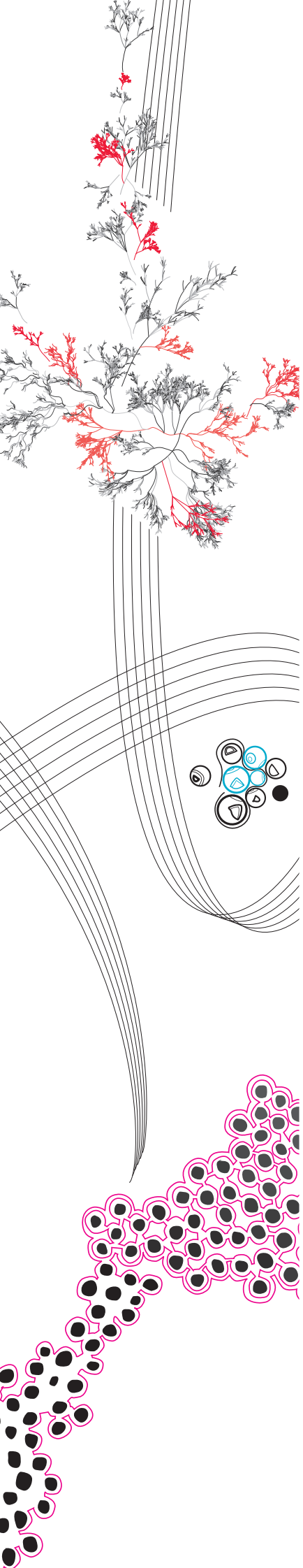


July, 2017

MASTER THESIS

Orchestrating Similar Stream Processing Jobs to Merge Equivalent Subjobs

B. van den Brink

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Database Systems

Exam committee:
Dr.ir. M. van Keulen
F.C. Mignet, MSc (Thales Research & Technology)
Dr.ir. R.B.N. Aly
Prof.dr. M.Huisman

UNIVERSITY OF TWENTE.

ABSTRACT

The power and efficiency of distributed stream processing frameworks have greatly improved. However, optimizations mostly focus on improving the efficiency of single jobs or improving the load balance of the task scheduler.

In this thesis, we propose an approach for merging equivalent subjobs between streaming jobs at runtime, that are generated from a predefined template. Since our template structure is similar to the structure of simple Spark Streaming applications, templates can be created with minimal development overhead.

Furthermore, we have analyzed the complexity of benchmarking Spark Streaming applications. Based on the results of this analysis, we have designed a method to benchmark Spark Streaming applications with the maximum throughput as metric.

This method is applied on performing an experimental analysis of the performance of merged jobs versus unmerged jobs on the CTIT cluster of University of Twente. Based on the results of this analysis however, we cannot conclude for which cases job merging results in an increase of the maximum throughput.

ACKNOWLEDGEMENTS

Firstly, I wish to express my gratitude to my main supervisor, dr.ir. M. van Keulen, for his critical and constructive feedback. Especially during the last stages of the project, his feedback and assistance proved invaluable for achieving this thesis.

I would like to thank F.C. Mignet, MSc for the large number of discussions at Thales. These discussions greatly helped me in shaping the concepts behind this thesis.

I would like to thank dr.ir. R.B.N. Aly for the numerous meetings we had. His technical expertise was of great value in working out the technical details of this project. Furthermore, I would like to thank him for preventing me from making PhD sized project on this topic.

I would like to thank prof.dr. M.Huisman for participating in the graduation committee and providing feedback during the last stages of this project.

I am indebted to all my family and friends for their persistent support during this project. Especially, I would like to express my deepest gratitude to my parents for supporting me in every non-technical aspect they could during the whole master curriculum.

Contents

CHAPTER 1

INTRODUCTION

1.1 Motivation

Big data technologies have taken big leaps in accessibility, scalability and performance. In 2004, Google wrote a paper about MapReduce, “a programming model and an associated implementation for processing and generating large data sets.” [1] This allows creating powerful distributed applications by only implementing two functions: a map and a reduce function. While current cluster-computing frameworks are more powerful and efficient, many are based on the concept of implementing functions that are naturally suitable for distributed processing.

An area where big data analysis can be of great relevance is the governmental section. Pavlin, Quillinan, Mignet, *et al.* [2] describe the challenges and opportunities of using big data for national security by discovering, analyzing and assessing possible criminal activities. This can be accomplished by using traces of activities from various sources captured in various large data sets, owned by national security agencies. The Dynamic Process Integration Framework (DPIF) [3] can be used to integrate processes that capture information from these sources and provide feedback to security agencies.

Ideally, tasks created by DPIF are converted to big data jobs in real-time and should connect to jobs that are already running to reuse information to optimize the system as a whole. However, as the authors state, traditional big data solutions are optimized towards distributing static computational tasks on a partitioned and distributed data set.

Furthermore, the tasks created by a specific DPIF process can share complex components. In section 1.2.1, we describe and example scenario about monitoring tweets in specific regions. Generating a big data job for each tasks is therefore inefficient, since the same computation is executed multiple times. Automatic merging of these components potentially can improve the performance drastically.

1.2 Problem exploration

In this section, we explore the problem of this thesis in more dept. Our first step is to provide a concrete scenario to further designate the problem. Subsequently, big data frameworks are reviewed to determine to which extend the problem is solved. In the following section, several improvements are considered. Finally, section 1.2.4 draws conclusions based on the reviews.

1.2.1 NER scenario

We illustrate this with a more concrete scenario that covers crisis management. This scenario will be used throughout the thesis. Crises get much attention on social media. This information is potentially very useful for security agencies. Terpstra, De Vries, Stronkman, *et al.* [4] describe how real-time Twitter analysis can be used for operational crisis management.

In our scenario, the security agency must monitor certain regions for crises. Therefore, it filters tweets covering these regions. However, only an insignificant percentage of the tweets

is annotated with coordinates of the location where the tweet is sent. Furthermore, in this case, a security agency is more interested in the location mentioned in the tweet, which is not necessary the send location.

To extract the location mentioned in a tweet, natural language processing (NLP) is needed. More specifically, a technique called named-entity recognition (NER). Lingad, Karimi, and Yin [5] researched a method for applying NER for this purpose.

After extracting the location of tweets, these tweets can be filtered for the regions that have to be monitored by the agency. This agency can then use classifying techniques [6] to recognize the type of the disaster.

Since in our practical example, it is not known on beforehand which regions are to be monitored, spawning a new real-time stream processing big data job for every request to monitor a specific region is necessary. Furthermore, since these jobs are only executed during crises, which are rare. Hence, continuously running the NER component, storing every result and use it when necessary is inefficient.

1.2.2 Big data frameworks

In this section, we review several big data frameworks. We start with MapReduce, which has had a high impact on the design of more recent frameworks, which are also reviewed in this section.

While MapReduce [1] offers a simple abstracted programming model for creating batch processing cluster jobs, it restricts the developer to map and reduce functions. Furthermore, the performance is limited, since it uses a distributed file system, in original implementation the Google File System [7], to store the results of the map phase and reduce phase.

Resilient Distributed Datasets (RDDs) resolve these limitations by providing “a distributed memory abstraction that lets programmers perform in-memory computations on large cluster” [8], while maintaining fault-tolerance through lineage. Spark is a cluster-computing framework that implements this abstraction. Spark has a rich API with a much richer set of functions, compared to MapReduce.

Discretized streams (D-Streams) is a processing model to process streams in batches [9]. This processing model is implemented on top of Spark Streaming. Batches are transformed to RDDs and executed as regular Spark jobs.

Spark SQL is a module for Spark that provides a new API, the DataFrame API, and enables executing SQL queries on datasets using Spark. This allows for combing existing Spark logic with the flexibility of using SQL queries on data. The catalyst optimizer is used to compute an efficient query plan for executing the SQL query. However, queries are only optimized within a job and Spark SQL does not allow for automatic reuse of intermediate results across jobs. Spark SQL can also be used in combination with Spark Streaming.

Storm is a real-time stream data processing system that is based on topologies. A topology is a directed acyclic graph (DAG) of bolts and spouts. A spout is inputs data in the topology. Bolts are processing elements that execute a computation on its input. Topologies are fixed at compile time. Therefore, it is not possible to add bolts dynamically and therefore reuse intermediate results in new tasks automatically.

1.2.3 Stream processing enhancements

In this section, we describe several extensions and optimizations for Storm and Spark Streaming that focus on adaptive processing.

The default Storm scheduler is a pseudo-random round robin task scheduler. Several schedulers are designed and researched to improve the performance of Storm [10] [11] [12] [13]

[14]. The main principle of these schedulers is similar: continuously analyzing the traffic and the load of Storm and adapt the scheduling of tasks in real-time according to these metrics to reduce the traffic load and optimize the workload balance.

While these researches show that the performance of Storm improves drastically by implementing these adaptive schedulers, the schedulers do not allow for merging topologies of different jobs or allow for modifying the topology of a job at runtime.

Additionally, a new scheduler for Spark Streaming has been proposed by Liao, Gao, Ji, *et al.* [15]. Since Spark Streaming is a micro-batch streaming processing framework, the latency is higher with Spark Streaming compared to real-time stream processing frameworks as Storm. This becomes worse with an unstable input, since the size of a batch is fixed at startup-time. The new scheduler allows for dynamically adjusting the batch interval size to minimize this latency.

In a recent research (May, 2017), Ren and Curé [16] propose a hybrid adaptive distributed RDF Stream Processing engine that enables continuous SPARKQL query evaluation, based on Spark Streaming and Kafka. This allows for adding components to existing queries instead of creating a new query and running it separately. However, jobs are limited to the expressiveness of the SPARKQL language.

1.2.4 Conclusions

In this section, we have reviewed big data technologies. Big data technologies have improved significantly in the past years in both performance terms and expressiveness. However, jobs are fixed at compile-time and the schedulers of these systems do not allow for merging equivalent components between these jobs.

Furthermore, we have examined several extensions and optimizations for both Storm and Spark Streaming. The listed adaptive schedulers for Storm optimize the scheduler and load balance according to runtime performance metrics. While this optimizes the system that is running the jobs as a whole, it does not take equivalent tasks into account.

Strider is an interesting approach, of which the paper is released during the writing of this thesis. This engine allows for continuous query evaluation. However, this engine in mainly designed for Internet of things applications processing sensor data. Furthermore, tasks are limited to the expressiveness of the SPARKQL language.

Concluding, without inter-job merging, the NER operation of the scenario in section 1.2.1 would be applied each tweet for each separate job. A high number of simultaneous jobs therefore has the consequence of a high amount of wasted resources.

1.3 Problem statement

Based on the previous section, we can conclude that similar job merging is not applied in current big data frameworks and its enhancements and that similar job merging can potentially result in a great optimization of computing resources.

In section 1.3.1, we elaborate on the problem of similar job merging and define the challenges in accomplishing this.

Secondly, a benchmarking method is needed to validate the job merger. In section 1.3.2, we elaborate on the problem of benchmarking jobs based on Spark Streaming, the framework we have chosen as the basis for our merging approach (refer to section 1.4).

1.3.1 Job merging

There are various challenges in designing a job merging approach. Referring to the NER scenario in section 1.2.1, job merging has to be accomplished at runtime, on-demand and without having information about upcoming jobs in the future.

The first challenge is the detection of equivalent tasks. This implies we have to determine function equality. Appel describes three kinds of function equality [17]. The author states that functional equality can only properly be determined if the expression itself is equal, or if the to be compared functions are references to the same expression.

However, there is another factor that is crucial in finding equivalent tasks. Not only the function of the task must equal, the input of the task must equal too. Otherwise, outputs of tasks are different with the consequence that the task cannot be merged. Concluding, determining equivalent tasks between two unrelated jobs is infeasible.

Furthermore, merging itself is a challenge. When a job is started, all job information is sent to the workers that are involved, which could be several hundreds. Modifying a running job would therefore require updating each of these workers and coordinate them according to the new job. An approach could be generating a new updated job and replacing the running job. However, this can cause large delays in the processing of records. Therefore, a merging approach has to be designed that does not touch currently running jobs.

1.3.2 Benchmarking

A suitable benchmarking method is needed for validating the job merger. The job merger, which is proposed in section 1.4, merges Spark Streaming jobs. In section 1.7, we describe the research method for using a Spark Streaming benchmarking method for validating the job merger.

In the original paper of Spark Streaming, Zaharia, Das, Li, *et al.* describe that they have benchmarked the Spark Streaming framework with the maximum throughput as metric [9]. This is defined as *the maximum bytes per second while keeping the end-to-end-latency below a given target*. Since Spark Streaming is a micro-batch processing framework, the end-to-end-latency can be fixed by setting the size in seconds of the batch interval.

However, important questions about their benchmarking method are left unanswered for the reader. The first question is: what is used as input? Spark Streaming reads from a stream from a particular source. For example, Kafka, or a distributed file that is read as a stream. And in the case Kafka is used, how many brokers are used? This impacts the performance of the benchmark and is therefore important to know.

Furthermore, how is the input rate regulated? As section 2.1 explains, streams are split and processed in batches. Using an input rate that is too high results in overflowing these batches and causes an unstable situation. This is further explained in section 4.1.

Furthermore, even if measures are taken to ensure that the application is running stable, it is important to verify this. Therefore, the following question arise: how can be confirmed that the applications for the benchmark are running stable?

Additionally, authors do not describe how metrics are collected. A possibility is using the built-in metrics system, described in section 2.1.3. However, this system has to be configured by the user. This impact the validity of the results and is therefore important to be described.

Finally, the paper does not explain how the throughput is calculated from these metrics. Therefore, the reader cannot validate this calculation.

Hence, we need to answer these questions in order to provide and use a reliable method for validating our job merger.

