Faculty of Fundamental Problems of Technology
Wroclaw University of Science and Technology

Online parameter optimization for Spark Streaming

Michał Kiełbowicz
Index no: 204441

Supervisors:
Dr Eng. Jakub Lemiesz
Dr Thomas Heinze

Politechnika Wrocławska
Wrocław 2017

Contents

Chapter 1

Introduction

In the modern world there is an increasing amount of data created every minute. Almost half of World's population has an Internet access. Nearly every person possesses a mobile phone. Simple items that are used on regular basis, like watches, are becoming part of Internet of Things network. Cars and other elements of common infrastructure are getting equipped with more and more sensors. All of this generates astounding amounts of data that can be analyzed in order to e.g. produce services of better quality, detect fraudulent activities or customize user's experience.

In common assumption the rate of global Internet traffic will reach 50 TB per second by 2018. Even now a single service such as Google Search Engine must handle approximately 40 000 requests per second[14]. Since nowadays mankind is relying on software tools much more than ten or twenty years ago, there is a need for scalable, fault-tolerant systems that are adaptable for future growth of data income. It's definitely not easy to create such infrastructure, though currently there is a variety of available solutions that can help with achieving these goals.

This thesis will study a novel architecture of data stream processing systems, which handle the workload efficiently by partitioning it into data sets with a predefined time interval, so-called micro-batches. The most prominent representative of this architecture is *Spark Streaming*. This has certain advantages and disadvantages compared to a classical data stream processing, currently widely represented by *Apache Storm*. The usage of the novel architecture is especially interesting for research on a dynamic optimization as such system can be more easily switched between different system configurations.

The goal of this work is to create a state-of-the-art online parameter optimization subsystem for *Apache Spark*, the ecosystem wrapping the streaming component. At the moment the only solutions available when it comes to tuning the cluster resources is either manual intervention, keeping everything fixed or using volatile solution from upstream source that is not said to be better suited for production grade needs. While sufficient for some of the use cases, it can also lead to allocating unnecessary resources that in turn leads to higher usage costs. This is particularly important in applications performing Complex Event Processing (CEP) on incoming load that varies a lot in time.

Developed solution is to be based on existing prototype of elastic scaling for FUGU[12][13] streaming engine. This requires reviewing existing ideas and adopting the system for Spark's micro-batch processing model. Afterwards the results shall be evaluated in contrast of existing approaches.

Chapter 2

Stream Processing Engine

2.1 Stream processing

There are a lot of cases that require real-time analysis. Those are usually critical services where information is coming on multiple channels (*streams*) and evaluation (*processing*) is done in an automatic way. A simple example of such system is dealing with transactions in online banking. In regular case whenever a transfer of large value is scheduled by a client, a phone confirmation is required. What is actually happening is that every few second servers are getting a lot of requests of such origin and they are automatically filtered based on user's individual safety settings. Those that are later flagged as suspicious are prioritized and send to call center for direct verification.

Such area of technology is called stream processing and comes with few requirements[17]:

1. **Online evaluation.** Processing needs to be done in a swift manner without adding severe slow-downs - such as performing costly I/O operations. Instead data should be analysed as it comes and stored afterwards only when necessary.

2. **Support of a high-level query language.** Developer must be provided with enough abstraction that allows easy creation of processing pipeline. An example of such tool is plain SQL where simple commands such as GROUP BY or WHERE are available. Current solutions go even further with providing tools for e.g. machine learning or writing code in a more high-level language.

3. **Handling stream imperfections.** Sometimes a tuple sequence can contain records that are somehow corrupted (e.g. by broken source). In such cases the system should be capable of handling imperfections without breaking.

4. **Determinism.** Data processing should always be done in the same, predictable way. Whenever some evaluation of tuple window needs to be repeated, it always delivers the same results - considering a case where information doesn't change.

5. **Usage of past data sets.** It may happen that analysis of historical information is needed when it comes to more complex queries. For instance considering our major online banking example, fraud detection could be also performed with analysis of previous user's activities. Because of that handling already stored and evaluated tuples is mandatory.

6. **Fault tolerance.** Processing systems are usually part of critical services. While being complex, it must also provide high-availability. Even if there is a severe failure on server side, restoration is to be done in a way that preserves real-time constraints.

7. **Elasticity.** On average basis intensity of a stream has a high variance. When running a full-time cluster can be a solution, more economic approach is preferred. System should be scaling proportionally to current load. This demands having a distributed model of computation where processing is done in parallel.

8. **Keeping latency constraint.** Since real-time system is used, this demands having a high-performance solution that is capable of maintaining *Quality of Service* (QoS) on a reliable level.

While some functionality can be achieved with RDBMS, the most prominent type of system for handling stream processing is called *Stream Processing Engine* (SPE). Generally speaking it is a major software platform enabling developer to create complex, parallelized, fault-tolerant data handling pipeline with low latency outcome.

Since early 2000s multiple systems serving this purpose have been created in spite of rapid growth of global information storage capacity, with the first ones being Aurora and Medusa research projects[20]. Those were quite simple in design with load shedding or certain lack of scalability. However the business value was strong enough to pursue this subject further with next iterations like Borealis or StreamCloud[1][11].

As it can be foreseen, solutions satisfying these conditions are quite complex in structure. Systems meeting all those expectations, such as *Apache Storm* or *Apache Flink*, exist and are already used by companies such as Twitter, Yahoo or Spotify[4]. There is however a great area for improvement.

2.2 General architecture

As said, stream processing consists of obtaining continuous sequence of tuples from one or more resources, processing it inside query specified by developer and outputting results in a real-time fashion.

While this may seem simple there are few major techniques that have in goal fault-tolerance and handling information in a distributed system. One of them is called *intra-operator parallelism*[8]. While partitioning incoming stream onto several worker nodes is quite logical step, another one is to divide it also by *query operators* as in Figure 2.1[11].

