on a testing cluster with slightly different specifications of the testing scenarios, this can be seen in several differences between the code in the *master* branch compared with the code in the *cluster-tests* branch which was used for the actual benchmark. The final implementation adjustments and benchmark of each framework were performed

during a period of several weeks on the *testbed* cluster using Apache Spark 1.4.1. After the benchmark, the *master* branch was updated with the script and code improvements discovered and performed during the benchmark, however the *master* branch remained configured for the testing cluster.

After the benchmark on the *testbed* cluster was done, several adjustments and extensions were made to the Spark benchmark application in the following order:

- export of flow statistics (done by Martin Jelínek)
- migration of Spark from version 1.4.1 to 2.1.0
- adjustments of the benchmark for execution in local mode

5.4.7 Migration to Spark 2.1.0

Migration of Apache Spark to version 2.1.0 consisted of several steps. Updating to version 1.5.0 introduced a new way of starting slave nodes via Spark provided *sbin/start-slave.sh* script. Since 1.5.0 it was no longer possible to parse and store ID of the Spark driver which could have been used to terminate the driver application by passing the driver ID to Spark provided *./bin/spark-class* script. The new way of killing the driver (done before the execution of another test) became with use of the script *bin/kill-cluster.sh* which terminates all Java processes on every Spark node in the cluster. This put a new requirement of restarting the driver and slave nodes before each test execution.

Spark 2.0.0 introduced a breaking change in the processing of the input *JavaPairDStream*. Function *foreachRDD* applied to each micro-batch formed from the input stream was no longer applicable to the implementation of *org.apache.spark.api.java.function.Function* interface passed as an argument and required instead an implementation of *VoidFunction* (found in the same package) due to API changes introduced in this Spark release. Version 2.0.0 also introduced a new accumulator API with added support for named accumulators which have their values automatically displayed in the Spark's web UI during the application execution.

5.4.8 Self-Contained Mode

The ability to execute the Spark benchmark in local mode required a different initialization of the input *JavaPairDStream* object. Transformation of several DStream objects into a new single DStream (their amount depending on the desired number of Kafka consumers) was replaced with a single DStream creation, because the processing of input cannot be distributed across several nodes when running on a single machine.

The Maven configuration for local mode deployment was put to a separate profile named *local* in *pom.xml*. For the automatic testing in local mode a new script *bin/all-test-localhost.sh* was created, containing all the necessary configuration variables and functionality (without any dependencies on any of the other scripts). The new script is similar to *bin/all-test-read.sh* used for automatic testing in standalone cluster mode with several differences. The script uses Screen window manager to start detached named terminal session each time a new test is executed to separate Spark output from the benchmark terminal output. After a result message is received by the designated Kafka topic the screen session is terminated and another test run can continue. Several steps are required for the execution of the script:

- Apache Zookeeper and Kafka needs to be running on localhost
- Topic set in *pom.xml kafka.consumer.topic* has to be filled with input flow data and the total amount of messages should be equal to the value of *TEST_DATA_RECORDS_SIZE* variable set in *App.java*
- Spark distribution has to be downloaded into the directory set in *pom.xml spark.home* property

5.5 Results

The benchmark was performed on the *testbed* cluster on one and four cluster nodes in four different configurations. The number of partitions of the input dataset was always set to the sum of all CPU cores available for each benchmark. As partition is the default unit of parallelism in Spark, if there would be less partitions than the number of available

CPU cores then not every core would be utilized. Although having more partitions is usually better than not having enough, the first benchmark has shown that a lot of partitions can lead to the decrease of performance.

Each test was executed ten times using modified version of script *bin/all-test-read.sh* (except for one test with only one result possibly due to outage). The benchmark results are the amounts of processed flows per second with Apache Spark executed for each testing scenario. The results are presented in terms of thousands of flows processed per second on average (k flow/s) computed from the total size of the dataset and benchmark execution time. Computed average of each test is compared with the observed worst (minimum throughput) and best (maximum throughput) cases over the course of all runs.