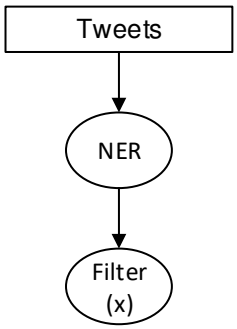
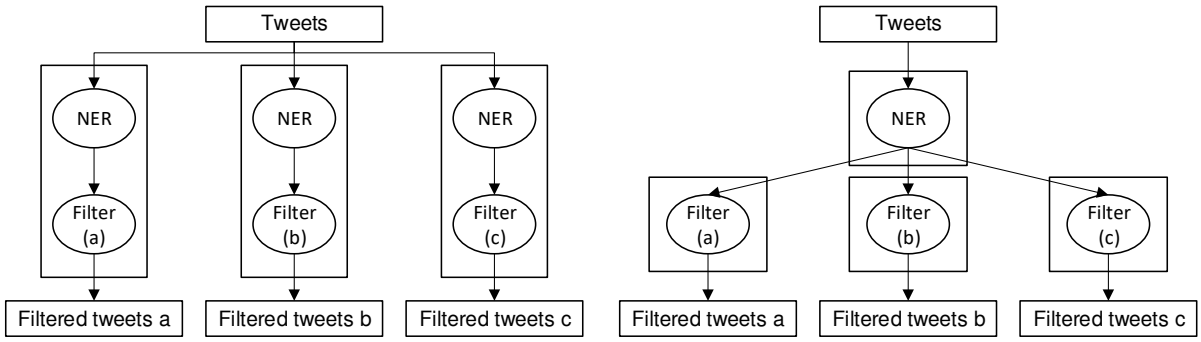


Figure 1.1: NER job template



(a) Unmerged jobs

(b) Merged jobs

Figure 1.2: From unmerged NER jobs to merged NER jobs

1.4 Proposed approach

In this section, we propose an approach for the problems stated in the previous section. In section 1.4.1, we propose an approach for merging Spark Streaming jobs, based on templates. In Section 1.4.2, we propose a benchmarking method.

1.4.1 Template-based job merging

In section 1.3.1, we explained that merging jobs submitted independently from each other, without prior information about the tasks of this job is infeasible because of the problem of function equality and the possible differences in input.

Therefore, we propose a job merging approach that is based on templates. Reconsider the NER scenario in section 1.2.1. The only difference between the concrete jobs of this scenario is the location for which the tweets are filtered. Therefore, we can construct a template as illustrated by fig. 1.1. Here, the filter condition is parameterized with *x*.

By applying a concrete parameter, a concrete job can be constructed. Since the tasks that are different between jobs, in the case the filter, are defined in the template, we can derive the tasks that can be merged from the combination of the template and the lists of applied arguments. In this way, the problems of function equality and possible differences in input are circumvented.

Furthermore, section 1.3.1 stated that merging itself is a challenge too, because modifying running jobs is difficult and restarting a job introduces a large delay. Therefore, our approach splits the job into smaller subjobs.

Figure 1.2 illustrates this conceptual idea. Each box represents a Spark Streaming job on the cluster. Figure 1.2a shows three instances of the template illustrated by fig. 1.1, with parameters *a*, *b* and *c*. Figure 1.2b shows the merged variant of these jobs. The number of tasks is reduced from 6 to 4. While the number of Spark Streaming jobs is increased from 3

to 4, the heavy NER task only has to be executed once for each tweet, which potentially has a big performance gain.

1.4.2 Benchmarking method

In section 1.3.2, we showed that the original Spark Streaming paper [9] has omitted several important aspects concerning Spark Streaming's notions of maximum throughput and micro-batch stabilization.

In section 4.1, we have further analyzed these aspects. The built-in backpressure algorithm of Spark Streaming is analyzed as a possible candidate to reach batch stabilization. This is shown by section 4.2.1. Based on this analysis, we propose a general benchmarking method for Spark Streaming applications.

1.5 Contribution

Our contribution consists of two parts. The main contribution is providing an approach for merging Spark Streaming jobs at runtime and validating this approach by benchmarking merged and unmerged jobs and comparing the results.

The second contribution is providing a method for benchmarking Spark Streaming jobs with the maximum throughput as metric. We contribute by explicitly addressing aspects missing in the evaluation of Spark Streaming described by Zaharia, Das, Li, *et al.*

1.6 Research and design questions

This section states the research and design questions that arise from the problem statement in section 1.3.

In section 1.4.1, we proposed a template-based job merging approach. Using this approach, one of the questions that is answered by this thesis is the following.

DQ1 How can similar stream processing jobs be orchestrated in order to merge equivalent subtasks?

To answer this question, we have split this question in several subquestions. Templates have to be defined by the developer. Our aim is to impose a minimum development overhead for creating job templates or converting existing jobs to our template structure. Therefore, the following question arises.

SQ1 How can job templates be defined with minimal development overhead?

Furthermore, as section 1.4.1 explains, jobs generated from templates are split into subjobs, of which the results can be reused by multiple job instances. A trivial method for splitting jobs is creating a subjob for every task. However, not all tasks are parameterized, which makes this method inefficient.

SQ2 How can we automatically group tasks to reduce the merging overhead while maintaining the ability to merge subjobs?

Results of subjobs must be shared with the next subjob of the task chain. However, this is not trivial, since jobs are intended to run distributed on a cluster.

SQ3 How can subjobs share intermediate results in a distributed environment?

As section 1.3.2 explains that a new benchmarking method is needed in order to validate the job merger. This leads to the second design question.

DQ2 How can Spark Streaming jobs be benchmarked with the maximum throughput as metric?

We want to further analyze the challenges in benchmarking Spark Streaming applications.

SQ5 What are the challenges of throughput benchmarking Spark Streaming applications?

Backpressure is potentially a technique to resolve some of the challenges. Therefore, we want to analyze how this can be incorporated in our benchmarking method.

SQ6 How can Spark Streaming's backpressure be used to stabilize applications during benchmarks?

Furthermore, metrics have to be collected and used to calculate the maximum throughput.

SQ7 How can application metrics be collected and used for calculating the maximum throughput?

Sharing similar subjobs involves an overhead, since there is a need for a sharing layer. Therefore, sharing similar subjobs is not beneficial if the overhead is bigger than the performance gain by deduplicating the subtasks. This results in the following research question.

RQ How well does runtime stream process orchestration with equivalent subjob merging perform compared to naïve streaming job scheduling?

1.7 Research method

This section describes the research method that is used for answering our research question, as defined in section 1.6. To answer this question, we have synthesized the NER scenario described in section 5.1 to a general template with a heavy and light task. This allows for more stable experiments and the use of synthesized data. Refer to section 5.1 for further explanation.

The following variables are used in our experiment.

Fan-out
Specifies the number of tasks that are merged and

Executors per job
The degree of parallelization on the cluster.

The metric we are interested in is the maximum throughput, which is a common metric used in literature for benchmarking stream processing applications. It is for example used to evaluate Spark Streaming and compare this framework to similar stream processing frameworks [9].

For each combination of variables used in the experiments, merged and unmerged variants of the jobs are executed on CTIT cluster of University of Twente. The results are interpreted to find a relation between the variables and the performance gain in merging.

1.8 Thesis structure

In chapter 2, we describe the general background needed to understand the subsequent chapters. This includes information about Spark, Spark Streaming, which is based on Spark and Kafka, which is used by the job merger and the experiments.

Chapter 3 describes the job merging solution.

Chapter 4 describes the challenges in benchmarking Spark Streaming and the solutions to these challenges incorporated in a benchmarking method.

Chapter 5 describes how the benchmarking method is applied to perform an experimental analysis on the job merger. Furthermore, this chapter shows and describes the results.

Chapter 6 draws the conclusions of this research.

Finally, chapter 7 discusses the limitations of this research and suggests future work.

CHAPTER 2

BACKGROUND KNOWLEDGE

In this chapter, we provide background knowledge for understanding the successive chapters.

2.1 Spark streaming

Spark Streaming is a high-throughput streaming framework, based on the Spark cluster computation framework. This framework is used in our job merging approach, described in chapter 3. In this section, we explain the components of Spark Streaming that are necessary to understand the succeeding chapters.

2.1.1 Discretized streams

Discretized streams is the processing model that is used in Spark Streaming [9]. In this processing model, streams are processed in intervals, which are set by the developer. During an interval, data is collected by the input receiver and temporarily stored in-memory.

After an interval, the dataset is converted to an Resilient Distributed Dataset (RDD) [8]. RDD is the main data structure and processing model of Spark, for distributed in-memory processing of large datasets. This concept is illustrated by fig. 2.1. The interval size here is 1 second.

The RDD processing model is used to process the data by converting the defined D-Stream transformations to RDD operations. This allows for fault-tolerant micro-batch processing of streams.

D-Stream API

D-Streams in Spark Streaming have a functional API, implemented in Scala. Each Spark Streaming application has the following components.

Input D-Stream
: Receives data from a specific source. For example, a Kafka (section 2.2) topic or a periodic read from a file on HDFS (Hadoop Distributed File System).

Transformations
: A transformation creates a new D-Stream by applying a specific operation on the, one or more, parent D-Streams.

Output operations
: For example, writing to a Kafka topic, a HDFS file or a Cassandra database [18].

Word count example

Consider the word count example in listing 2.1. This example is reused in section 3.2.1. Irrelevant configuration details are replaced with `<placeholders>`.

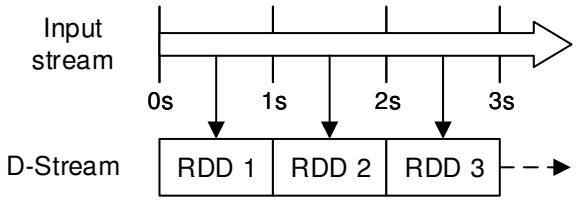


Figure 2.1: Discretizing an input stream

```
1  val ssc = new StreamingContext(<sparkContext>, Seconds(1))

3  KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder](
     ssc,
5    <kafkaParams>,
     Set[String]("word-count") // input topic
7  )
     .filter(_._2.toLowerCase.contains("twente"))
9    .flatMap(_._2.split(" "))
     .map(x => (x, 1L))
11   .reduceByKey(_ + _)
     .filter { case (_, count) => count > 1 & count < 5 }
13   .writeToKafka(<kafkaConfig>, s => new ProducerRecord[String, (String, Long)]("
         result", s) // outputTopic
     )
15
   ssc.start()
17 ssc.awaitTermination()
```

Listing 2.1: Word count example

We first give a general explanation of this example. The application reads tweets, filters these tweets for a specific string. Then, each tweet split into individual words. These words are grouped and converted to `(word, count)` pairs, where count is occurrence of this word for each filtered tweet in the current micro-batch (refer to section 2.1.1). The last step is filtering the words that have a count in a predefined range.

Next, we explain per line of code what this implementation does.

Line 1 `StreamingContext` initialization

Lines 3-7 Input D-Stream reading from Kafka topic *word-count*

Line 8 Filtering tweets that contain the string *twente*.

Line 9 Splitting the tweet into words.

Line 10 Creating `(word, 1)` pairs.

Line 11 Group and reduce the pairs to `(word, count)`.

Lines 12 Filter words using condition with count in $(1, 5)$

Lines 13, 14 Write to Kafka topic *result*

Line 17 Start the application

The code from listing 2.1 translates to the D-Stream graph shown in fig. 2.2. In the Spark Streaming implementation, D-Streams and the corresponding D-Stream graph are immutable.

2.1.2 System architecture

Figure 2.3 illustrates the most important components of the system architecture of Spark Streaming. The components are as follows.

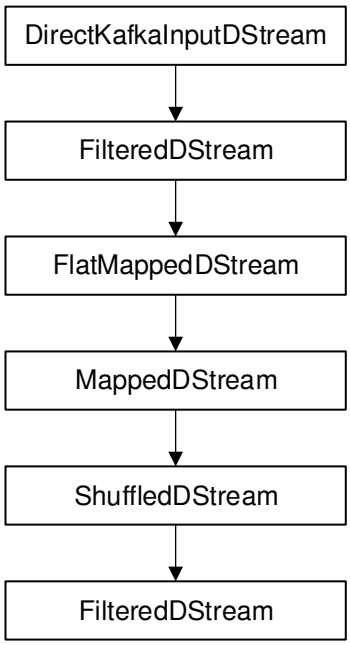


Figure 2.2: D-Stream graph of word count example

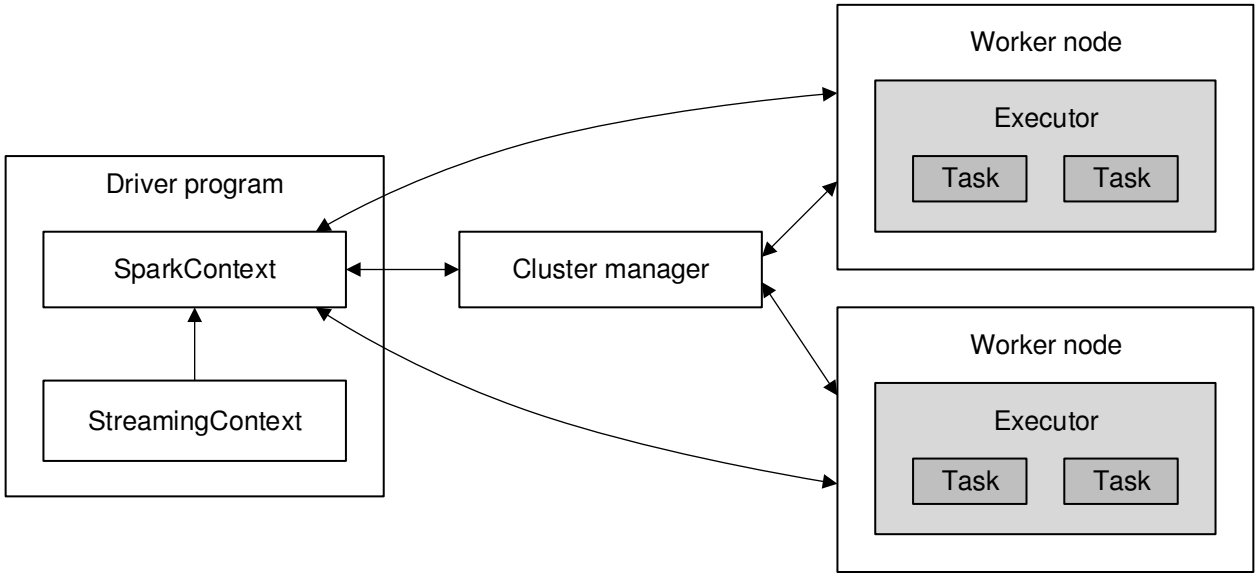


Figure 2.3: Spark cluster architecture, extended from [19]

Driver program
The JVM process that hosts the `StreamingContext` and the `SparkContext`.