With such approach data from current stream window is randomly partitioned onto several worker nodes. Each one of them processes its part of information and in the end output is ready for further management. Intra-query parallelism adds another division between parts of query that end with *stateful operator*. In opposition to stateless ones (like FILTER or MAP), stateful operators require handling tuples with keeping some common information across cluster. In the example from figure 2.1 we deal with GROUP BY that repartitions tuples by some specific key attribute. That means that tuples of specific value should be partitioned onto certain worker node. In this thesis such parts of query, split by stateful operators, will be called *stages*. It is assumed that their dependencies can be represented with *Directed Acyclic Graph* (DAG).

Should any of worker nodes break down, SPE implementation handles it by rescheduling it's load onto another worker. Since scalable system should be able to both decrease and increase number of nodes, a typical approach is to have a master that manages all the slaves. Such part of network not only can keep track of available resources, but also e.g. balance the load, keep track of data flow or designate worker that gets substream that failed to be processed by another one.

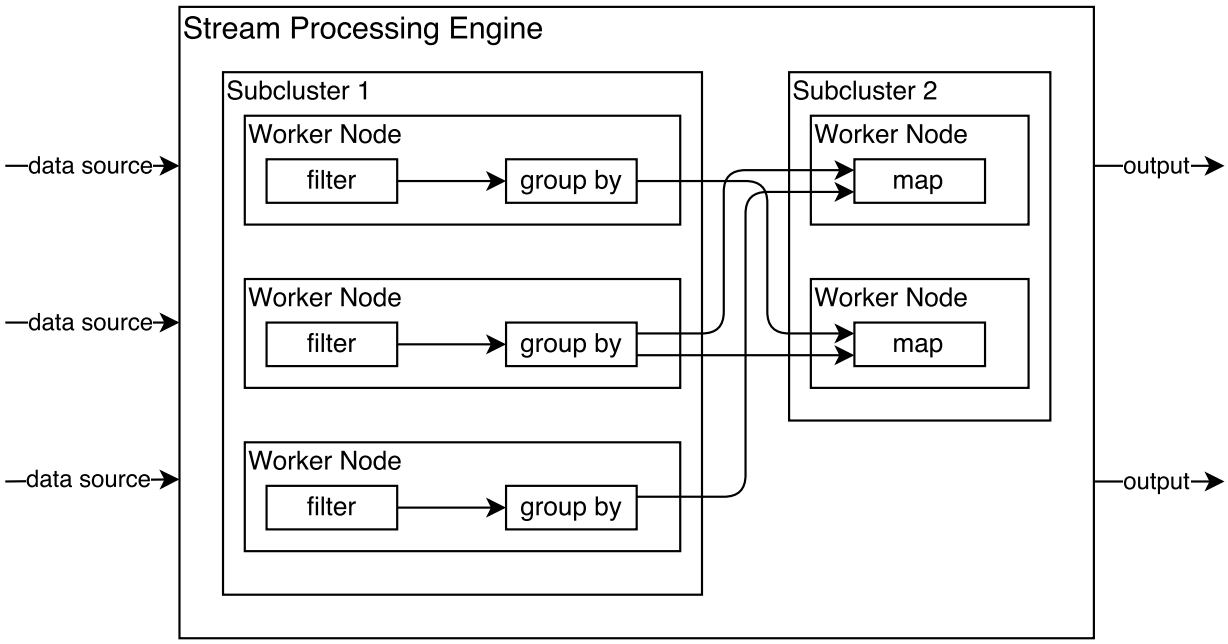


Figure 2.1: Sample query

Chapter 3

Spark Streaming

3.1 Apache Spark

Apache Spark [3] is a project started in 2009 on University of California, Berkeley. It is a cluster computing framework primarily targeted at large-scale data processing. Nowadays, in Big Data world it is the most active and prominent project. Programming interface is extremely rich with more than 80 operators available. One of the major advantages is caching data in each node's heap memory in order to avoid heavy network or I/O operations.

Each operator belongs to one of two categories:

- **Narrow** - Number of dataset partitions remains unchanged. All of the operations are performed at the same locality. Examples: *map*, *filter*.
- **Wide** - Number of dataset partitions may change. The operations may require some information exchange between cluster nodes. Examples: *groupBy*, *join*.

The computational model itself is divided into three tiers:

1. **Job** representation of the whole query ending with a single output destination
2. **Stage** part of the query up to nearest wide operator
3. **Task** stage query performed on a single partition

The key data structure used for processing is called **Resilient Distributed Dataset** (RDD) [18]. It is an abstract class representing distributed collection of elements that can be operated on in parallel. Due to immutability any transformation made on it returns a new set of partial results while utilizing in-memory efficiency, which means that I/O operations are not performed as long as there is enough stack memory to contain current data partition. By default every time an action is called, Spark goes through graph of RDDs and computes all the transformations in order to obtain specific output structure.

One can think about Spark as an inverted actor model. The actor model, similar in concept to query-based fine-grained SPEs, is represented by a set of threads where each contains it's own state and has a certain function capable of performing some data transformations and sending it further. Here it is inverted - data stays in place as much as possible and the functions are travelling over the network. This is a major reason for the performance impact since it decreases a lot of both network and I/O load.

3.2 D-Streams

Since this thesis is focused on Stream Processing Engines, from our perspective the most critical component of Apache Spark is *Spark Streaming* package. While most of SPEs are based on processing windows of tuples, Spark Streaming implementation instead reuses RDD concept under abstraction called *Discretized Stream* (D-Stream) on time intervals[19].

D-Stream is a moving window of RDDs. Each of them corresponds to data gathered during certain time interval - such dataset is called micro-batch. Whenever the window moves outside of certain data range, outdated partitions are dropped from system memory.

The project is by design a state-of-the-art SPE meeting all important requirements:

1. **Online evaluation.** For internal computations Spark Streaming is relying on Core package. Because of this it gets all the advantages like in-memory processing out of the box.

2. **Support of a high-level query language.** A programmer can use Scala, Java or Python in order to create processing pipeline. Many more languages are supported with unofficial support.

3. **Handling stream imperfections.** Whenever processing of certain time interval fails and gets out of order, it is rescheduled with higher priority.