First benchmark was performed on a single *vm_large* machine with 32 CPU cores (Figure 5.3) where the average base throughput without any processing done on the input flows reached almost 1200 k flow/s. The throughput decreased for all the other operations to approximately 650 k flow/s.

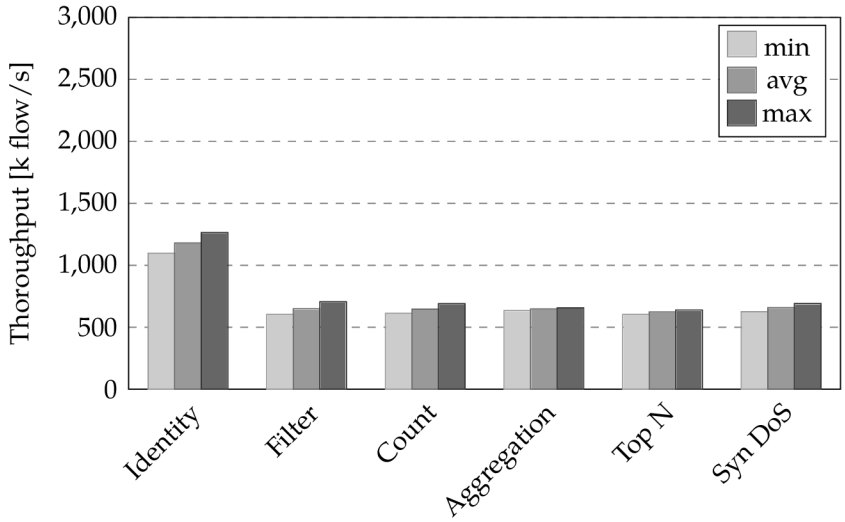

Figure 5.3: Spark benchmark on 1 *vm_large* machine (32 CPU cores)

The second benchmark was performed on a single machine with only 16 CPU cores. The results (shown in Figure 5.4) depict an overall

increase in throughput on every test confirming a suspicion that the significantly lower throughput which resulted on the first benchmark was caused by shuffling (redistribution of data across partitions) of incoming messages before they were being processed, possibly because of network and I/O overload which has lead to increased task scheduling time. Using only 16 CPU cores the average base throughput increased to almost 1500 k flow/s and the performance on every other operation has increased by at least 40%.

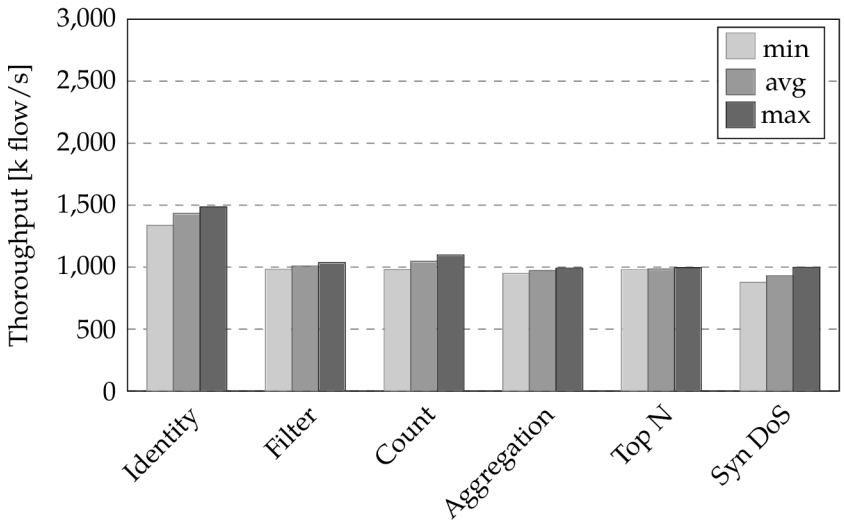

Figure 5.4: Spark benchmark on 1 *vm_normal* machine (16 CPU cores)

Another benchmark was performed on 4 *vm_medium* machines connected in a network with 8 CPU cores available on each machine (Figure 5.5). For identity operation the maximum throughput reached almost 2000 k flow/s and all the other operations performed on average at 1600 k flow/s. These results demonstrate that Spark is able to use the available CPU cores more efficiently if the computing is distributed across several nodes in cluster rather than when it is done on a single machine.