StreamingContext
The entry point for all Spark Streaming functionality. This object contains the D-Streams and a reference to the SparkContext.

SparkContext
The entry point for all Spark functionality. This is used by Spark Streaming to execute jobs for each batch.

Cluster manager
Manages the resources of the cluster. The cluster manager assigns worker nodes to executors.

Worker node
A node on the cluster that runs application code.

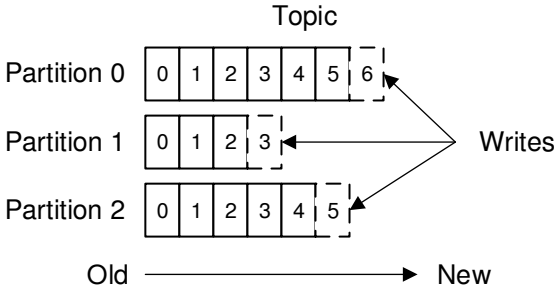


Figure 2.4: Topic partitioning, figure based on [21]

Executor
A process launched for an application on a worker node that executes tasks.

Task
A unit of work sent to one executor.

2.1.3 Metrics system

Spark has a built-in metrics system that provides information about the Spark application that is running. This system is used in our research for benchmarking Spark Streaming applications. In this section, we describe the components of this system that are used in our research.

The metrics system allows users to register sinks to which the metrics are reported to. In our research, we used the `CsvSink`, that writes metrics to a number of CSV files.

Using the `CsvSink`, metrics are written once per polling period. This polling period can be defined by the user. Every period, the current value for each metric is appended to the CSV files, with for each metric a different file.

2.2 Kafka

Kafka is a distributed messaging system for log processing that is scalable and has a high throughput [20]. In Kafka, messages are grouped into topics. Each topic is a stream of messages of the same type. Producers can write messages to such topic.

Topic messages are stored at a set of servers, called brokers. Consumers can subscribe to one or more topics from these brokers to receive these messages. By partitioning topics to brokers, load can be balanced across servers. This concept is illustrated by fig. 2.4.

Messages in kafka consists of an array of bytes. This has as advantage that everything can be used as message, as long as the message limit, which is configurable, is not reached. However, serializers and deserializers are needed to process messages. For the coordination of brokers and consumers, Zookeeper [22] is used.

CHAPTER 3

JOB MERGING

In this chapter, we describe our solution to the job merging problem introduced in section 1.3.1. The objective is to answer ***DQ1**: How can similar stream processing jobs be orchestrated in order to merge equivalent subtasks?*

The description of our job merging solution is divided in two sections. In section 3.1, we explain the concept of template-based merging that is applied in our job merging solution. In the second section, section 3.2, we explain how this concept is applied to merging jobs with Spark Streaming.

3.1 Template-based job merging

In the ideal situation, developers can submit their stream processing job to a cluster, and the resource manager of the cluster automatically analyzes the code of the job is analyzed and compared to running jobs for common tasks. The resource manager of the cluster, to which jobs are submitted, then distillates these tasks and executes these as a single subjob to prevent executing the task twice.

However, as explained in the problem statement, this type of code analysis is extremely hard, because this involves extensional functional equality [17]. Therefore, we propose template-based job merging. Instead of merging arbitrarily jobs, we ask the developer to define jobs according to a predefined structure, called a template.

3.1.1 Job template

Templates contain parameterized tasks. Instantiating a template can be accomplished by applying concrete parameters to a template. Using this approach, the job merger can use this information to determine which tasks are equal and can be merged. In this way, we circumvent the problem of complex code analysis.

A template is composed of two components: the input source and a chain of tasks, which can be parameterized. Each task applies an operation on the input, or the result of the previous task. In our NER scenario, the template would contain two tasks: named-entity recognition and a parameterized filter for the location.

Figure 3.1 illustrates the concept of defining and instantiating a template. Figure 3.1a shows three operations: a, b and c with the corresponding parameters x_1, x_2 and x_3. By applying $x = (1, 2, 3)$ to the template, the job in fig. 3.1b is generated.

3.1.2 Merging jobs

We explain template-based job merging by using the example illustrated by fig. 3.1. The template consists of three tasks: a, b and c. The concrete job generated from the template consists of the instantiations of these tasks: $a(1)$, $b(2)$ and $c(3)$. These are executed as separate subjobs. The purpose is to reuse these subjobs when new jobs are submitted with equivalent subjobs.

(a) Template structure

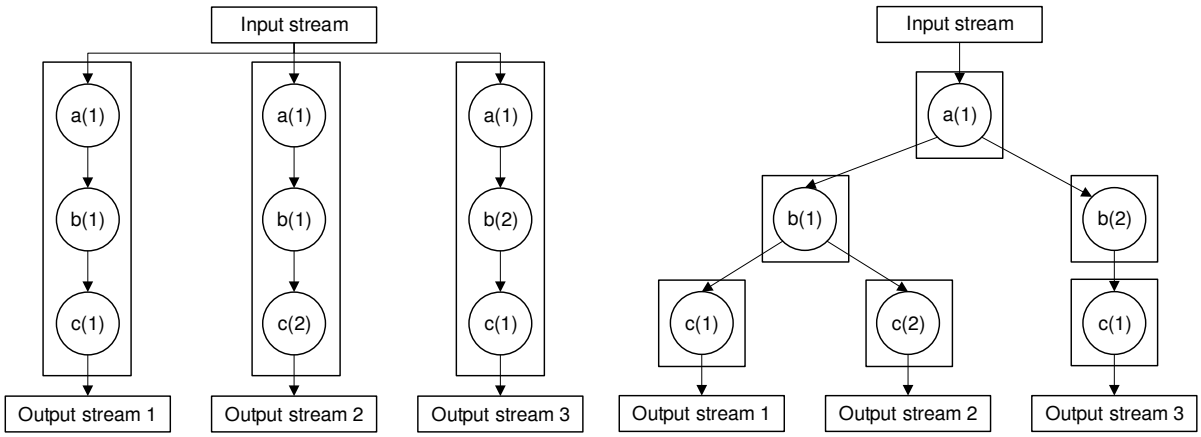
(b) Example job with parameters $x = (1, 2, 3)$

Figure 3.1: From template to job

Therefore, the following conditions must be met.

- the inputs of a subjob must be equivalent;
- the task executed on the input must be equivalent.

If we combine these to conditions, we can conclude that the largest common prefix of parameters defines the subjobs that can be merged.



(a) Unmerged jobs

(b) Merged jobs

Figure 3.2: Three merged jobs, with parameters $(1, 1, 1)$, $(1, 1, 2)$ and $(1, 2, 1)$

Figure 3.2 illustrates this principle of prefix matching. Figure 3.2a shows three jobs: $(1, 1, 1)$, $(1, 1, 2)$ and $(1, 2, 1)$. Figure 3.2b shows the merged variants of these jobs. This figure shows that $c(1)$ is executed twice because the prefixes, and thus the inputs to these two subjobs, do not match. However, the number of tasks executed is still reduced from 9 to size. Depending on the complexity of the tasks, this can result in a large performance gain.

3.2 Applying template based job merging to Spark Streaming

In the previous sections, we have explained the concept of our job merging approach. This this section, we describe how we applied our approach on Spark Streaming jobs by answering the subquestions of **DQ1** defined in section 1.6.

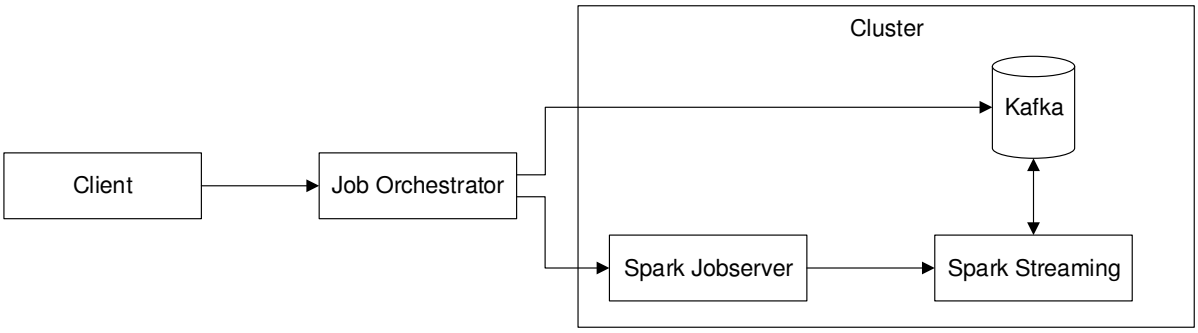


Figure 3.3: Architecture overview

```
1 object TemplateExample {
    def runJob(ssc: StreamingContext, param1: String, param2: String, ...) =
3     input
        .op1(...)
5       .op2(...)
        .op3(...)
7       ...
}
```

Listing 3.1: Template structure

Our job merging approach is implemented as a job orchestrator. Instead of submitting jobs directly to Spark, jobs are submitted to the job orchestrator using a simple REST interface. This is illustrated by fig. 3.3. The job orchestrator splits the job into smaller subjobs which are submitted to Spark using the Spark Jobserver [23]. Additionally, the job orchestrator creates Kafka topics for intermediate result sharing between subjobs. Section 3.2.3 explains this in more detail.

3.2.1 Template structure

Developers have to define their application using our template structure in order to benefit from job merging. Businesses make a trade-off between development costs and the costs of cluster resources. Therefore, it is important to reduce the development costs as much as possible.

Hence, we have decided to reuse the structure that already exists in many Spark Streaming applications and allowing developers to convert their applications if necessary and wrap them in our template structure.

Listing 3.1 shows the structure of the template. The tasks are wrapped in a method called `runJob`, which in turn is wrapped in a Scala object with an arbitrary name. The template designer must define the following elements:

StreamingContext
: The variable name of the `StreamingContext` must be defined in the method definition, in this case `ssc`. This allows using the `StreamingContext` in for example the input definition.

Parameter list
: The list of parameters is defined in the method definition. In listing 3.1, the parameters named `param1` and `param2`.

Input
: The input D-Stream on which the subsequent tasks are executed, called `input` in the example.

Chain of tasks
: The tasks applied on the input. These are regular D-Stream API methods. In the ab-

stracted example, the tasks are `op1`, `op2` and `op3`.

In order to extract the information from a template, scalameta is used. This is a metaprogramming tool, which allows us to parse Scala code using pattern matching and tree visiting. Therefore, we did not have to write a complex parser. Furthermore, template definitions are written as valid Scala code. There further minimizes the development overhead, since regular integrated development environments for Scala can be used.

Word count example

To illustrate how applications can be defined as we template, we reuse the word count example explained in section 2.1.1.

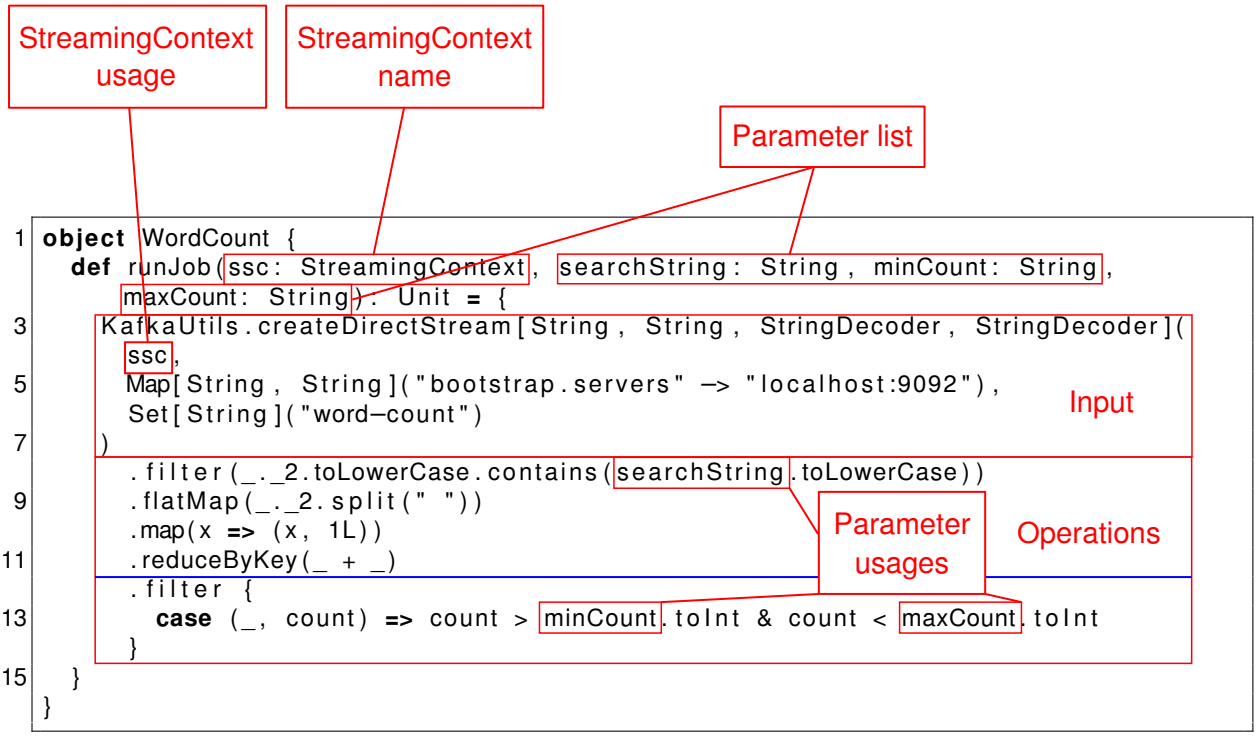


Figure 3.4: Word-count example explained

Figure 3.4 shows the implementation of this word-count template. We explain per line of code what this implementation does. The figure itself shows how this template fulfills our defined structure.