4. **Determinism.** All of partitioning and processing operations are done in predictable manner without randomized subcomponents.

5. **Usage of past data sets.** A sliding window of batches can be established. Apart from that high-level languages provide an easy way of extending job's application with more complex tools. This means that even if a single RDD consists of data from recent five minutes it is still possible to analyze tuples from last six hours with complex machine learning algorithms.

6. **Fault tolerance.** Faulty tasks are rescheduled as stream imperfection. Their loads are distributed on existing worker nodes with higher priority. This is an easy and efficient process due to discrete nature of streaming data structure.

7. **Elasticity.** Dynamic resource allocation is available. Whenever ratio of processing time to batch duration gets out of fixed bounds, a number of executors is adjusted properly. This solution is profoundly imperfect and replacing it with a more solid one is the purpose of this thesis.

8. **Keeping latency constraint.** Because of lower bound for micro-batch time interval, the latency is usually higher than in fine-grained SPE systems. The optimal response time is equal to batch duration.

On Figure 3.1 a sample Spark Streaming application is presented. Each RDD consists of multiple partitions that are later transformed using narrow or wide operators. Each subset of operators is wrapped into entity known as stage. And in the end there are two jobs differed by output destination (all previous steps are shared).

It is worth noting here that apart from query job running on cluster there are also special jobs called *receivers*. Those are ongoing processes that end only when main application is stopped. Their responsibility is to obtain data from streaming jobs and distribute chunks of it as partitions across entire cluster.

A single flow of application query is known as **batch** and it represents base abstraction in streaming model.