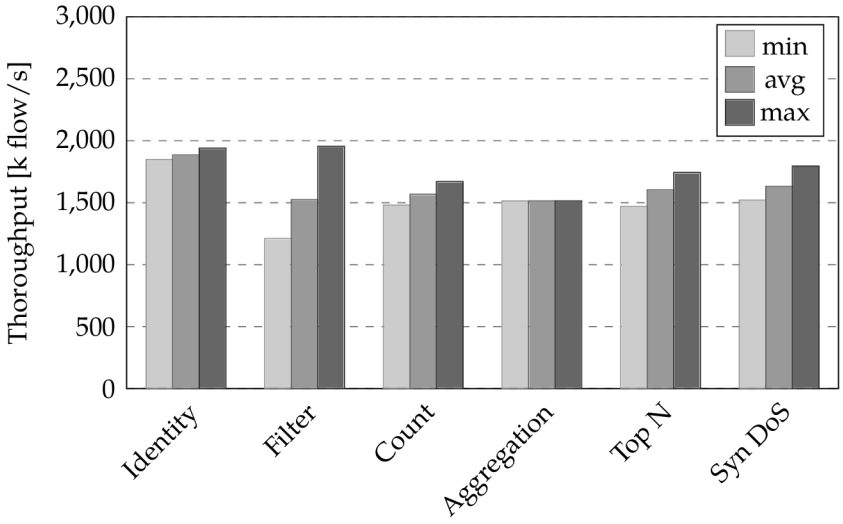

Figure 5.5: Spark benchmark on 4 *vm_medium* machines (32 CPU cores)

The fourth and last benchmark was performed on 4 *vm_small* nodes with decreased number of the total available cores to 16 (Figure 5.6).

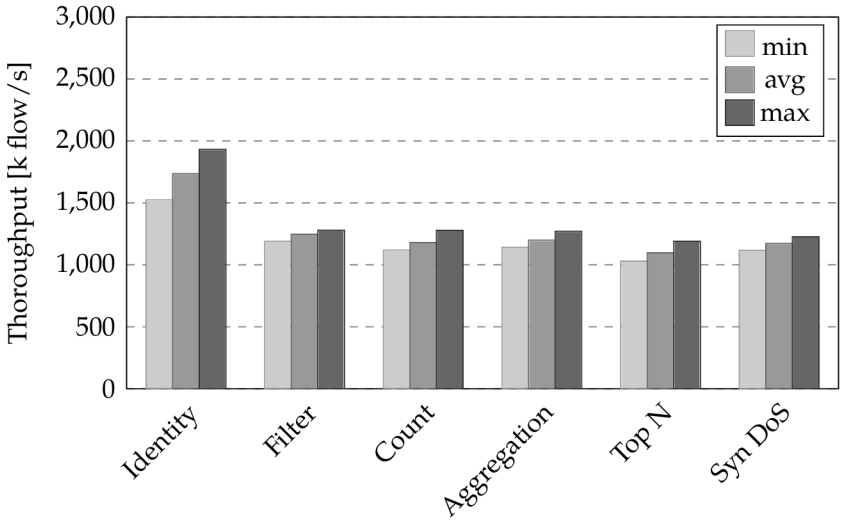

Figure 5.6: Spark benchmark on 4 *vm_small* machines (16 CPU cores)

Compared to the previous benchmark, the average base throughput has decreased only by 8%, however the throughput of every other

test was affected more significantly. *Count*, *Top N* and *Syn DoS* were most affected by the reduced amount of CPU cores, the results show a decrease of throughput for these operations by almost a third when compared to the previous benchmark.