Lines 3-7 Input from a topic `word-count` on a local Kafka (refer to section 2.2) server publishing tweets.

Line 8 Filtering tweets that contain the string `searchString`.

Line 9 Splitting the tweet into words.

Line 10 Creating `(word, 1)` pairs.

Line 11 Group and reduce the pairs to `(word, count)`.

Lines 12-14 Filter words using condition $count >= minCount \wedge count <= maxCount$

3.2.2 Grouping tasks

Figure 3.4 shows that the word count example has 5 tasks (from the first to the last filter). Each of these tasks could be converted to a subjobs. However, this means that each subjob

is submitted separately on the cluster and potentially even run on separate nodes. This is inefficient and unnecessary if we group these tasks.

On the other hand, we could group all tasks into one subjob. However, we would then eliminate the ability to merge only the equivalent tasks of jobs and sharing the results to different following tasks.

In fig. 3.4, only 2 out of 5 tasks utilize the parameters defined in the method definition. Our grouping approach is based on this type of parameter usage in tasks. In our approach, we walk through each task and define group splitting points. For each task, the job merger decides whether to group the current task with the previous task group based on the following condition: $paramList \subset previousGroupParamList$.

Applying this on the template in fig. 3.4 results in two task groups. These groups are separated by the blue line. The splitting point is defined at this point because $\{minCount, maxCount\} \not\subset \{searchString\}$.

3.2.3 Inter-job communication

Subjobs are submitted and run as individual Spark Streaming applications and are therefore allocated to specific nodes in the cluster by the resource manager. As a consequence, individual subjobs of a specific job may run on different nodes. Therefore, there is a need for a distributed method for sharing intermediate results.

Our approach is based on using a distributed message broker. This message broker must meet the following conditions.

1. The message broker must perform well on large clusters.
2. The message broker must be integrable in Spark Streaming applications using existing libraries.
3. The message broker must have support for multiple separated channels.

Kafka meets these three conditions. Spark Streaming has out-of-the-box support for reading from and writing to Kafka topics. The word-count example in fig. 3.4 uses a Kafka topic called `word-count` as input and using the Kafka input D-Stream that exists in Spark Streaming.

In our approach, a unique Kafka topic is automatically created for each subjob by our job merger. Figure 3.5 shows the Kafka topics created for the jobs illustrated by fig. 3.2b where intermediate results are written to and read from.

Furthermore, subjobs output Scala objects. As described in section 2.2, messages in Kafka exist of byte arrays. To bridge this gap, the Kryo serializer [24] is used to serialize objects before writing them to Kafka and deserialize these object after reading from Kafka.

3.2.4 Subjob generation

In this section, we explain how runnable subjobs are generated from operation blocks. As the previous section explains, Kafka topics are used for sharing intermediate results with the next subjob. For each subjob, a Kafka topic is created.

Regarding the input of a subjob, there are two situations: the first subjob in the chain has an input defined by the template. For the following subjobs, the input is inserted by the subjob generator. Scalameta [25] is used for generating the subjob code, which is then compiled using the Scala compiler.

In appendix A.1, three example subjobs are shown, generated from the example in fig. 3.4

The first job has parameters {searchString = "weather", minCount = "3", maxCount = "10"}. This results in the subjobs shown in listing A.2 and listing A.2

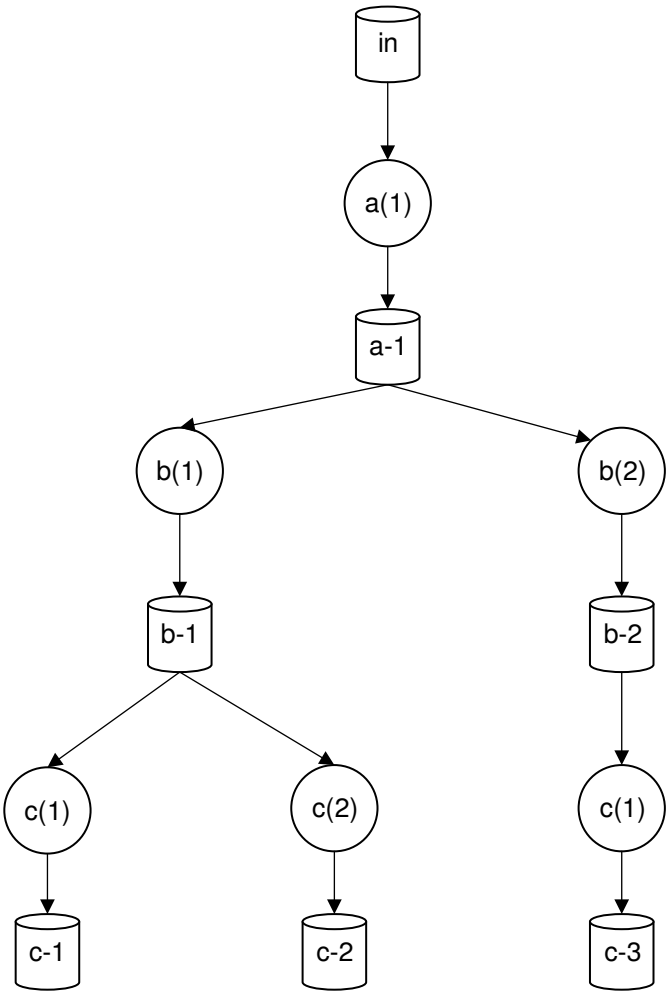


Figure 3.5: Inter-job communication

The second job has parameters {searchString = "weather", minCount = "2", maxCount = "5"}. This job reuses the job shown in listing A.2 and generates a new subjob shown in listing A.4

Internally, a tree of jobs is administrated to keep track of which subjobs are already running and which jobs have to be generated and run. Spark Jobserver [23] is used for submitting jobs on the cluster. This is illustrated by fig. 3.3.

3.3 Conclusions

The design question of this chapter is: ***DQ1****: How can similar stream processing jobs be orchestrated in order to merge equivalent subtasks?* In this section, we state to which extend our job merging approach succeeded in providing an answer to this question.

In section 3.1, we stated that in the ideal situation, developers can submit their jobs to the resource manager of the cluster and that the resource manager takes care of finding equivalent tasks. The resource manager then merges these tasks to reduce the load on the cluster.

However, this involves very complex code analysis. Furthermore, this would involve intensional function equality, which, to the best of our knowledge, is not achieved for programming languages [17].

Nevertheless, we have successfully designed and implemented a real-time job merging approach without the need of checking functional equality by using template-based jobs. Templates are composed of an input source and a chain of parameterized tasks. By applying arguments on a template, a concrete job is generated, which consists of separate subjobs. Using prefix

matching between template instances, the job merger determines equivalent subjobs, of which the results are reused.

This job merging approach is implemented for Spark Streaming applications. To accomplish this, the following subquestions are answered.

***SQ1**: How can job templates be defined with minimal development overhead?*

In our approach, jobs templates are defined as regular Scala objects, with enforces restrictions of the structure. Furthermore, tasks are defined using the D-Stream API. Therefore, existing simple Spark Streaming applications can be converted to this structure and wrapped in this Scala object with minimal development overhead.

***SQ2**: How can we automatically group tasks to reduce the merging overhead while maintaining the ability to merge subjobs?*

Instead of converting each task to one subjob, tasks are grouped. Therefore, the number of subjobs is reduced. We have accomplished this by decided for each task whether it can be grouped which the previous task group using this condition: $paramList \subset previousGroupParamList$.

***SQ3**: How can subjobs share intermediate results in a distributed environment?*

Not all subjobs of a specific job run on the same nodes. Therefore, our job merger creates a Kafka topic for each subjob. These topics are used for distributed intermediate result sharing.

CHAPTER 4

THROUGHPUT BENCHMARKING SPARK STREAMING

To validate the job merging approach describer in the previous chapter, a benchmarking method for Spark Streaming applications is needed. We have chosen to base our benchmarking approach on the evaluation method of Spark Streaming as described by Zaharia, Das, Li, *et al.* [9]. We use the same metric as Zaharia, Das, Li, *et al.*: the maximum throughput. This metric is defined as the maximum bytes per second processed while keeping the end-to-end-latency below a given target.

However, this paper omitted crucial details in order to reproduce their evaluation method. As explained in section 1.3.2, important questions are left unanswered. Questions such as: how is benchmarking data streamed to the application? How is the input rate regulated to prevent overflowing the benchmarking setup? Where the micro-batch stabilized before the experiment started?

Therefore, we have designed a new benchmarking method. This chapter answers the following question: ***DQ2**: How can Spark Streaming jobs be benchmarked with the maximum throughput as metric?*

During our research, we discovered that benchmarking Spark Streaming in a valid way is complex and challenging. This relies in the fact that Spark Streaming is a micro-batch computing framework. Therefore, we analyze the challenges of throughput benchmarking Spark Streaming applications in section 4.1.

One of the most important challenges is stabilizing micro-batches before benchmarking. Spark Streaming's built-in backpressure algorithm can be used to address this challenge. Therefore, we analyze the behavior of the backpressure algorithm in section 4.2.1 and describe in section 4.2.2 how this information can be used in our benchmarking method.

Finally, metrics of the benchmark have to be collected for calculating the maximum throughput. This is described in section 4.3.

4.1 Challenges in benchmarking Spark Streaming jobs

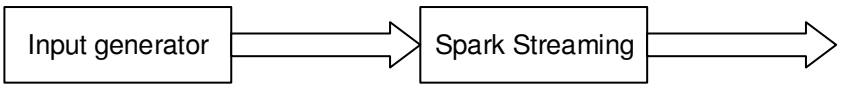


Figure 4.1: Benchmark setup

In this section, we elaborate on the challenges with respect to benchmarking Spark Streaming jobs. This chapter relies on a setup as illustrated by fig. 4.1. An input generator generates messages that are processed by the benchmarked application running on Spark Streaming.

The main challenge covers the fact that Spark Streaming is a micro-batch framework, rather than a real-time continues processing such as Storm [26] and S4 [27].

As section 2.1.1 explains, the batch interval size is fixed by the application developer. At the time a new batch starts, the data collected during the previous batch is processed. If the

capacity of the cluster is high enough, the actual processing time is less than the batch interval size.

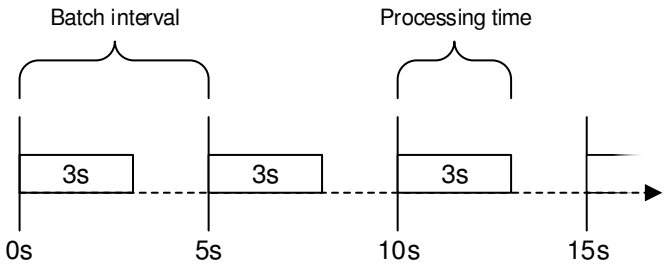


Figure 4.2: Stable, but not saturated

Consider fig. 4.2. In this figure, the interval size is 5 seconds. The cluster has enough capacity to process the stream. This is shown by the fact that the actual processing time is only 3 seconds. After the batch is processed, the cluster waits 2 seconds until the next batch starts. Hence, the application is running stable, but is not saturated.

However, this situation is undesirable when benchmarking a Spark Streaming application. We can measure the throughput, however this is not the maximum throughput.

To address this issue, the rate of the input stream can be increased for this benchmark. This will increase the load of the cluster. However, we do not know the desired throughput, since this is actually what we want to measure.

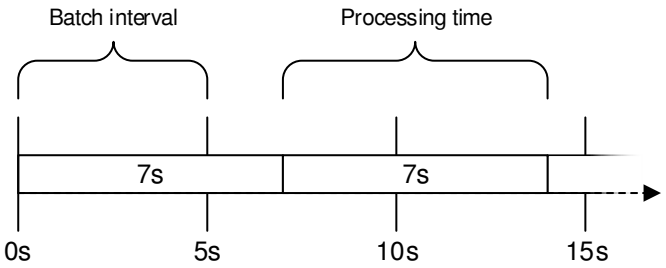


Figure 4.3: Saturated, but not stable

Increasing the input rate too much results in a saturated, but unstable application state. This is illustrated by fig. 4.3. In this figure, the processing time is 2 seconds larger than the interval size.

The result of having such situation is that the current job, each batch is executed as a separate Spark job, is not finished once the job for the new batch is scheduled. This results in a continuously increasing queue of jobs that are not yet started. Eventually the master node, which administrates these jobs, will be overflowed.

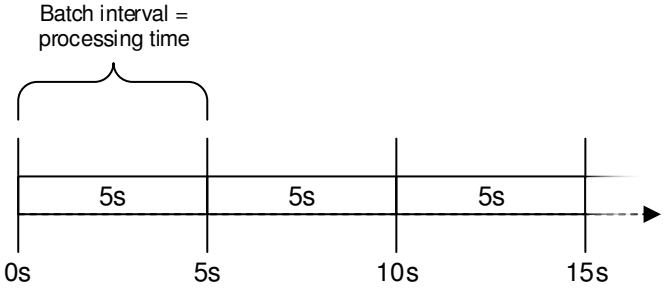


Figure 4.4: Optimized state

Hence, in the desired situation, the batches are both stable and saturated. In this chapter, we call this the optimized state. The processing time is then equal to the batch interval size. This is illustrated by fig. 4.4

The challenge of reaching this optimized state is controlling the rate of the input stream used for the benchmark. This has to be accomplished dynamically during the benchmark by using information about the processing time of recently completed batches and the number of records processed in this batch.

The second challenge is collecting metrics and calculating the maximum throughput using these metrics. For this purpose, the built-in metrics system is used. Section 4.3 describes how this system is used for this collection.

However, this metrics system does not record the maximum throughput. Furthermore, metrics are saved only for the last completed batch for each polling period. Therefore, the metrics system has to be correctly configurated and a method is necessary for calculating the maximum throughput for the metrics that are available. This is our second challenge.