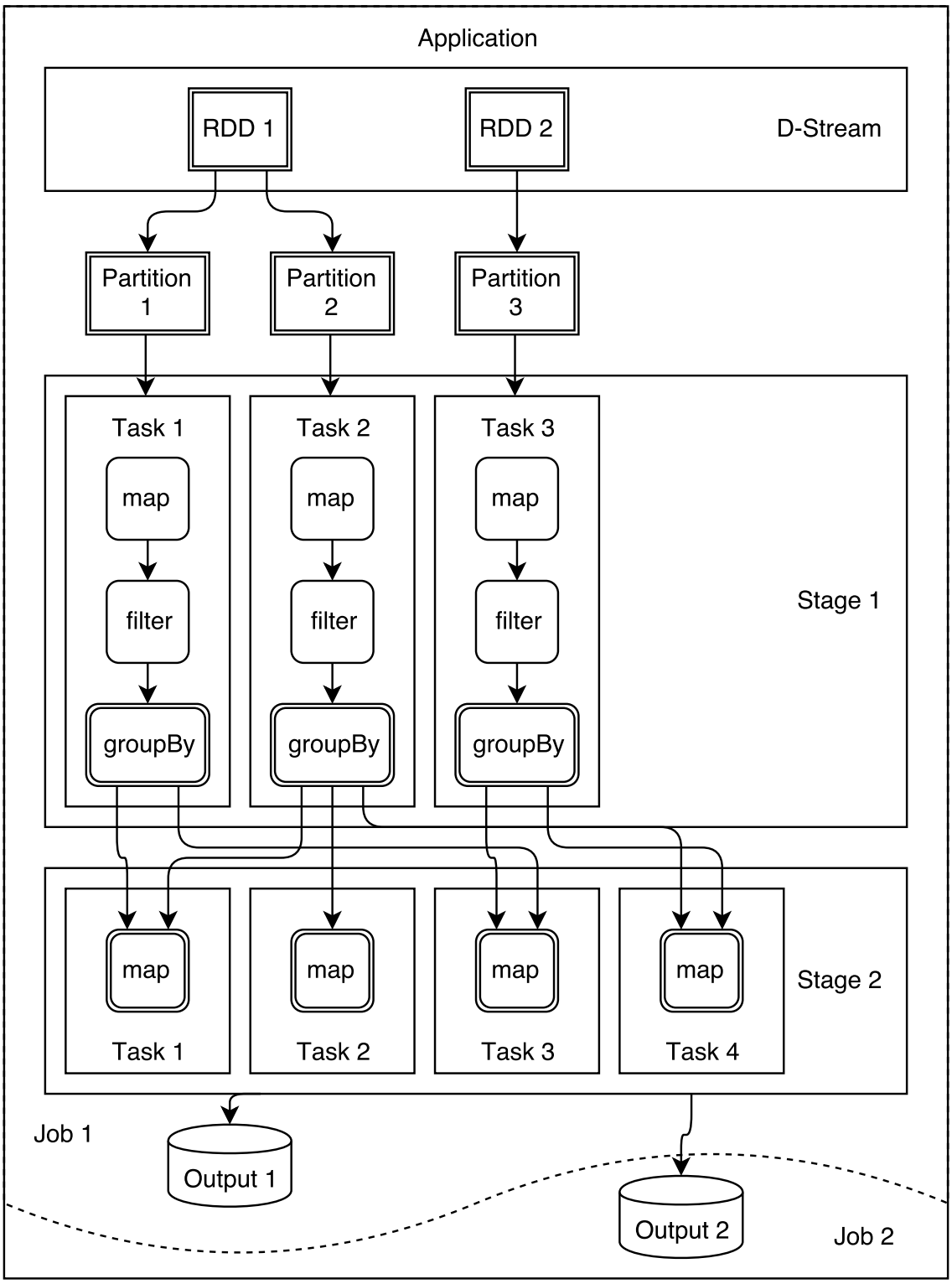


Figure 3.1: Sample Spark Streaming pipeline

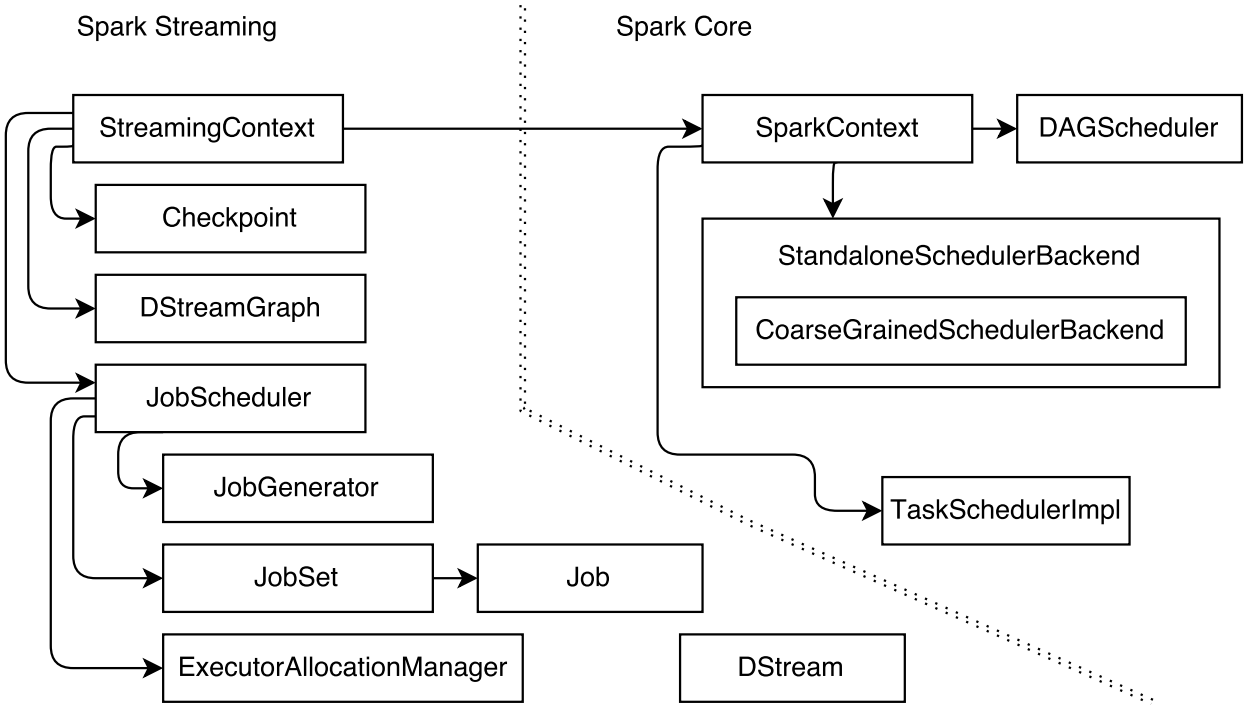


Figure 3.2: Spark components

3.3 System design

3.3.1 Cluster

The streaming subproject in Apache Spark reuses certain core components. Thus the cluster representation remains the same as in it's main part. It consists of four major entities:

- **Cluster manager** (also called **Master**) is a special node managing available resources on workers.
- **Worker Node** (also called **Slave**) is a machine providing CPU, memory and storage resources.
- **Driver program** is software containing the developed application that gets hooked up with Spark's runtime. It provides programming code which is later executed using available resources.
- **Executor** is a special JVM thread that is spawned on a worker in order to perform query evaluation on subset of partitioned data. By default each executor is responsible for processing a single task.

3.3.2 Components

On Figure 3.2 there is a simple visualization of Spark component entities. Those are the main classes supporting functionality that matters in this research.

- SparkContext
  The base class initializing runtime with immutable configuration and providing tools for processing pipeline creation. Essentially a client of Spark's execution environment.

- DAGScheduler
  Responsible for breaking job's tasks into stages and scheduling them on cluster resources. Stages, as well as tasks, are split by wide operators which require shuffling existing data partitions. In the end each task is sent to a single executor. Another important part of it's functionality is memory management. Keeps track of both cached and shuffled RDDs so there is no need of recomputing results. Maintains data locality - i.e. minimizes communication between different worker nodes.

- SchedulerBackend
  An interface for cluster backend. With default settings a CoarseGrainedSchedulerBackend is used in standalone mode (StandaloneSchedulerBackend). A major trait of this implementation is holding resources while a job is being processed, instead of bounding executor lifetime by task duration.

- TaskSchedulerImpl
  A class managing distribution of task. By default a FIFO (First-In-First-Out) order is maintained. At first all required tasks are run on cluster resources. Whenever any executor is released, it gets filled with next task waiting in the queue.

- StreamingContext
  Similar to SparkContext. Main purpose of its functionality is to initialize streaming elements.

- Checkpoint
  Manages DStream state backups on HDFS.

- DStreamGraph
  Handles job generation.

- JobScheduler
  Generates and schedules job on cluster. For each batch there are as many jobs as there are *sinks* (i.e. output destinations). Those are grouped inside JobSets that are later passed for pipeline processing.

- Job
  Describes application's data flow. It consists of tasks that are later divided into stages by Spark Core. Whenever a job fails it gets rescheduled with higher priority by JobScheduler.

- ExecutorAllocationManager
  An implementation of native Spark dynamic resource allocation solution.

3.3.3 Execution model

For purposes of this section, following query example will be used:

```
val lines = ssc.socketTextStream("localhost", 9999)
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(word => (word, 1))
val wordSums = wordCounts.reduceByKey(_ + _)
wordSums.print()
```