5.5.1 Framework Comparison

Similarly to the Spark benchmark tool developed as part of this work other benchmarks with identical testing scenarios were developed to measure the performance of distributed flow processing in Apache Storm and Apache Samza on the *testbed* cluster. The development was coordinated to ensure that each benchmarking tool would implement the testing scenarios correctly for the results to be comparable. Because of different capabilities of each of the tested frameworks, the benchmark specification had to be adjusted several times during the implementation because of discovered limitations in some of the tested frameworks. All of the tested frameworks needed to be able to implement the benchmark. An example of one limitation discovered in Apache Spark during the development was the lack of support for count window of a set number of messages. It was also not possible to test Apache Spark on two cluster nodes as this leads to one master node delegating work to the other slave node with added latency from the network communication when compared to testing using only a single node.

The average results of each of the tested frameworks on single cluster node (the first two benchmarks) are shown in Figure 5.7, and the average results obtained from testing on four cluster nodes (third and fourth benchmark) are shown in Figure 5.8.

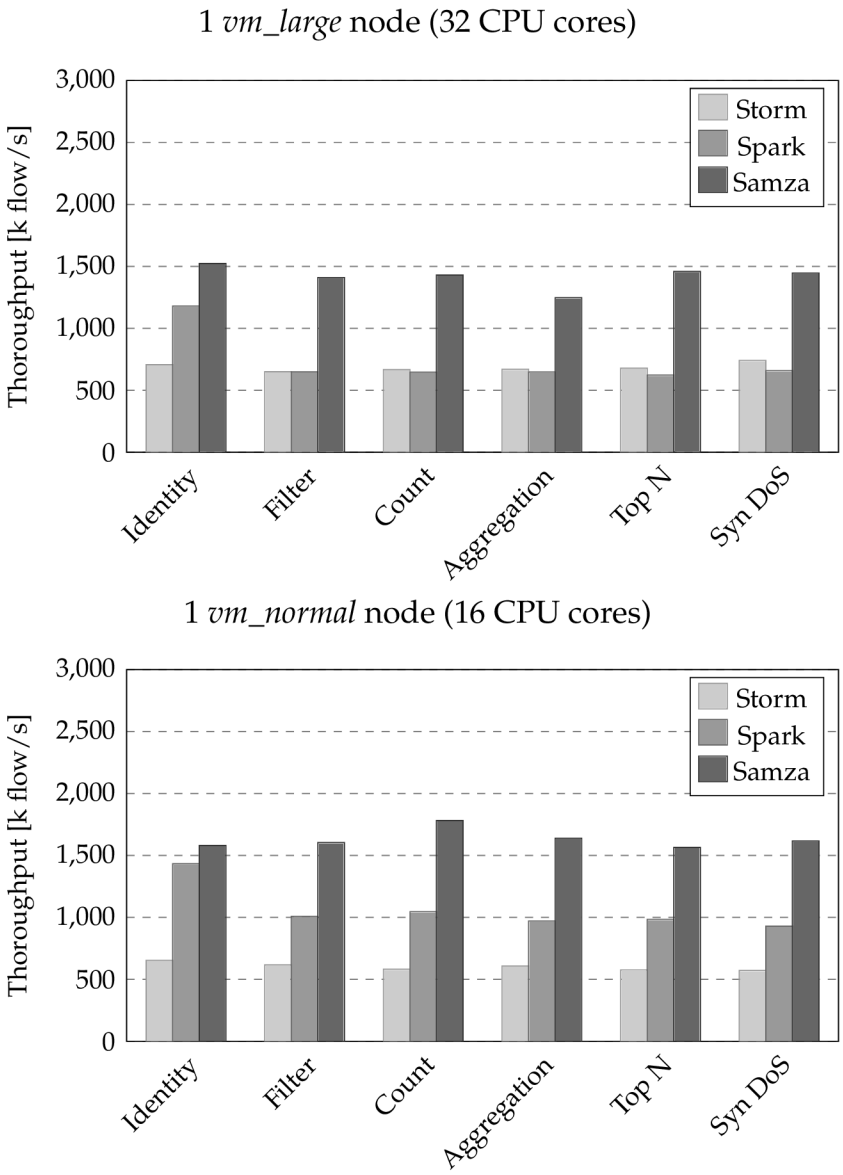

Figure 5.7: Performance benchmark using single node [1]