4.2 Reaching an optimized application state

Reaching an optimized state is important to generate a valid throughput. The processing time has to be as close as possible to the batch interval. Therefore, we proposed a benchmarking method in section 1.4.2 that uses Spark Streaming's built-in backpressure algorithm.

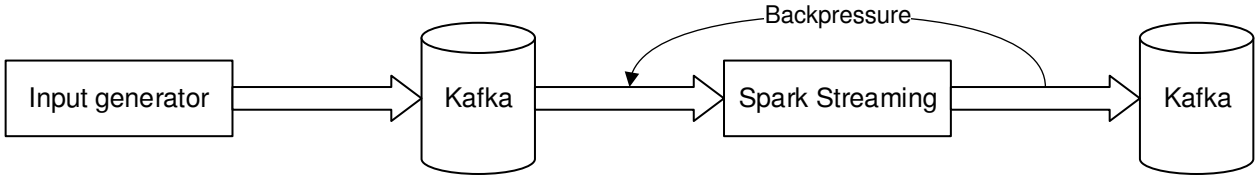


Figure 4.5: Backpressure

As explained in section 2.1, Spark Streaming applications have a receiver that collects data from the input stream that is processed during the next batch. The built-in backpressure algorithm allows for monitoring the processing time and using this for a feedback loop for controlling the input rate of the receiver. This is illustrated by fig. 4.5.

Since Kafka buffers records using the file system, backpressure effectively shifts the buffer from the memory of the receiver to the file system used by Kafka. As an effect, the input rate of the receiver is set to match the capacity of the cluster and the batches run in an optimized state.

However, there are two aspect that have to be considered. The rate of the input generator must be higher than the capacity of the cluster to prevent an unsaturated state.

Furthermore, the input rate is initially too high. Backpressure adjusts the input dynamically during running the application.

4.2.1 Backpressure explanatory experiment

To illustrate this effect, we have run a small experiment with backpressure enabled on a consumer laptop with 8 gb RAM and an Intel i5-6200U processor. We have used the Kafka record generator from section 5.2 to generate the input with a frequency of 500 records per second.

The results of this explanatory experiment are shown below. Figure 4.7 shows that the input size is around 500 records per batch for the first ten batches. Since this equals the defined

frequency, the effect of the backpressure is not visible yet. However, the batch durations, as shown in fig. 4.8, are approximately 6 seconds, which is six times the batch interval.

This results in an increasing scheduling delay, shown in fig. 4.6. Consequently, since batches are created once per interval, this in turn results in an increasing queue of unprocessed batches.

Subsequently, after 10 batches, backpressure causes the input size to drop to 10 records per batch as shown by fig. 4.7, to compensate for the high scheduling delay. At the moment the scheduling delay is close to 0, the input rate is set at 80 records per batch. This is the rate that results in batches of which the processing time equals the batch interval, that is 1 second (see fig. 4.9). In this experiment, reaching the optimized state took 76 seconds (fig. 4.8).

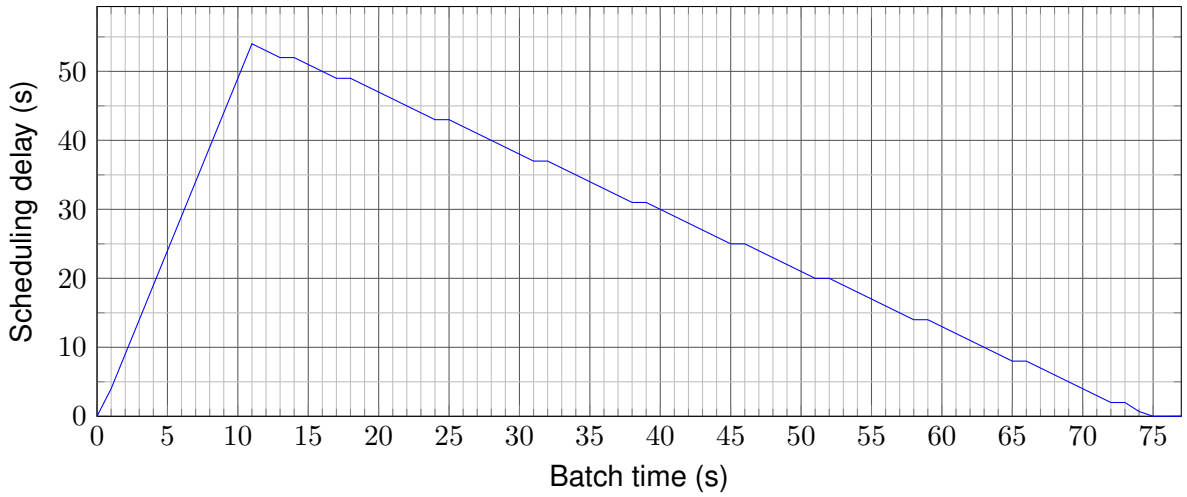


Figure 4.6: Scheduling delay

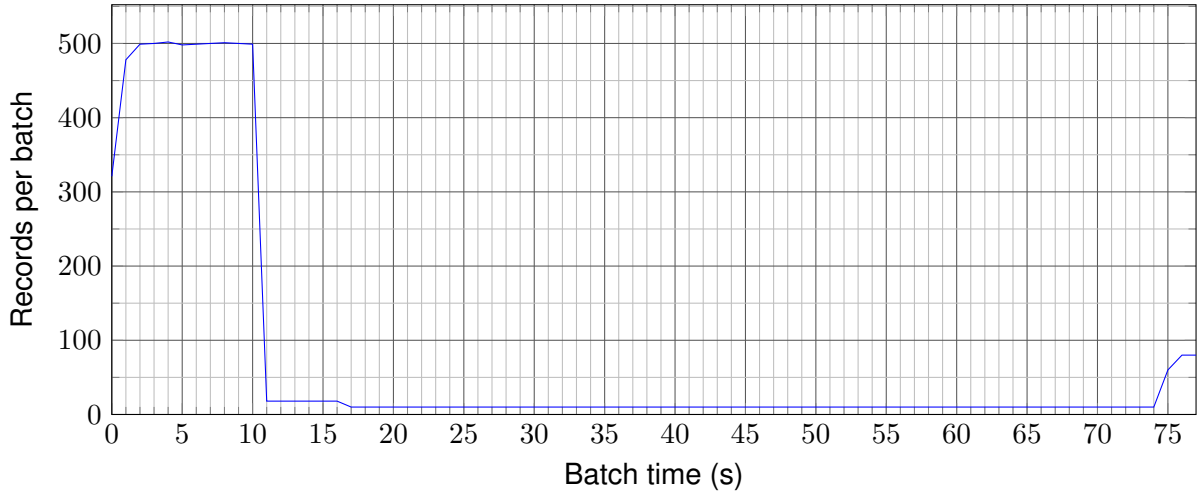


Figure 4.7: Batch input size

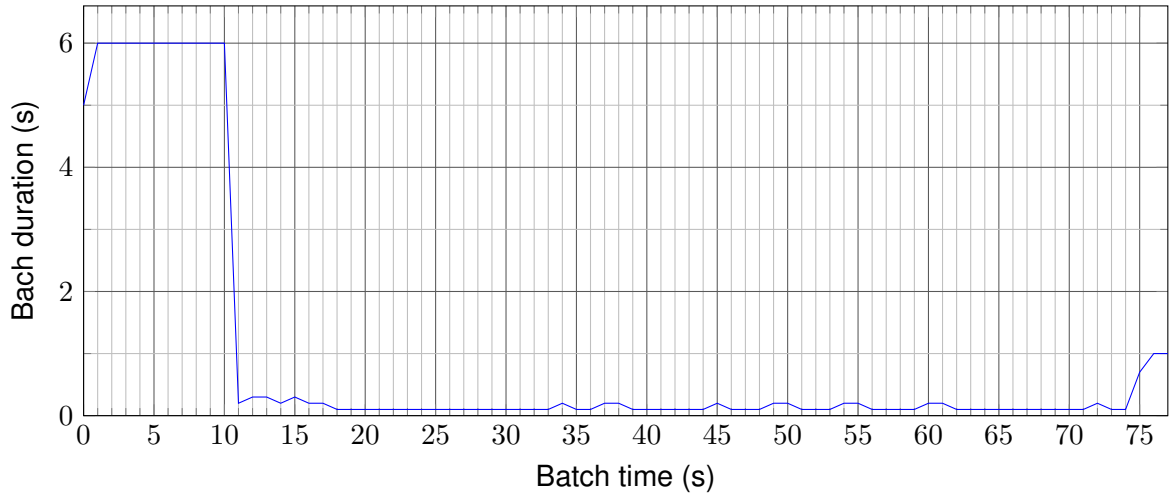


Figure 4.8: Bach duration

4.2.2 Applying backpressure

In the previous section, we explained the behavior of micro-batches when using backpressure and how this enables us achieving a processing time that equals the batch interval. The experiment shows that a number of iterations is needed to reach an optimized state.

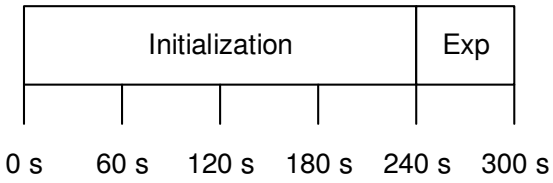


Figure 4.9: Throughput batch timeline

Since we are only interested in the throughput of Spark during the optimized state, we split the lifespan of the experiment into two phases: the initialization phase, during which Spark is in an unstable state, and the experiment phase, during which backpressure has stabilized Spark. These phases are shown in fig. 4.9.

In this figure, the duration of the initialization phase is set at 240 seconds, more than three times the actual initialization duration of our explanatory experiment. Therefore, we expect that this will be large enough for the actual experiment in chapter 5.

4.3 Metric collection and throughput calculation

In this section, we explain how Spark Streaming job metrics can be collected and how the throughput can be calculated using these metrics.

For collecting metrics, we use the built-in metrics system of Spark, described in section 2.1.3. As explained, the metrics system writes the current state of the application once per predefined polling period.

The following metric properties are used in our research:

- lastCompletedBatch_processingStartTime (abbreviated as processingStartTime)
- lastCompletedBatch_processingEndTime (abbreviated as processingEndTime)
- lastCompletedBatch_processingDelay (which is another name for the processing time of a batch)
- totalProcessedRecords (sum of processed records for all completed batches)

Since metrics are only reported once per polling period, this period must be smaller than the batch interval to prevent that batches are not recorded. As a consequence, duplicate entries may appear, especially if the application is not stabilized and the batches are bigger than defined.

Therefore, the first step is to duplicate entries, to ensure that there is a one-to-one mapping from the metric records to the batches.

After filtering duplicate entries, the next step is to discard the batches of the initialization phase. Figure 4.10 shows a segment of the timeline in fig. 4.9. The dotted lines show the separation of batches. This figure illustrates an example of how batches are filtered. To following formula is used:

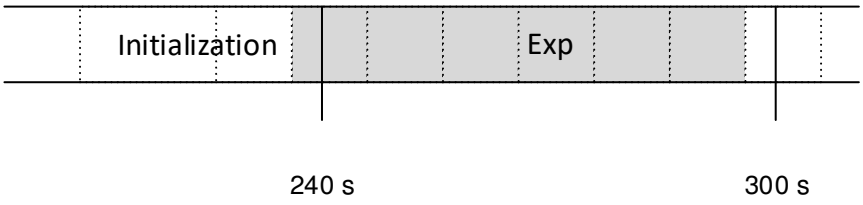


Figure 4.10: Filtering batches

$$\begin{aligned} processingStartTime + batchInterval/2 > endTime - expDuration \\ \wedge processingStartTime < endTime - batchInterval \\ \wedge processingEndTime < endTime \end{aligned} \tag{4.1}$$

In this formula, $endTime$ is the end time of the whole experiment, $expDuration$ is the duration of the experiment part and $batchInterval$ is the configured batch interval size. The conditions ensure the following:

1. More than half of the batch is lays after the start of the experiment phase.
2. The batch starts one interval before the end of the experiment phase.
3. The end of the actual processing time of the batch is before the end of the experiment phase.

After filtering the batches, the throughput per batch is calculated using the following formula:

$$\frac{totalProcessedRecords - prevTotalProcessedRecords}{max(batchInterval, processingDelay)} \tag{4.2}$$

In this formula, $prevTotalProcessedRecords$ is the $totalProcessedRecords$ of the previous batch. By calculating the difference, the number of records of the current batch is calculated.

Despite of choosing a liberal duration of the initialization phase, the application can still be unstable, since we have no method to check at runtime whether the initialization phase is long enough.

By using the maximum of the batch interval and the processing delay in the denominator of the formula, we are able to calculate the throughput, even if Spark is in an unstable state. However, this reduces the reliability of the experiment, which should be taken into account when interpreting the results of the benchmark.

4.4 Conclusions

In the introduction of the chapter, we showed that critical questions about the benchmarking method used by Zaharia et al in [9], the original Spark Streaming paper, are left unanswered. Therefore, we were not able use reuse this method for validating our job merging approach.

Hence, in this chapter, we designed a new benchmarking method for Spark Streaming applications. The main question of this chapter is: ***DQ2****: How can Spark Streaming jobs be benchmarked with the maximum throughput as metric?*

To answer this question, we have defined several subquestions. The first subquestion is about further analyzing the challenges of design such benchmarking method that were not addressed by Zaharia et al. This question is: ***SQ5****: What are the challenges in throughput benchmarking Spark Streaming applications?*

In section 4.1, we answered this question by describing two challenges. This first challenge is regulating the data input rate of the benchmark. An input rate that is too high results in an unstable application and an input rate that is too low result in the benchmark running under capacity. The benchmarking method must assure that the micro-batches are both stable and saturated.

Therefore, we proposed using backpressure for regulating the input rate. This resulted in the second subquestion of this chapter: ***SQ6****: How can Spark Streaming's backpressure be used to stabilize applications during benchmarks?*

The first step in answering this question is analyzing the backpressure behavior by running an experiment. Therefore, we have run an experiment, described in section 4.2.1. In this experiment, 76 seconds were needed for stabilizing the application. This information is used to describe in section 4.2.2 how backpressure can be applied to stabilize applications during benchmarks.