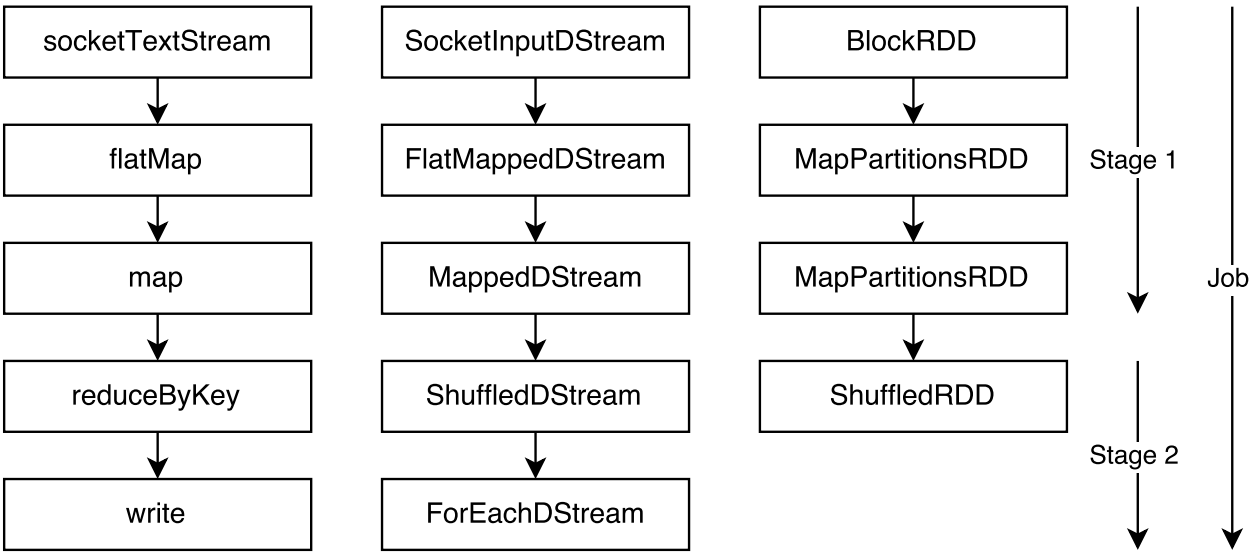


Figure 3.3: Query DAG

This job uses StreamingContext scc in order to create a stream receiver. Each line of input is split by blank space. Then partial output is grouped by word and number of occurrences is summed. In the end the results are printed to a command line. The entire pipeline is based on DStreams. When unwrapped following structure is presented:

```
ForEachDS[ShuffledDS[MappedDS[FlatMappedDS[SocketInputDS[String]...]
```

Each of DStream implementation classes corresponds to specific query operator - i.e. *.map(...)* is generating MappedDStream, *.socketTextStream(...)* is generating SocketInputDStream and so on. DAG representation of this query can be found on Figure 3.3. At first number of partitions is equal to ratio of micro-batch and receiver time intervals. This means that if receiver's block size is 200ms and streaming window is of 1s then it splits incoming stream into 5 partitions.

Application flow in Spark Streaming is pretty complex and reuses some of *Core* components. In order to understand it, a step-by-step analysis was performed (see Figure 3.4).

1. Blocks of data from a fixed time interval are united under a single RDD. Job queries are created for each desired output.
2. Each job from current set gets its own thread and is scheduled for processing.
3. Job's RDD function carrying out all the computations is generated and ran.
4. Partitioning is validated and job are split into stages on shuffle boundaries. All preceding stages must be completed. If this isn't the case then the current schedule is attached to waiting queue and previous sets of tasks are designated for computing. At last all necessary partitions are traced. The tasks are validated for serialization and submitted.
5. Schedulable pool of tasks is created and system checks for conflicts between stages.
6. A list of alive executors along with their number of free CPU cores is obtained.

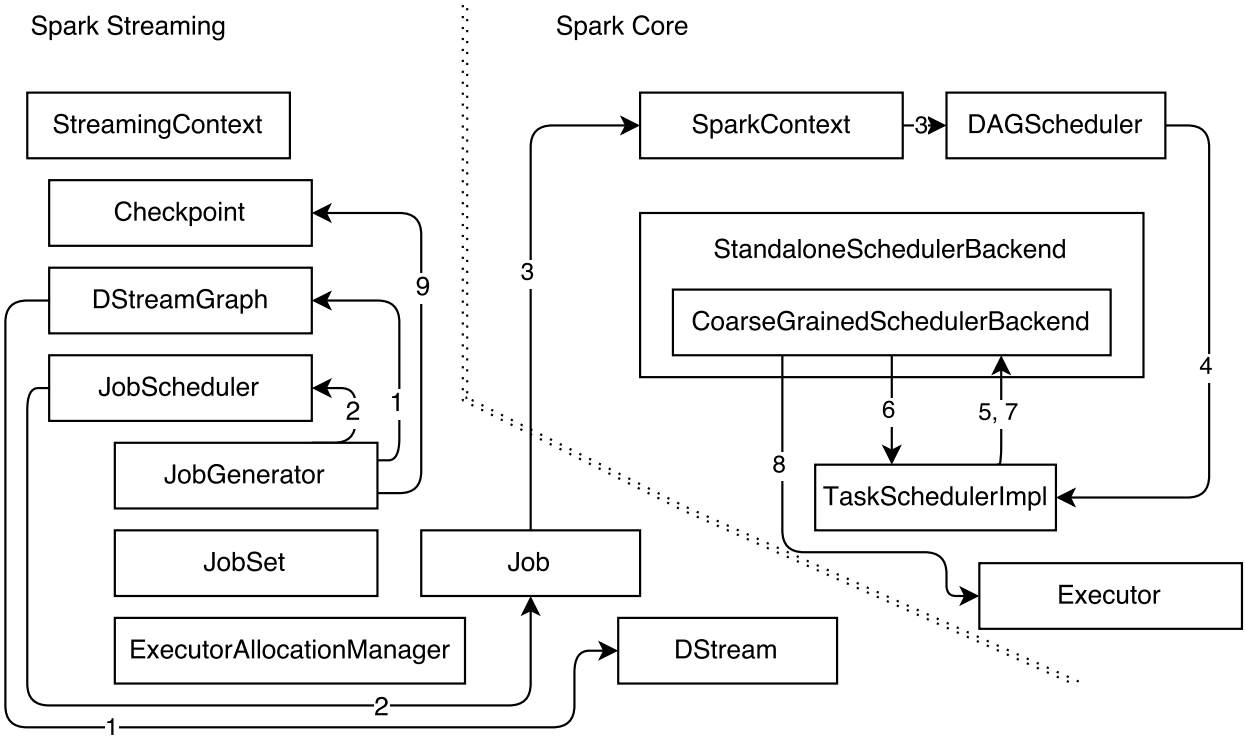


Figure 3.4: Execution model

7. Task scheduler sorts pending task sets (stages) by their time of arrival and priority. When this is done then using round-robin algorithm they are sent to the executors. It is important here that all tasks belonging to a single stage are run on lowest possible locality level. That is at first they are scheduled on a single worker node, later on machines belonging to the same rack and so on until level of entire cluster is allowed. This provides simple load balancing and lowers network traffic.