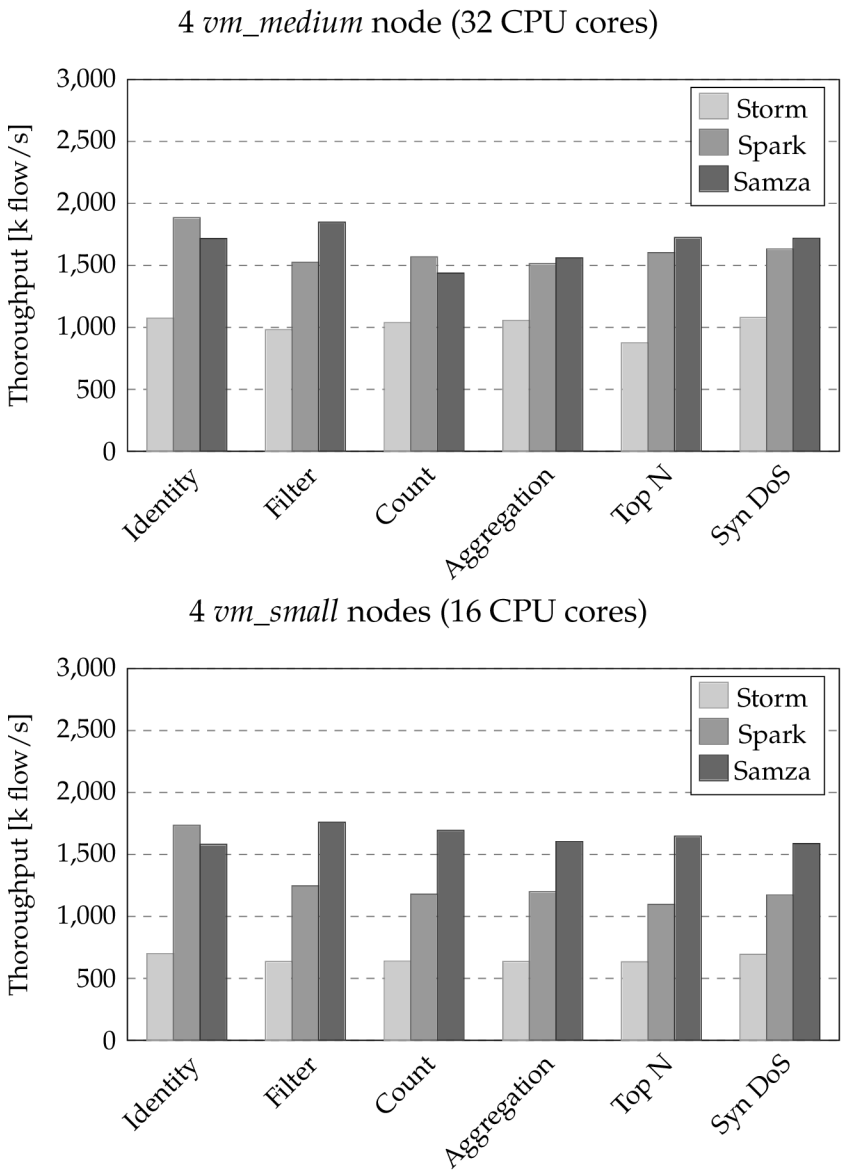

Figure 5.8: Performance benchmark using 4 cluster nodes [1]

Each tested systems has performed at least 500 k flow/s in every test and has fulfilled the minimal performance requirement set by the benchmark [1] of 300 k flow/s to handle both normal traffic and peaks that emerge during attacks and other anomalies. The results show that Spark and Samza perform better than Storm in almost every scenario and can therefore be utilized for simultaneous flow analysis of multiple networks.