The second challenge we have analyzed is about collecting application metrics and use this information to calculate the maximum throughput of the application during the benchmark. This resulted in ***SQ7****: How can application metrics be collected and used for calculating the maximum throughput?*

In section 4.3, we presented an approach to answer this question. This approach uses the built-in metrics system of Spark. We described important configuration aspects, explained how the metrics are processed and presented a formula for calculating the maximum throughput.

CHAPTER 5

EXPERIMENTAL ANALYSIS

The objective of this chapter is describing the experimental analysis of the job merger, described by chapter 3. Merging jobs allows to share computations between these jobs, which potentially results in lower resources, or a higher throughput, depending on the utilization of the cluster Spark in running on.

The objective of the experimental analysis is answering ***RQ**: How well does runtime stream process orchestration with equivalent subtask merging perform compared to naïve streaming job scheduling?* Therefore, experiments are designed to compare the performance of merged jobs to unmerged jobs. The methodology is described in the following sections.

As a performance metrics, the maximum throughput is used. A method for throughput benchmarking Spark Streaming jobs is explained in chapter 4. This method is used in our experimental analysis.

However, this method is designed for benchmarking single jobs. Since we benchmark the performance of all nodes of the cluster, the benchmarking method of chapter 4 is modified for this purpose.

5.1 Synthetization of the NER scenario

For this experiment, we have synthesized the NER scenario described in section 1.2.1. This scenario provides a real-world use case for which we expect a large performance when running using our job orchestrator.

There are several reasons for using a synthetic variant of this scenario. Firstly, the NER scenario has tweets as input. These tweets have varying length. These variations influence the throughput, which reduces the validity of the experiments.

Secondly, the complexity of the NER operation depends on the text of the tweet, which causes an additional irregularity in the throughput.

Finally, the synthetic version of this scenario provides more flexibility for future work. In this research, we only focus on a two-step application. However, our synthetic version can be extended to a bigger number of steps. Furthermore, the load of each step can be adjusted.

5.1.1 Concept

The NER application consists of two steps: applying named-entity recognition on the tweets and filtering for a specific region. The NER operation is a complex and heavy operation, which we would like to merge. The filtering step, however, is very light, but different for each job.

Therefore, we are able to abstract this application to an application that only has these two aspects. This abstraction is illustrated by fig. 5.1. Our synthetic version consists of two tasks: a heavy and a light task.

(a) NER and then filter for location

(b) Abstracted application

Figure 5.1: Abstraction of the NER application

5.1.2 Implementation

We have implemented the concept described in the previous using the template structure described in section 3.2.1.

The template consists of two times a task, based on input from Kafka. This task takes the length of the input, which is a fixed number of random bytes (see section 5.2), multiplies this with itself for n number of times.

In the first map task, $n = 10^7$, to simulate the high-load NER task. In the second map task, $n = 25000$, to simulate the low-load location filter task. In future experiments, these numbers can freely be set to another number to simulate a different complexity.

By outputting the results of this computation to the next task or to a Kafka topic, it is prevented that the JIT compiler removes this code block as an optimization.

To enforce that the job merger this the job into two subjobs, the two tasks are parameterized with an id. For each job, the id of the first task is equal and different for the second task. This enforces the job merger to merge the first task and creating separate subjobs for the second task.

5.2 Input generation

Since the job used for our benchmark does not modify the content of the records, in contrast to the original NER use case, we are able to use synthetic data. We have used a message generator that generates messages that consist of a random array of bytes.

The message generator has three parameters:

1. message length,
2. frequency and
3. Kafka topic name.

The size and the frequency of the messages are predefined. The frequency is set at 1000 messages per second. This is higher than the capacity of the cluster when running one instance of the application described in section 5.1. The message size is set at 5 kB, which is approximately the size of a tweet JSON.

5.3 Benchmarking merged jobs

The benchmarking method described in chapter 4 is designed for benchmarking single Spark Streaming applications. However, validating the job merger involves running and benchmarking multiple jobs simultaneously. This section explains how this is accomplished.

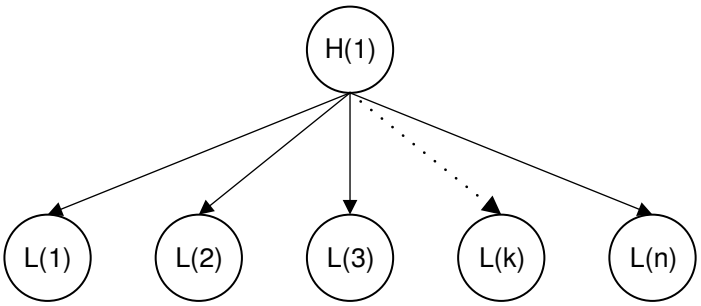


Figure 5.2: Fan-out

Therefore, we introduce how the concept of fan-out is used in this experimental analysis. Figure 5.1b illustrates the benchmark application. The heavy (H) task is shared and the light (L) task is different for each job. Hence, running n jobs result in 1 heavy subjob and n light subjobs. The fan-out in this context is defined as n, the number of light subjobs. This is illustrated by fig. 5.2. Note that each combination of heavy and light tasks belong to a single job, but that the heavy task is shared among all jobs.

Furthermore, the throughput of multiple jobs combined has to be calculated. In the case of unmerged jobs, there are no subjobs and each job directly relates to a Spark Streaming job of the cluster. Therefore, we calculate the throughput of each Spark job and calculate the average of these results in this case.

However, in the merged variants of the experiments, jobs are split in subjobs. Therefore, there are more Spark Streaming jobs running on the cluster than the number of jobs that are submitted. In our case, $|sparking\_streaming\_jobs| = fanout + 1$, since the heavy task is shared and the light task is different for each submitted job.

Hence, while the heavy subjobs runs on the cluster, only the throughput of the light task subjobs are taken into account. These subjobs are at the end of the task chain and therefore output the final results of the process and therefore, these throughputs are comparable to the throughput of the single unmerged jobs.

5.4 Cluster setup

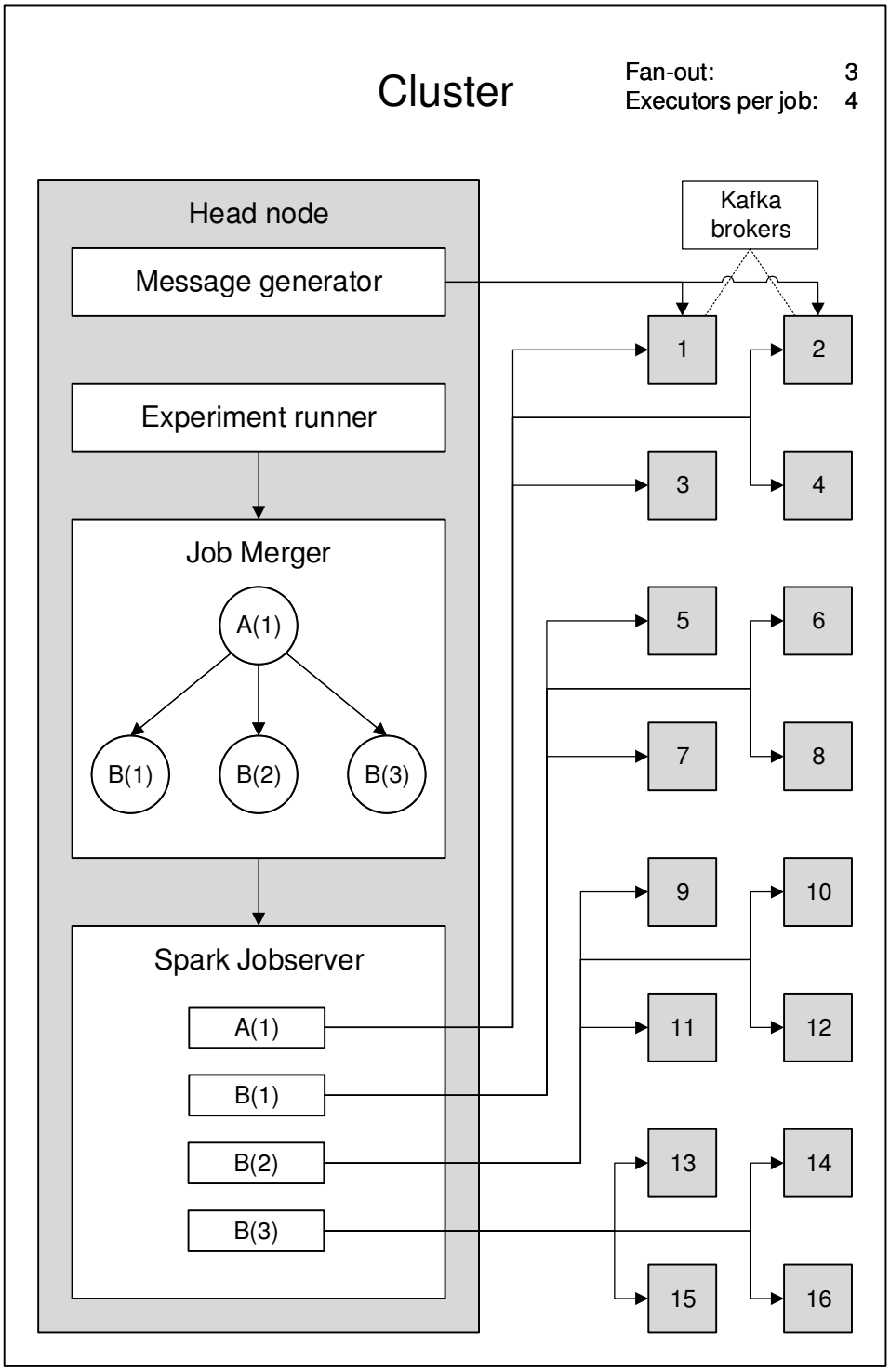


Figure 5.3: Cluster configuration

Figure 5.3 shows the system configuration for executing the experiments. The figure shows one head node and 16 worker nodes, of which node 1 and 2 are running Kafka. In order to reduce the complexity of the figure, this is less than the actual 53 worker nodes running on the cluster, of which 15 are running Kafka.

This configuration consists of four components, which are described below. As shown by the figure, these four components run on the head node of the cluster.

Message generator
: This component generates random messages used as input for the benchmark. Section 5.2 describes this in more detail.

Experiment runner
This component consists of a script that runs an experiment for every combination of fan-out and executors per job. After each experiment, the script collects the results and kills the running jobs.

Job merger
Generates subjobs as explained in section 3.2.4. This component is modified in order to be able to run both the merged and unmerged variant of a certain experiment.

Spark Jobserver
This is used by the job merger to start and stop Spark jobs on the cluster.

5.5 Experiment setup

There are three variables of which the combination is identical for each experiment:

- Merged (true or false)
- Executors per job (1 to 19, uneven)
- Fan-out (1 to 9, uneven)

Hence, there are 100 experiments. Since we are interested in the maximum throughput of the cluster, it is required that each node of the cluster is used. Otherwise, the throughput could theoretically be increased by using more nodes. Hence, since the cluster we have used has 53 worker nodes, the fan-out multiplied by the number of executors per job must be higher 53.

As fig. 4.9 shows, each experiment takes 300 seconds. The batch size is set on 10 seconds. The first 240 seconds of an experiment is used for stabilizing the backpressure algorithm of Spark Streaming. We expect that the application is stabilized after 240 seconds. The last 60 seconds, that is 6 batches, are used for the experiment itself.

Since we cannot assure if the length of the initialization phase is long enough, it is possible that not all experiments are stabilized after this phase. We can check this by counting the actual completed batches and comparing this with the expected number of batches. If the actual number of batches is lower, this is an indication of an unstable application. We also include a reliability indication in our results. This is calculated by dividing the actual number of batches by the expected number of batches.

$$reliability = \frac{actualNumberOfBatches}{\frac{experimentDuration}{intervalSize} * fanOut} \tag{5.1}$$

In section 4.3, we described that we are still able to calculate the maximum throughput, even though the application was not fully stabilized.

5.6 Results

Out of 100 experiments, 26 have succeeded. Out of the merged variants, 21 experiments succeeded. Experiment 22, with fan-out 5 and executors / job 3, is the last experiment with merged jobs that succeeded.

Out the unmerged variants, only 5 experiments succeeded. The last experiment of this variant that succeeded has a fan-out of 9 and 1 executor / job (experiment 41).

In section 5.5, we stated that is it necessary to use all nodes of the cluster in order to be able to retrieve the maximum throughput. However, the maximum number of nodes used where

both the merged and unmerged variant succeeded is 15 (experiment 22 and 23). Therefore, the maximum throughput is not reached.

The reason that all following experiments did not succeed is that the Spark Jobserver is not stable enough for Spark Streaming applications. From the logs, we can conclude that there were issues with stopping jobs. Therefore, newly submitted jobs were not started.

5.6.1 Results table

Appendix A.2 shows a table of the results. It has the following columns. The metrics apply to all Spark jobs in the case of unmerged jobs, and all light jobs, or leafs of the job tree, in the case of merged jobs.

Fan-out
The fan-out of the experiment as explained in section 5.3

Executors / job
The number of executors used per Spark job. Refer to section 2.1.2 for an explanation of the term executor.

Merged
Whether the job merger is used.

Avg processing time
The mean measured processing time of all corresponding batches.

Deviation processing time
The standard deviation of the processing time of the batches, weighted for each job.

Throughput
The mean throughput of all batches of all corresponding Spark job, calculated as described in section 4.3.

Deviation throughput
The standard deviation of throughput, weighted for each job.

Reliability
The reliability calculated as explained in section 5.5.

5.6.2 Graphs

This section shows the visualization of the results in the form of graphs. Each graph consists of three components.

Lines
The graph consists of two result lines: one for the merged and one for the unmerged experiments.

Error bars
The error bars show the standard deviation.