8. An RPC (Remote Procedure Call) with serialized tasks is sent to executor for processing.

9. In the end a checkpoint of current state is made and saved on HDFS (depending on application's configuration).

3.4 Resource management

Dynamic Resource Allocation for Spark Streaming has been recently introduced in version 2.0. Unfortunately due to it's instability it's nor recommended for professional usage and thus it is left undocumented. It provides a very simple implementation called ExecutorAllocationManager which is activated every n seconds (see Figure 3.5). The instantiation occurs in JobScheduler start method provided that the application doesn't run in local mode. During this time interval, information about all micro-batch processing duration is collected and a mean processing time is calculated upon activation. Then the ratio r of this and batch window time is taken and depending on the thresholds $low, high$ one of three following things might happen:

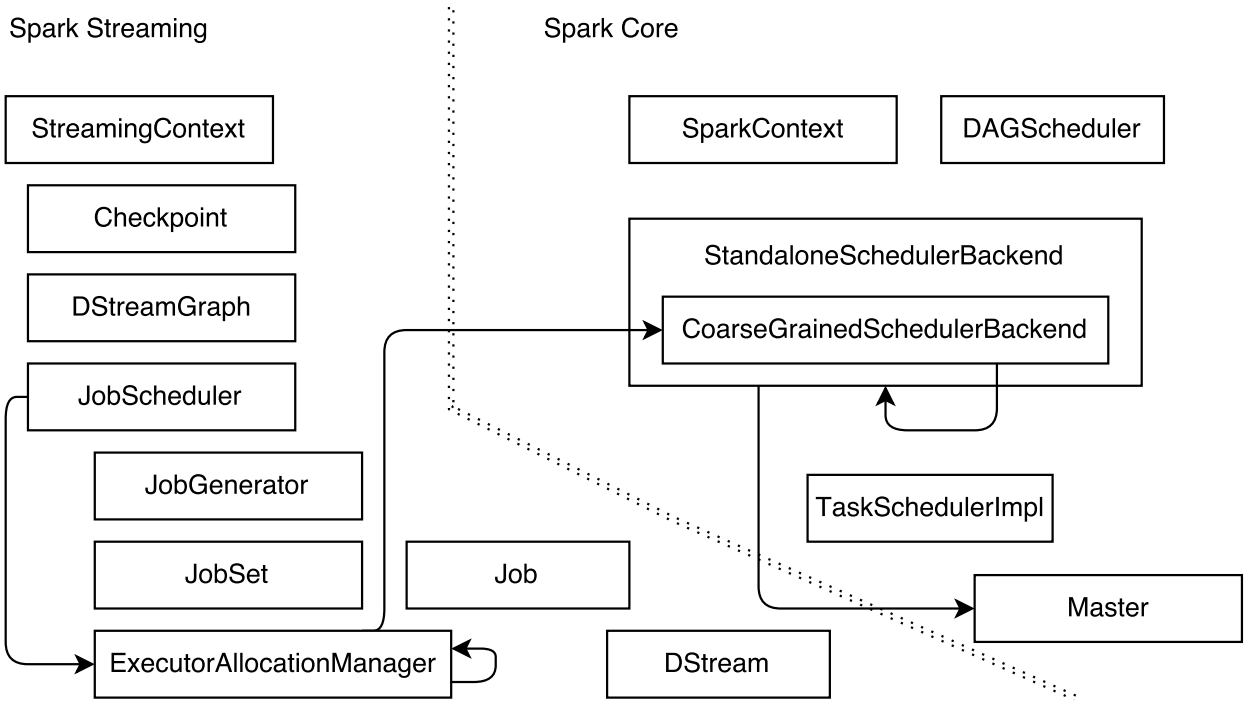


Figure 3.5: Dynamic Resource Allocation

1. $r \in [low, high]$ nothing happens
2. $r \leq low$ a single random executor, which isn't a receiver, is killed
3. $r \geq max$ $round(r)$ executors are requested from the backend

The calls are made by ExecutorAllocationClient which is one of the interfaces implemented by CoarseGrainedSchedulerBackend. If there is a request of adding few executors to the resource pool then the master node searches for the least occupied machine and allocates them. In other case the executor chosen to be removed is selected in a random way and later waits to be killed. That means that nothing happens until current batch is done being processed which in turns prevents system failure and redoing some of the current work.

While this is a fairly simplistic approach it lacks desired quality. For instance while taking a mean value of processing time from a certain duration, it totally neglects the variance parameter. This means that a fresh spike in resource usage won't change system behaviour until next activation of ExecutorAllocationManager.

Moreover the system neglects to take recent history into consideration. Each time interval may highly differ in terms of executors needed which prompts a lot of requests for resource changes. It's an easy way to destabilize the system since rapid changes highly disrupts locality of executors in the cluster.

There are a lot of solutions available for regular Stream Processing Engines [15]. Yet in case of Spark Streaming the approach needs to be reinvented in spite of it's innovative coarse-grained architecture.

Chapter 4

FUGU

4.1 Overview

FUGU is a state of the art elastic Stream Processing Engine[12][13]. Due to immense research effort put into establishing its online resource profiling model it was used here as the main source of ideas for creating a matching component in Spark's execution mechanism.

On a network level it is traditionally represented by a cluster of host machines and a master node. It is capable of performing both stateful and stateless operations. Dataset is processed in a fine grained way - i.e. a sliding window of incoming tuples is evaluated. As in Spark computation model, this system also processes query as a Directed Acyclic Graph (DAG). Each operator is assigned to a single worker and is responsible for performing a single query step. A single host might run several operators. The total number of hosts might change in time depending on cluster load.