The differences of each framework need to be taken into account when selecting the right distributed stream processing system for flow analysis. The micro-batching nature of Spark Streaming needs to be considered with its effect on latency which will vary depending on the used micro-batch interval. Storm should offer the best latency of the three frameworks, however the achieved performance in the benchmark does not make Storm suitable for data analysis of multiple networks. The performance of Samza could be significantly affected if Samza would not be used in combination with the Kafka streaming system because of its close integration.

The behavioral differences, cluster configuration options and active development of the tested technologies make it a complex problem to determine which system is more suitable. The benchmark to which this work contributed can be used to measure performance of structured data processing on any of the stream processing systems available.

5.6 Potential Improvements

The Spark benchmark was primarily implemented to perform the required functionality of every testing scenario, although some performance optimizations and adjustments were done during the benchmark development, such as enabling *Kryo* serialization library for the distribution of data to worker nodes and experimenting with different storage levels and micro-batch intervals. However, performance tuning was not the primary concern and there could potentially still be additional ways which could lead to further improving Spark's throughput with better utilization of the cluster resources.

The performance benchmark [1] was published in 2016, several technologies has since then emerged or matured which could improve

the benchmark implementation results if used, but were not available or mature enough at the time of its development.

It should be possible to improve the performance of Storm by updating its benchmark implementation to use Heron[3] which is a successor to Apache Storm, at the time of writing in beta stage of development, but considered stable enough for production use. Heron scales better, offers easier debugging and higher throughput for applications originally written for Apache Storm with no application code changes required.

Spark Structured Streaming (Page 24), introduced in Apache Spark 2.0, at the time of writing in alpha stage, could be a better choice than Spark Streaming for implementation of the Spark benchmark. Structured streaming would need to be tested to determine its effect on performance and unlike Heron it would require code changes to a different API. Additionally, because of its early stage of development, structured streaming might not yet be suitable for production use.

Apache Beam and Cloud Dataflow (Page 16) could simplify the benchmark implementation in the future to single application, because it would only be required to implement the testing scenarios once in the unified *Apache Beam SDK* and then the application could be executed via various runners to determine the performance of each processing engine. It brings to question whether all the capabilities required to implement the testing scenarios would be available in the *Beam API* and whether running the benchmark application written once in Beam would not significantly decrease the throughput of each tested engine when compared to native implementation using the particular streaming engine API.

With the write once and run on your selected processing engine approach, the resources required to implement the benchmark with Beam would reduce significantly and it would be ensured that the benchmark is implemented the same way for each framework. However, even at the time of writing, there is no Beam runner for Samza and Storm. Possibly, with the use of framework such as Beam, there would be less control over the underlying streaming application implementation which could lead to not achieving the throughput possible with development using the underlying engine's API.

3. `https://twitter.github.io/heron/`

6 Conclusion

The primary aim of this work was to implement benchmarking application in Apache Spark to measure its throughput on large volume of small structured messages based on real network data. The benchmarking application was implemented in Java using Apache Spark streaming module and shell scripts to orchestrate the benchmark deployment and execution. The benchmark is able to measure throughput on any number of cluster nodes using a set of operations based on real network analysis methods. The benchmark was further extended to be executable in local mode on single machine and to work with Apache Spark 2.1.0.

During a set period of time the benchmark was executed on a dedicated cluster of seven virtual machines along with similar benchmarking applications implemented in Apache Storm and Apache Samza. The obtained results have contributed to article *A Performance Benchmark of NetFlow Data Analysis on Distributed Stream Processing Systems* [1] which further elaborates on the collected results from all three frameworks. The collected results demonstrate that all three tested frameworks can be used to monitor network traffic. Spark and Samza which have significantly outperformed Storm can even be used to simultaneously monitor traffic from multiple networks.

The most actively developed distributed stream processing frameworks are presented in the theoretical part of this work along with other technologies relevant to the practical part. Chapter 5 describes the benchmark, the implemented application and proposes several suggestions which could in time further improve the benchmark performance or its maintainability.