Circles
The circles show the results for the individual experiments in 10^4 kB/s. Per experiment, there are two circles: an inner and an outer circle. The ratio of the radius of the inner circle to the outer circle show the reliability of the experiment. The formula used to calculate the reliability is shown in section 5.5.

Executors / job = 1

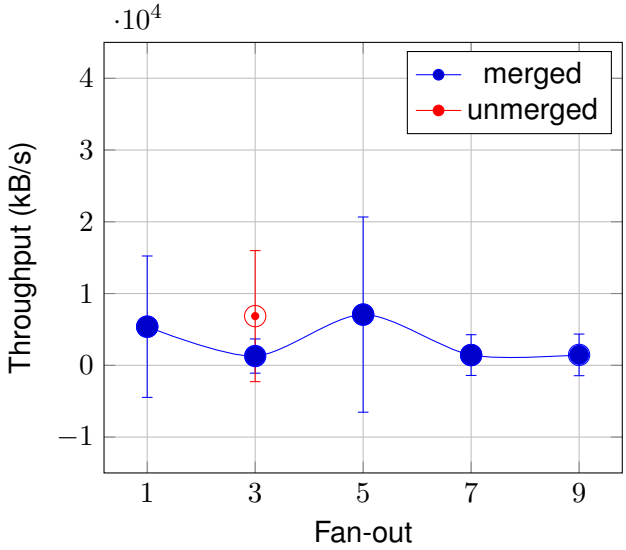


Figure 5.4: Executors / job = 1

The throughput is approximately 5.5 times as high for the unmerged case, compared with the merged case. The reliability for the unmerged experiment is one third, which is a repeating pattern for the following experiments. This indicates that the actual number of batches is less than expected for the experiment phase.

Therefore, the backpressure algorithm did not adjust the input rate on time. This conclusion is supported by the fact that appendix A.2 shows high batch durations for the corresponding experiments.

Furthermore, the throughput is remarkably low for the merged jobs with a fan-out of 3, 7, and 9. Furthermore, the deviation of these experiments is lower compared to the other experiments.

Executors / job = 3

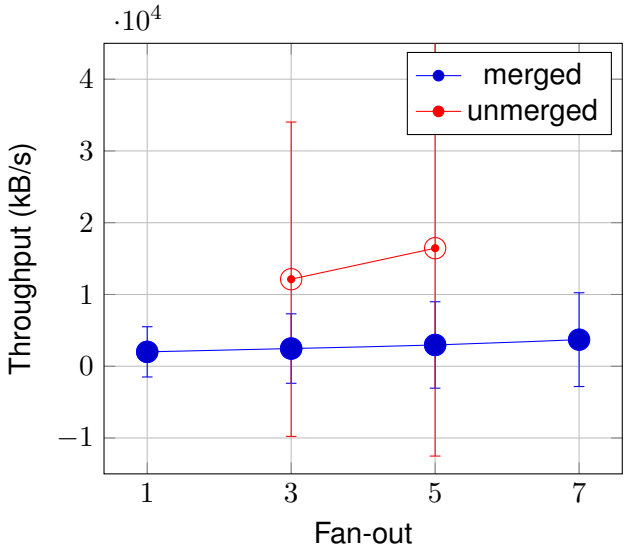


Figure 5.5: Executors / job = 3

The throughput of the unmerged jobs is roughly 5.5 times as high compared to the merged jobs. However, the deviation of these jobs is too high to draw any conclusions.

Furthermore, the throughput is increasing with the fan-out. Since a bigger fan-out results in a higher cluster utilization for a fixed number of executors per job. We stated at the start of this section that full cluster utilization is not reached by our experiments. Therefore, new jobs run on separate nodes and have a minor effect on the performance of other jobs.

Furthermore, the deviation is again high.

Executors / job = 5

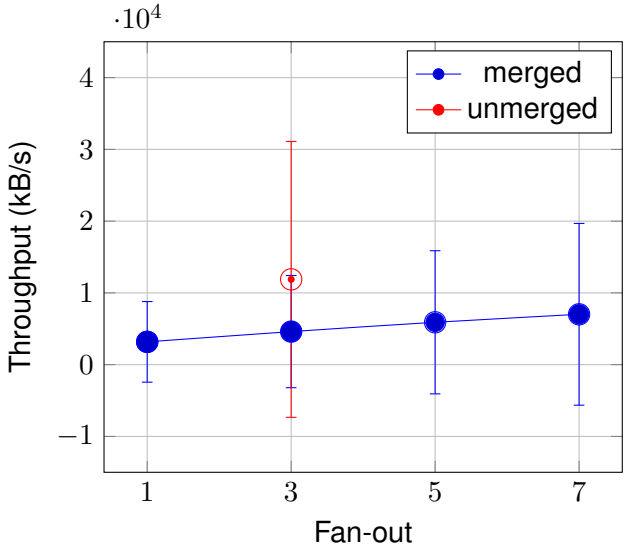


Figure 5.6: Executors / job = 5

The throughput of the only succeeded unmerged job is 2.5 times as high as its merged counterpart, again with an very high deviation. The throughput of the merged jobs is increasing with the fan-out.

Executors / job = 7

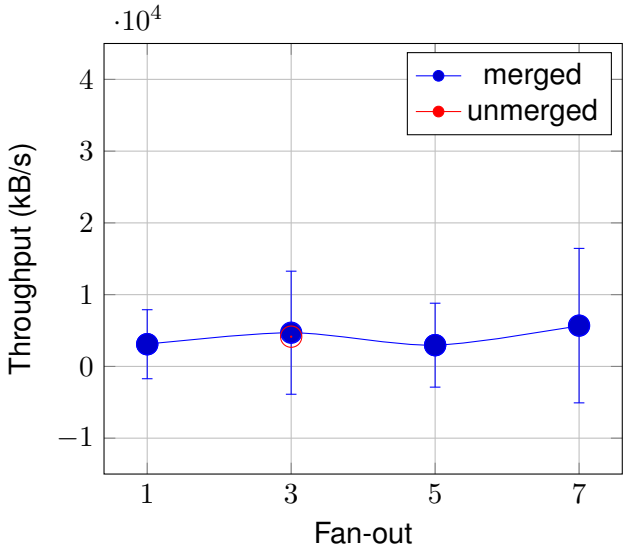


Figure 5.7: Executors / job = 7

The only succeeded unmerged experiment, which is difficult to see in the figure, has a very low reliability of approximately 5 percent. The throughput is varying for the fan-out.

Executors / job = 9

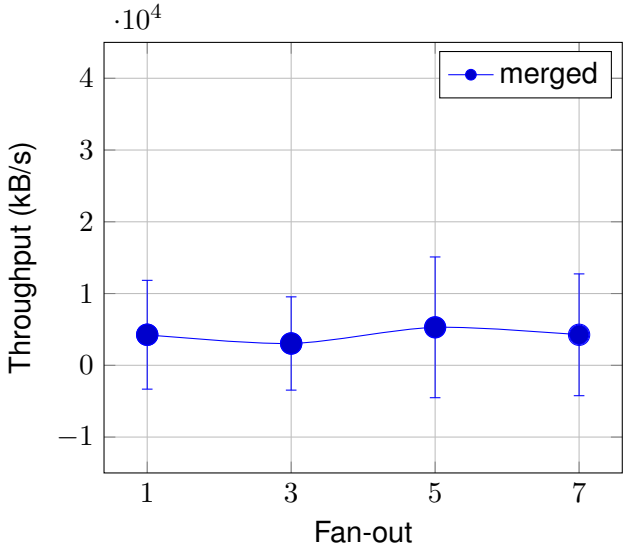


Figure 5.8: Executors / job = 9

Only four out of 5 merged experiments succeeded, with a throughput varying for the fan-out. None of the unmerged experiments succeeded.

Fan-out = 3

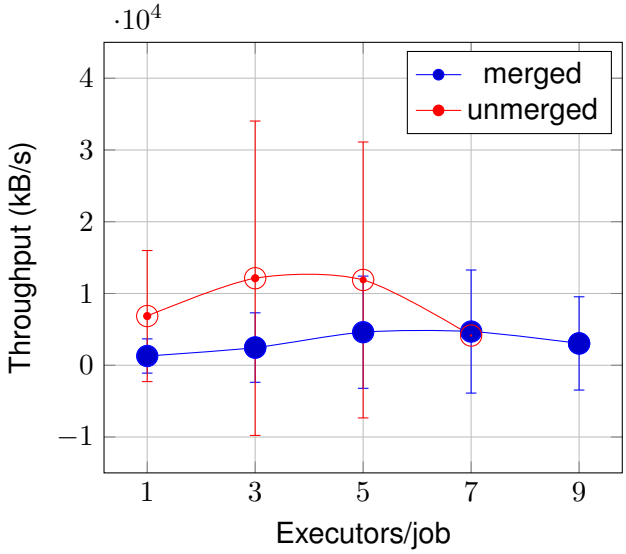


Figure 5.9: Fan-out = 3

For the figure above, the fan-out is fixed and the number of executors per job is increasing, to examine the relation between the number of executors per job and the throughput. The fan-out is fixed at 3, because for this fan-out, the highest number of experiments succeeded.

The figure shows that the throughput is increasing with the number of executors per job, up until 5 executors per job. From then, the throughput is decreases. Our expectation is that for a higher number of executors per job, the administration overhead of the executors is lower than the benefit of utilizing more nodes on the cluster.

5.7 Conclusions

The objective of this chapter is to answer the following research question.

***RQ**: How well does runtime stream process orchestration with equivalent subjob merging perform compared to naïve streaming job scheduling?*

For this purpose, we have reused our benchmarking method, explained in chapter 4. As benchmarking application, we have successfully used a synthesized version of the NER scenario.

We have generated 100 experiments with a varying number of executors per job and fan-out. 50 experiments cover merged jobs using the job merged and 50 experiments their unmerged variants, to compare their performances. These experiments are executed on the CTIT cluster of University of Twente

Out of these experiments, 26 have been completed. The main reason that the other experiments have not completed is that the Spark Jobserver proved not stable enough for use in combination of Spark Streaming applications.

Since, only 5 experiments with unmerged jobs succeeded, and because of the high deviation and low reliability of the results, we are not able reason about the performance of merged jobs versus unmerged jobs. Therefore, the answer to our research question is inconclusive.

CHAPTER 6

CONCLUSIONS

Big data technologies have taken big leaps in power and efficiency. While much research is accomplished on improving the efficiency of individual jobs, equivalent subjobs of similar stream processing jobs are not merged. The main question of this research is therefore: ***DQ1**: How can similar stream processing jobs be orchestrated in order to merge equivalent subtasks?*

We have successfully designed and implemented an orchestrator that merges subjobs generated from job templates. Using code analysis, templates are split into groups of tasks. By applying parameters to such template these groups are executed as individual subjobs, that communicate via Kafka topics. By analyzing the parameters of currently running jobs, we are able to reuse the results of equivalent subjobs.

We have defined a new benchmarking method for Spark Streaming applications for validating our job merger. Since critical details about the evaluation method used for Spark Streaming are missing in the original paper, we have made an analysis on the complex challenges of such benchmark method.

These challenges have been addressed by answering ***DQ2**: How can Spark Streaming jobs be benchmarked with the maximum throughput as metric?* Our approach on based on dynamically adjusting the input rate of the benchmark accomplished using the built-in backpressure algorithm of Spark Streaming.

This method is used to answer the research question of this thesis: *How well does runtime stream process orchestration with equivalent subtask merging perform compared to naïve streaming job scheduling?* Therefore, a set of experiments, varying in the fan-out and the number of executors per job, is executed on the CTIT cluster of University of Twente, to compare the performance of unmerged jobs to merged jobs.

Only 26 experiments of the 100 generated experiments have succeeded without errors. This is mainly the result of software that is not yet stable. Furthermore, the results have a high deviation and partly a low reliability. Therefore, the results are inconclusive and we are unable to answer our research question. While we have succeeded in designing and implementing a job merging approach, further research is needed to validate this approach.

CHAPTER 7

DISCUSSION AND FUTURE WORK

In section 7.1, we describe the limitations of this research and in section 7.2, we suggest future work.

7.1 Discussion

7.1.1 Extra end-to-end latency

In Spark Streaming, records are processes in batch intervals. In our merging approach, a record is processed by several subjobs, with each their own latency caused by the micro-batch approach of Spark Streaming. Hence, merging jobs increases the end-to-end latency of record processing.

7.1.2 Job restrictions enforced by the template structure

The template structure defined in section 3.2.1 enforces restrictions on the expressiveness of jobs. These restrictions allow us to determine the merge points of jobs to improve the efficiency of resource usage. To reduce the complexity of this thesis project, we have decided to only accept jobs that execute a chain of tasks on an input. Section 7.2.2 suggests future work on this topic.

7.1.3 Fixed initialization phase

In section 4.2.2, we described that the length of the initialization phase of the benchmarking method is fixed in advance. This phase is needed for the backpressure algorithm to adjust the input rate to the capacity of the system that is running the benchmark.

The length of this phase we have used in our validation is based on an experiment that we have executed. Furthermore, we have taken into account that the actual needed phase can be larger than expected. However, as mentioned in section 5.6, the results show that an even larger initialization phase than used was needed. Therefore, we suggest future work on dynamically setting the length of this phase section 7.2.3.

7.1.4 Difference reliability merged and unmerged experiments

The results in section 5.6 show a large difference in the reliability of the merged and unmerged variants of experiments. The reliability of experiments with merged jobs is 100 percent in all cases, while the reliability of experiments with unmerged jobs is less than 34 percent.

This can be explained as follows. Our benchmarked application consists of two tasks. A heavy and a light task. As described in section 5.3, the throughput of a merged job is calculated by using the metrics of the generated light subjob.

However, the throughput is restricted by the heavy subjob, since the light subjobs process the results of the heavy subjob. Therefore, the light subjobs always have a processing time smaller

than the batch interval, and are always stabilized. Since the unmerged jobs contain both tasks, this restriction does not apply on the unmerged variants of the experiments.