Such approach is creating an elegant data flow through cluster network and meets the requirements of a real-time SPE:

1. **Online evaluation.** The data is processed as it comes without any predictable delay.
2. **Support of a high-level query language.** FUGU relies on Borealis[1] query language.
3. **Handling stream imperfections.** User is capable of adjusting the cluster so that it can handle faulty stream.
4. **Determinism.** All of partitioning and processing operations are done in predictable manner without randomized elements.
5. **Usage of past data sets.** Cluster keeps all data that belong to query's window.
6. **Fault tolerance.** The system is capable of load replaying which is triggered whenever an error occurs.
7. **Elasticity.** FUGU was designed with elastic scaling in mind. It has comprehensive approach which shall be discussed in the next section.
8. **Keeping latency constraint.** This is a real-time system which is not batch-based. Output is available as soon as it is processed which keeps latency at minimum level.

This system represents another iteration of classical Stream Processing Engine. It differs a lot from Spark project with continuous model of computation without using batch construct, poorer choice of query language and alternative cluster management. Even though it's a far less popular

Parameter	Range example	Description
Lower threshold	$\{0, 0.01, \ldots, 0.5\}$	Lower Bound (LB) for resource usage
Lower threshold duration	$\{3, 4, \ldots, 10\}$	LB breaches before cluster reconfigures
Upper threshold	$\{0.7, 0.76, \ldots, 0.9\}$	Upper Bound (UB) for resource usage
Upper threshold duration	$\{2, 3, 4\}$	UB breaches before cluster reconfigures
Grace period	$\{1, 2, \ldots, 5\}$	Minimum time between reconfigurations
Bin packing algorithm	{First-Fit, Last-Fit, ...}	Algorithm for operator placement

Table 4.1: Parameter space

tool for stream evaluation with significant performance loss it does have an optimization solution far more developed than most open-source systems.

4.2 Resource management

Early versions of FUGU were using reinforcement learning for estimating the amount of resources needed for query processing without violating latency constraint. It was later replaced with a white-box statistical learning model.

The optimization model is based on finding the best parameters promising optimal resource usage and preserving latency constraint. The list of the parameters can be found on Table 4.1. Each one is determined from a preconfigured range.

As in any statistical learning model the optimization component requires a training dataset. This is taken from utilization history. However only the meaningful part is taken into consideration.

Since intensity of incoming data might vary a lot, some information is useless for dynamic elasticity. For that reason a mechanism called Adaptive Window is used[6]. It keeps dynamically sized sliding window of past data based on total CPU usage by entire cluster. Whenever the system registers a steady period of resource usage it will cut the previous information giving the optimization model the most suitable set of training data.

Adapitive Window tries to add elements in one-by-one fashion and checks for variance disruption by investigating all possible cuts of existing window into two non-empty subwindows. Should any significant change occur, the older data is removed.

Cost function takes as parameters requested configuration and utilization history given by online profiler. It is later used by search algorithm for judging selected solutions. It checks the system behavior for them and either returns large value if latency constraint is violated or resource cost otherwise.

Search space consists of all possible parameter configurations given preconfigured ranges. Later on a Recursive Random Search heuristic is used to find most optimal setting. This phase relies on repeating two steps recursively until a locally the best solution is found. At first the algorithm randomly probes different parts of the search space. It calculates cost function for each such area and later shrinks it's the search space around the best result.

Chapter 5

Optimization system

5.1 Overview

The solution is loosely based on FUGU approach to resource management with respective to architecture of Apache Spark. Our model allows online parameter optimization that gets automatically adjusted to stream complexity and load.

During studies a lot of approaches have been considered. Some, like introduction of Queue Theory elements or Bayesian optimization were rejected due to either lack of time necessary to investigate them fully or uselessness against Spark Streaming model of computations[9][2].

Several ideas have been adopted though. Those include but are not limited to usage of Adaptive Windowing, parameter space based on preconfigured ranges or relying with the analysis on query DAG [6][13][15]. Moreover after careful study of Spark architecture, a special algorithm was determined for fast simulation of cluster behavior. This allows exact prediction of how any resource change would affect stream processing.

5.2 Approaches

The major obstacle in this work is that the system under work is not built from scratch. Therefore it is not possible to presume certain optimization model in advance and simply adjust components of SPE to handle it. Overall several research projects have been analyzed with purpose of extracting promising ideas.

5.2.1 Trigger

The current optimization model embedded in original Spark code is not a production grade solution as discussed in Section 3.4. Nevertheless Databricks, a company overseeing it's development, figured out a partial solution to some business issues[7]. It's targeting a case where a company has a lot of cluster resources but their stream of data is not that demanding. Moreover there is nearly no need of processing incoming data as it comes. For that purpose a special trigger can be activated once a day or so in order to evaluate all of unprocessed stream that was received by a fraction of available worker nodes. In other words small part of the cluster is working all the time accepting incoming data but the processing jobs are activated only when it is worth it. This solution is targeting very specific scenario where there indeed might be a need of performing a data analysis on a large stream dataset but with incremental updates of smaller yet not negligible interval. Since our solution is targeting more real-time systems, this idea was dismissed.

5.2.2 Queue theory

SPE can be also reviewed under Queue Theory model. It is fair to say that there is a queue of tuples waiting to be processed by several workers. This can be treated as G/G/k queue with general (arbitrary) distribution of rate of incoming messages, general distribution of processing time and k servers. Such model is not easy to work with but considering Spark architecture few simplifications may be applied.

First of all it is not possible to work with the cluster on atomic level, checking every single worker and strictly monitoring the flow of data. Incoming information is gathered inside a micro-batch that is executed by every slave node in parallel. Therefore this collapses the problem into G/G/1 queue - we can treat entire cluster as a single service handling incoming batch entities. We no longer work on a tuple level but rather on a larger macroscale.