To further extend upon this work, the benchmark could be rewritten to use Spark Structured Streaming which might eventually replace the Spark Streaming module. The application could also be updated to work with the latest version of Apache Kafka which has a different *consumer API*. Additionally it might be possible to further improve the application throughput via additional testing and performance tuning.

Bibliography

1. ČERMÁK, Milan; TOVARŇÁK, Daniel; LAŠTOVIČKA, Martin; ČELEDA, Pavel. *A Performance Benchmark of NetFlow Data Analysis on Distributed Stream Processing Systems*. In IEEE/IFIP Network Operations and Management Symposium 2016 (NOMS 2016). Istanbul, Turkey: IEEE Xplore Digital Library, 2016. ISBN 978-1-5090-0222-1.
2. *Flow Analysis Versus Packet Analysis. What Should You Choose?* [online]. NetFort, 2014 [visited on 2017-04-08]. Available from: `https://www.netfort.com/wp-content/uploads/PDF/WhitePapers/NetFlow-Vs-Packet-Analysis-What-Should-You-Choose.pdf`.
3. *NetFlow* [online]. Wikipedia, 2017 [visited on 2017-04-01]. Available from: `https://en.wikipedia.org/wiki/NetFlow`.
4. *Cisco IOS NetFlow* [online]. Cisco, 2017 [visited on 2017-04-01]. Available from: `http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html`.
5. KLEPPMANN, Martin. *Making Sense of Stream Processing* [online]. California: O'Reilly Media, 2016 [visited on 2017-03-19]. ISBN 978-1-491-93728-0. Available from: `http://www.oreilly.com/data/free/stream-processing.csp`.
6. BOLEN, Alison. *3 things you need to know about event stream processing* [online]. SAS Insights [visited on 2017-03-19]. Available from: `https://www.sas.com/en_us/insights/articles/big-data/3-things-about-event-stream-processing.html`.
7. *Apache Spark^{tm} - Lightning-Fast Cluster Computing* [online]. Apache Software Foundation, 2017 [visited on 2017-03-01]. Available from: `http://spark.apache.org`.
8. KARAU, Holden; KONWINSKI, Andy; WENDELL, Patrick; ZAHARIA, Matei. *Learning Spark: Lightning-Fast Big Data Analysis*. California: O'Reilly Media, 2015. ISBN 978-1-449-35862-4.
9. LASKOWSKI, Jacek. *Mastering Apache Spark 2* [online]. GitBook, 2017 [visited on 2017-03-12]. Available from: `https://www.gitbook.com/book/jaceklaskowski/mastering-apache-spark`.

10. LASKOWSKI, Jacek. *Spark Streaming* [online]. GitBook, 2017 [visited on 2017-03-12]. Available from: `https://www.gitbook.com/book/jaceklaskowski/spark-streaming`.

11. LASKOWSKI, Jacek. *Spark Structured Streaming* [online]. GitBook, 2017 [visited on 2017-03-12]. Available from: `https://www.gitbook.com/book/jaceklaskowski/spark-structured-streaming`.

12. ZAHARIA, Matei; DAS, Tathagata; ARMBRUST, Michael; XIN, Reynold. *Structured Streaming In Apache Spark: A new high-level API for streaming* [online]. The Databricks Blog, 2016 [visited on 2017-03-18]. Available from: `https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html`.

13. POINTER, Ian. *What Spark's Structured Streaming really means* [online]. InfoWorld, 2016 [visited on 2017-03-18]. Available from: `http://www.infoworld.com/article/3052924/analytics/what-sparks-structured-streaming-really-means.html`.

14. ZAPLETAL, Petr. *Introduction Into Distributed Real-Time Stream Processing* [online]. Cake Solutions, 2015 [visited on 2017-03-19]. Available from: `http://www.cakesolutions.net/teamblogs/introduction-into-distributed-real-time-stream-processing`.

15. ZAPLETAL, Petr. *Comparison of Apache Stream Processing Frameworks: Part 1* [online]. Cake Solutions, 2016 [visited on 2017-03-25]. Available from: `http://www.cakesolutions.net/teamblogs/comparison-of-apache-stream-processing-frameworks-part-1`.