7.1.5 Increasing throughput with fan-out

One of the interesting aspects of the results is the increasing throughput relating with an increasing fan-out. Since the fan-out equals the number of jobs in the case of the unmerged experiments and equals the number of subjobs that execute the light task in the case of the merged experiments, it is expected that an increase of the fan-out results in a decrease of the throughput of the jobs.

There are two possible explanations of the fact that this is not the case, which do not exclude each other. The first explanation is that this effect is insignificant because of the large deviations.

The second explanation is that full capacity of the cluster has not been reached. Therefore, there are nodes on the cluster that remain idle. An increase of the number of jobs running results in a higher number of utilized nodes. Hence, the throughput does not need to decrease.

7.2 Future work

7.2.1 Integrate job merging in Spark Streaming

Our job merging orchestrator is an extra layer that runs separate of Spark Streaming and submits job using the Spark Job server. To improve the performance of job merger, we suggest research on integrating job merging in Spark Streaming.

This allows for tight integration with the built-in scheduler and therefore, no separate communication system is needed for sharing intermediate results between tasks.

7.2.2 Support more complex job structures

Section 7.1.2 describes that the template structure in our approach only allows defining templates that consists of chains of tasks. Further research is needed on template structures that allow directed acyclic graphs (DAGs) of tasks to allow for merging jobs with a greater complexity.

7.2.3 Dynamically adjusting initialization phase

Section 7.1.3 describes that the initialization phase is set in advance. Our results show that the length of the initialization phase we used was too short in some cases.

Ideally, the length of this is dynamic for each benchmark. We suggest research on methods to measure in real-time whether a Spark Streaming application is stable and use this information to decide if the experimental phase can be started.

7.2.4 Improving backpressure

The built-in backpressure algorithm of Spark Streaming only throttles the input rate of the receiver. This is illustrated by fig. 4.5. As a consequence, the input generator must use a frequency of which we know that is too high to be processed by the benchmark application in order to prevent an unsaturated state.

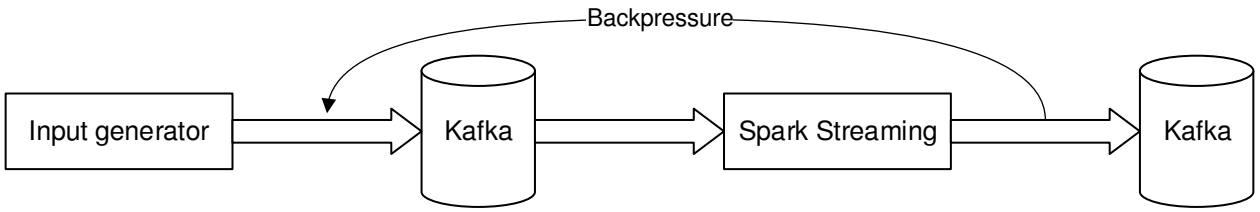


Figure 7.1: Backpressure suggested

Therefore, we suggest extending the backpressure algorithm used in our benchmarking method to regulate the frequency of the input generator instead. This is illustrated by fig. 7.1

7.2.5 New validation job merger

Further research is needed on validating the job merger. We suggest that the job server is validated using a smaller cluster. In this way, using this cluster at full capacity is reached with running less jobs. This reduces the chance of errors. Furthermore, the Spark Jobserver is in active development. Therefore, problem with the stability of this software can possibly be addressed in the future.

CHAPTER 8

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *To appear in OSDI*, p. 1, 2004.

[2] G. Pavlin, T. Quillinan, F. Mignet, and P. de Oude, "Exploiting intelligence for national security," *Strategic Intelligence Management: National Security Imperatives and Information and Communications Technologies*, p. 181, 2013.

[3] G. Pavlin, M. Kamermans, and M. Scafes, "Dynamic process integration framework: Toward efficient information processing in complex distributed systems," *Informatica*, vol. 34, no. 4, pp. 477–491, 2010.

[4] T. Terpstra, A. De Vries, R. Stronkman, and G. Paradies, *Towards a realtime Twitter analysis during crises for operational crisis management*. Simon Fraser University, 2012.

[5] J. Lingad, S. Karimi, and J. Yin, "Location extraction from disaster-related microblogs," in *Proceedings of the 22Nd International Conference on World Wide Web*, ser. WWW '13 Companion, Rio de Janeiro, Brazil: ACM, 2013, pp. 1017–1020, ISBN: 978-1-4503-2038-2. DOI: 10.1145/2487788.2488108. [Online]. Available: http://doi.acm.org/10.1145/2487788.2488108.

[6] S. Karimi, J. Yin, and C. Paris, "Classifying microblogs for disasters," in *Proceedings of the 18th Australasian Document Computing Symposium*, ser. ADCS '13, Brisbane, Queensland, Australia: ACM, 2013, pp. 26–33, ISBN: 978-1-4503-2524-0. DOI: 10.1145/2537734.2537737. [Online]. Available: http://doi.acm.org/10.1145/2537734.2537737.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," 2003.

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, USENIX Association, 2012, pp. 2–2.

[9] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania: ACM, 2013, pp. 423–438, ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522737. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522737.

[10] J. Fan, H. Chen, and F. Hu, "Adaptive task scheduling in storm," in *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*, vol. 01, Dec. 2015, pp. 309–314. DOI: 10.1109/ICCSNT.2015.7490758.

[11] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, ser. DEBS '13, Arlington, Texas, USA: ACM, 2013, pp. 207–218, ISBN: 978-1-4503-1758-0. DOI: 10.1145/2488222.2488267. [Online]. Available: http://doi.acm.org/10.1145/2488222.2488267.

[12] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware '15, Vancouver, BC, Canada: ACM, 2015, pp. 149–161, ISBN: 978-1-4503-3618-5. DOI: 10.1145/2814576.2814808. [Online]. Available: http://doi.acm.org/10.1145/2814576.2814808.

[13] J. Zhang, C. Li, L. Zhu, and Y. Liu, "The real-time scheduling strategy based on traffic and load balancing in storm," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Dec. 2016, pp. 372–379. DOI: 10.1109/HPCC-SmartCity-DSS.2016.0060.

[14] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *2014 IEEE 34th International Conference on Distributed Computing Systems*, Jun. 2014, pp. 535–544. DOI: 10.1109/ICDCS.2014.61.

[15] X. Liao, Z. Gao, W. Ji, and Y. Wang, "An enforcement of real time scheduling in spark streaming," in *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*, Dec. 2015, pp. 1–6. DOI: 10.1109/IGCC.2015.7393730.

[16] X. Ren and O. Curé, "Strider: A hybrid adaptive distributed RDF stream processing engine," *CoRR*, vol. abs/1705.05688, 2017. [Online]. Available: http://arxiv.org/abs/1705.05688.

[17] A. W. Appel, "Intensional equality ;=) for continuations," *SIGPLAN Not.*, vol. 31, no. 2, pp. 55–57, Feb. 1996, ISSN: 0362-1340. DOI: 10.1145/226060.226069. [Online]. Available: http://doi.acm.org/10.1145/226060.226069.

[18] Apache Software Foundation. (May 2016). Apache Cassandra, [Online]. Available: http://cassandra.apache.org/.

[19] ——, (Nov. 2016). Spark 1.6.3 Documentation, [Online]. Available: http://spark.apache.org/docs/1.6.3/.

[20] J. Kreps, N. Narkhede, J. Rao, *et al.*, "Kafka: A distributed messaging system for log processing," NetDB, 2011.

[21] Apache Software Foundation. (Jul. 2017). Apache Kafka, [Online]. Available: http://kafka.apache.org/documentation.

[22] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems.," in *USENIX Annual Technical Conference*, vol. 8, 2010, p. 9.

[23] Spark Jobserver contributers. (Feb. 2016). Spark Jobserver, [Online]. Available: https://github.com/spark-jobserver/spark-jobserver.

[24] Esoteric Software. (Jan. 2017). Kryo serializer, [Online]. Available: https://github.com/EsotericSoftware/kryo.

[25] M. Duhem and E. Burmako, "Parser macros for scala," Tech. Rep., 2015.

[26] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm @twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14, Snowbird, Utah, USA: ACM, 2014, pp. 147–156, ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2595641. [Online]. Available: http://doi.acm.org/10.1145/2588555.2595641.

[27] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *2010 IEEE International Conference on Data Mining Workshops*, Dec. 2010, pp. 170–177. DOI: 10.1109/ICDMW.2010.172.

APPENDIX A

APPENDIX

A.1 Code listings

```
import ...

object CPUIntensiveJob {
  def runJob(ssc: StreamingContext, id0: String, id1: String): Unit = {
    KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder](ssc
        , Map[String, String]("bootstrap.servers" -> "localhost:9092"), Set[String](
        "test-input")).map(records => {
      val id = id0
      val iterations = 10000000
      val operand = records._2.length
      var result = operand
      var i = 0
      while (i < iterations) {
        result *= operand
        i += 1
      }
      (records._2, result)
    }).map(records => {
      val id = id1
      val iterations = 25000
      val operand = records._2
      var result = operand
      var i = 0
      while (i < iterations) {
        result *= operand
        i += 1
      }
      (records._2, result)
    })
  }
}
```

Listing A.1: Generic experiment job in job merging format

```
import ...

object WordCount extends SparkStreamingJob {
  private val searchString = "weather"

  override def runJob(ssc: StreamingContext, jobConfig: Config): Any = {
    KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder](
      ssc,
      Map[String, String]("bootstrap.servers" -> "localhost:9092"),
      Set[String]("word-count")
  )
    .filter(_._2.toLowerCase.contains(searchString.toLowerCase))
    .flatMap(_._2.split(" "))
    .map(x => (x, 1L))
    .reduceByKey(_ + _)
    .writeToKafka({
      val p = new Properties()
      p.setProperty("bootstrap.servers", "localhost:9092")
      p.setProperty("key.serializer", classOf[StringSerializer].getName)
      p.put("value.serializer", classOf[KryoSerializer])
      p
    }, s => new ProducerRecord[String, AnyRef]("word-count-d0-1", s))

    ssc.start()
    ssc.awaitTermination()
  }
  override def validate(ssc: StreamingContext, config: Config): SparkJobValidation
    = SparkJobValid
}
```

Listing A.2: Example word count job 0

```
import ...

object WordCount extends SparkStreamingJob {
  private val minCount = "3"
  private val maxCount = "10"

  override def runJob(ssc: StreamingContext, jobConfig: Config): Any = {
    KafkaUtils.createDirectStream[String, AnyRef, StringDecoder, KryoSerializer](
      ssc,
      Map[String, String]("bootstrap.servers" -> "localhost:9092"),
      Set[String]("word-count-d0-1")
  )
    .map(_._2.asInstanceOf[Tuple2[String, Long]])
    .filter({
      case (_, count) =>
        count >= minCount.toInt & count <= maxCount.toInt
    }).writeToKafka({
      val p = new Properties()
      p.setProperty("bootstrap.servers", "localhost:9092")
      p.setProperty("key.serializer", classOf[StringSerializer].getName)
      p.put("value.serializer", classOf[KryoSerializer])
      p
    }, s => new ProducerRecord[String, AnyRef]("word-count-d1-1", s))

    ssc.start()
    ssc.awaitTermination()
  }

  override def validate(ssc: StreamingContext, config: Config): SparkJobValidation
    = SparkJobValid
}
```

Listing A.3: Example word count job 1

```
import ...

object WordCount extends SparkStreamingJob {
  private val minCount = "2"
  private val maxCount = "5"

  override def runJob(ssc: StreamingContext, jobConfig: Config): Any = {
    KafkaUtils.createDirectStream[String, AnyRef, StringDecoder, KryoSerializer](
      ssc,
      Map[String, String]("bootstrap.servers" -> "localhost:9092"),
      Set[String]("word-count-d0-1")
  )
    .map(_._2.asInstanceOf[Tuple2[String, Long]])
    .filter({
      case (_, count) =>
        count >= minCount.toInt & count <= maxCount.toInt
    }).writeToKafka({
      val p = new Properties()
      p.setProperty("bootstrap.servers", "localhost:9092")
      p.setProperty("key.serializer", classOf[StringSerializer].getName)
      p.put("value.serializer", classOf[KryoSerializer])
      p
    }, s => new ProducerRecord[String, AnyRef]("word-count-d1-2", s))

    ssc.start()
    ssc.awaitTermination()
  }

  override def validate(ssc: StreamingContext, config: Config): SparkJobValidation
     = SparkJobValid
}
```

Listing A.4: Example word count job 2

A.2 Results table

ID	Fan-out	Executors/job	Merged	Avg processing time	Dev processing time	Avg throughput	Dev throughput	Reliability
1	1	1	true	657	31	5382	9853	1.000
3	1	3	true	2172	431	2003	3500	1.000
5	1	5	true	1398	1464	3177	5618	1.000
7	1	7	true	904	738	3092	4809	1.000
9	1	9	true	2129	1721	4260	7581	1.000
10	3	1	false	26098	18963	6855	9132	0.333
11	3	1	true	747	157	1290	2386	0.944
12	3	3	false	9949	92	12127	21905	0.333
13	3	3	true	610	468	2466	4840	1.000
14	3	5	false	9201	446	11886	19225	0.278
15	3	5	true	1917	3092	4606	7818	0.944
16	3	7	false	62085	0	4143	0	0.056
17	3	7	true	2012	1769	4696	8575	0.944
19	3	9	true	365	234	3041	6499	1.000
21	5	1	true	1847	1278	7068	13602	1.000
22	5	3	false	9962	60	16448	28967	0.333
23	5	3	true	865	518	2964	6021	1.000
25	5	5	true	1999	2345	5903	9974	0.867
27	5	7	true	899	623	2959	5848	1.000
29	5	9	true	2019	1132	5292	9808	0.967
31	7	1	true	307	157	1427	2840	1.000
33	7	3	true	2063	2246	3704	6536	0.976
35	7	5	true	1658	2180	7015	12665	0.905
37	7	7	true	1783	1379	5684	10763	0.976
39	7	9	true	622	502	4253	8487	0.905
41	9	1	true	470	238	1454	2900	0.889