16. ZAPLETAL, Petr. *Comparison of Apache Stream Processing Frameworks: Part 2* [online]. Cake Solutions, 2016 [visited on 2017-03-25]. Available from: `http://www.cakesolutions.net/teamblogs/comparison-of-apache-stream-processing-frameworks-part-2`.

17. ELLINGWOOD, Justin. *Hadoop, Storm, Samza, Spark, and Flink: Big Data Frameworks Compared* [online]. DigitalOcean, 2016 [visited on 2017-03-25]. Available from: `https://www.digitalocean.com/community/tutorials/hadoop-storm-samza-spark-and-flink-big-data-frameworks-compared`.

18. MSV, Janakiram. *All the Apache Streaming Projects: An Exploratory Guide* [online]. The New Stack, 2016 [visited on 2017-04-01]. Available from: https://thenewstack.io/apache-streaming-projects-exploratory-guide/.

19. DESAI, Ankur. *Real-Time Event Streaming: What Are Your Options?* [online]. MapR, 2016 [visited on 2017-03-19]. Available from: https://mapr.com/blog/real-time-event-streaming-what-are-your-options/.

20. KREPS, Jay. *Benchmarking Apache Kafka: 2 Million Writes Per Second* [online]. LinkedIn Engineering, 2014 [visited on 2017-03-25]. Available from: https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines.

21. *Apache Kafka* [online]. Apache Software Foundation, 2017 [visited on 2017-03-25]. Available from: https://kafka.apache.org.

22. *Apache Storm* [online]. Apache Software Foundation, 2017 [visited on 2017-03-25]. Available from: http://storm.apache.org.

23. *Apache Samza* [online]. Apache Software Foundation, 2017 [visited on 2017-03-25]. Available from: http://samza.apache.org.

24. *Apache Flink: Scalable Stream and Batch Data Processing* [online]. Apache Software Foundation, 2017 [visited on 2017-03-25]. Available from: https://flink.apache.org.

25. *Apache Apex* [online]. Apache Software Foundation, 2017 [visited on 2017-04-01]. Available from: https://apex.apache.org.

26. *Apache Beam* [online]. Apache Software Foundation, 2017 [visited on 2017-04-01]. Available from: https://beam.apache.org.

27. PERRY, Frances; MALONE, James. *Dataflow and open source - proposal to join the Apache Incubator* [online]. Google Cloud Platform Blog, 2016 [visited on 2017-04-01]. Available from: https://cloudplatform.googleblog.com/2016/01/Dataflow-and-open-source-proposal-to-join-the-Apache-Incubator.html.

28. PERRY, Frances. *Apache Beam graduates from incubation: Try it today on Google Cloud Dataflow* [online]. Google Cloud Big Data and Machine Learning Blog, 2017 [visited on 2017-04-01]. Available from: `https://cloud.google.com/blog/big-data/2017/01/apache-beam-graduates-from-incubation-try-it-today-on-google-cloud-dataflow`.

29. *The CAIDA Anonymized Internet Traces 2015 Dataset* [online]. CAIDA: Center for Applied Internet Data Analysis, 2015 [visited on 2017-04-01]. Available from: `http://www.caida.org/data/passive/passive_2015_dataset.xml`.

A Attachments

The code of the implemented Spark benchmark and Kafka producer used during the benchmark development is available and hosted on GitHub

- **Spark benchmark:** `https://github.com/securitycloud/spark` There are currently three branches in the repository:
  - **master:** contains the latest functionality prepared to run on the testing (development) cluster
  - **cluster-tests (deprecated):** branch used during the final testing period configured to run on *testbed* cluster
  - **localhost-tests:** copy of the latest master branch which can be used to execute the benchmark in local mode with the use of customized script *bin/all-tests-localhost.sh*
- **Kafka producer:** `https://github.com/securitycloud/kafka/tree/master/kafka-filip`

The source code can also be found in the Information System of Masaryk University dissertations archive

- `https://is.muni.cz/auth/th/374137/fi_m/`