The problem with general distribution can be targeted in several ways. The most popular one would be to use the Kingman's formula as in model created in early stages of Apache Flink project development[15]. It is part of it's latency model but the G/G/1 queue is used on a lower level. In a way similar to Spark, stream processing flow is represented by a DAG of stages and tasks. Each stage vertex represents a set of tasks performed with determined level of parallelism. Now a single task vertex is treated as a single server queueing system. This allows the model to estimate stage-local waiting time independently of amount of available resources - i.e. independently of how much it is possible to parallelize single stage.

Later on this tool is used in order to maintain latency constraint when optimization model is using linear objective function for finding a solution with lowest possible degree of total parallelism. Should the algorithm find a more optimal solution than the current one, the cluster is scheduled for reconfiguration. This might sound like a right step in the right direction. Unfortunately authors of the article made several assumptions that didn't seem to be convincing. For instance there is no clue what happens when the queue becomes unstable.

5.2.3 Reinforcement Learning

Original model lying behind FUGU optimization subsystem was using a State-Action-Reward-State-Action (SARSA) algorithm in order to determine whether cluster reconfiguration should be made[12]. It uses the utilization history together with two manually fixed parameters: learning rate and importance. Nevertheless this approach was tested against newer white box model and while it proposed cheaper configurations, it also violated latency constraint more significantly[13].

5.3 Model

Our model follows white-box statistical learning approach. It is based on a assumption that entire cluster is heterogeneous - i.e. each worker node brings exactly the same resources. The amount of resources needed is calculated in a deterministic way. Entities of Spark query DAG are somewhat complex. This work simplifies it with a requirement that each executor thread should be performing at most one task at a time.

As in FUGU, the resource management consists of both static and dynamic parameters, cost function and optimization algorithm. Each of those modules shall be discussed separately. While some of the choices are quite obvious, it was not easy to introduce them as the Spark system was not developed with dynamic elasticity in mind.

This model represents local optimization. It only runs within a single application and is not responsible for master node actions. It's a result of choosing a way of implementing the solution that does not affect upstream Spark codebase. The default cluster manager that was kept in mind during designing process is Spark Standalone. Further details will be explained in Section 5.4.

5.3.1 Parameter space

Due to constraints put on the solution by Spark architecture not all of the parameters can be automatically adjusted in online time. They are going to be divided between static (the ones that have constant value) and dynamic (the ones which value will be calculated during application's life).

Static parameters:

1. Minimal and maximal number of executors required by resource manager in total. This includes receivers.
2. Granularity of changes determining minimum shift in number of executors. When set, the manager won't kill less than n executors and will add at least n executors if corresponding change is required.
3. Adaptive window parameter δ which influences sensitivity of this algorithm. After adding an element to the window, if it doesn't change mean value, this parameter serves as the upper bound for probability of performing shrink operation. Given two consecutive subwindows if their mean value differs significantly this parameter determines probability of dropping the older one.
4. Number of executors to keep in advance. The optimization part is very precise about the optimal number of executors. Removing some from the cluster is a quick operation however adding more involves spawning further JVM threads which is an expensive operation.

Dynamic parameters:

5. Range of upper threshold for the number of breaches which determines sensitivity of latency constraint. The system analyses past batches and for given number of executors it determines number of such breaches. Nevertheless the system should allow some elasticity in spite of not overfitting the statistical model.
6. Optimal number of executors running in the cluster and processing the application. It is determined by optimization algorithm with constant bounds.
7. Adaptive window size. It depends on three factors:
   - Fixed δ value determining change sensitivity
   - Highest load value from all available batch history serving as a ratio normalizing all the values into range $[0, 1]$.
   - Due to stability issues adaptation of window size is considered only after several batches arrive into the system. Their number depends on the batch history length limit, which is set programmatically. The later reflects on how fast the system can adopt. If it is too low then cluster might succumb to perform many changes in a fast pace.

#	Configuration key	Value example
1.	minExecutors	2
1.	maxExecutors	20
2.	granularity	2
3.	adaptiveWindow.delta	0.5
4.	backupExecutors	2
5.	thresholdBreaches	$[2, 4, \ldots, 128]$

Table 5.1: Preconfigured parameters

In relation to previously mentioned optimization models this one drops few parameters:

- **Spark**
  Each optimization step is considered after successful processing of a batch thus removing independent time interval for checking optimal configuration. It is particularly efficient because this way the cluster resources are adjusted before entire query processing is performed. The method determining number of executors is flexible and because of that lower and upper bounds of processing time are not needed in this model.

- **FUGU**
  Grace period is not needed since killing executors is done in a soft manner. As for bin packing algorithm the Spark Core only supports two algorithms: FIFO (default) and FAIR. Due to implementation complexity for now are only the first one is considered.

5.3.2 Simulator

The utilization history must be somehow evaluated with prospected number of executors. In FUGU this was done by assessing amount of resources needed by a single query operator and using it to distribute them onto minimal number of hosts using subset sum heuristics. In Spark this can't be done as the query at lowest level is processed by stage part which can be executed on arbitrary host machine.

Other solutions try estimating amount of CPU time needed for processing a single record from batch and then scaling it for given load. This however is a loose model that is subjected to a lot of uncorrelated factors such as random failures within cluster or randomness of data partitioning.

After careful analysis of Spark execution model it was determined that batch execution can be well simulated given information about all of its tiers. For that matter a special component in the system was created that registers entire batch structure - that is information about batch itself, jobs, stages and tiers. Together it forms a tree structure visualized on figure 5.1.

This thesis is focused on FIFO (First In First Out) scheduling order. Batches and jobs are processed in strictly sequential way which means that their evaluation is not performed in parallel at all. This however does not apply for models used on other tiers:

- Stages are split into levels. They are submitted to the queue only after their parent stages are already processed. However several independent stages might be scheduled at a single moment.

- Tasks are queued in a strictly pure FIFO fashion. When a stage is submitted the queue is populated with its tasks. Then they undergo processing on cluster's executors. If there are not enough resources to begin evaluating all the tasks from the queue simultaneously the first

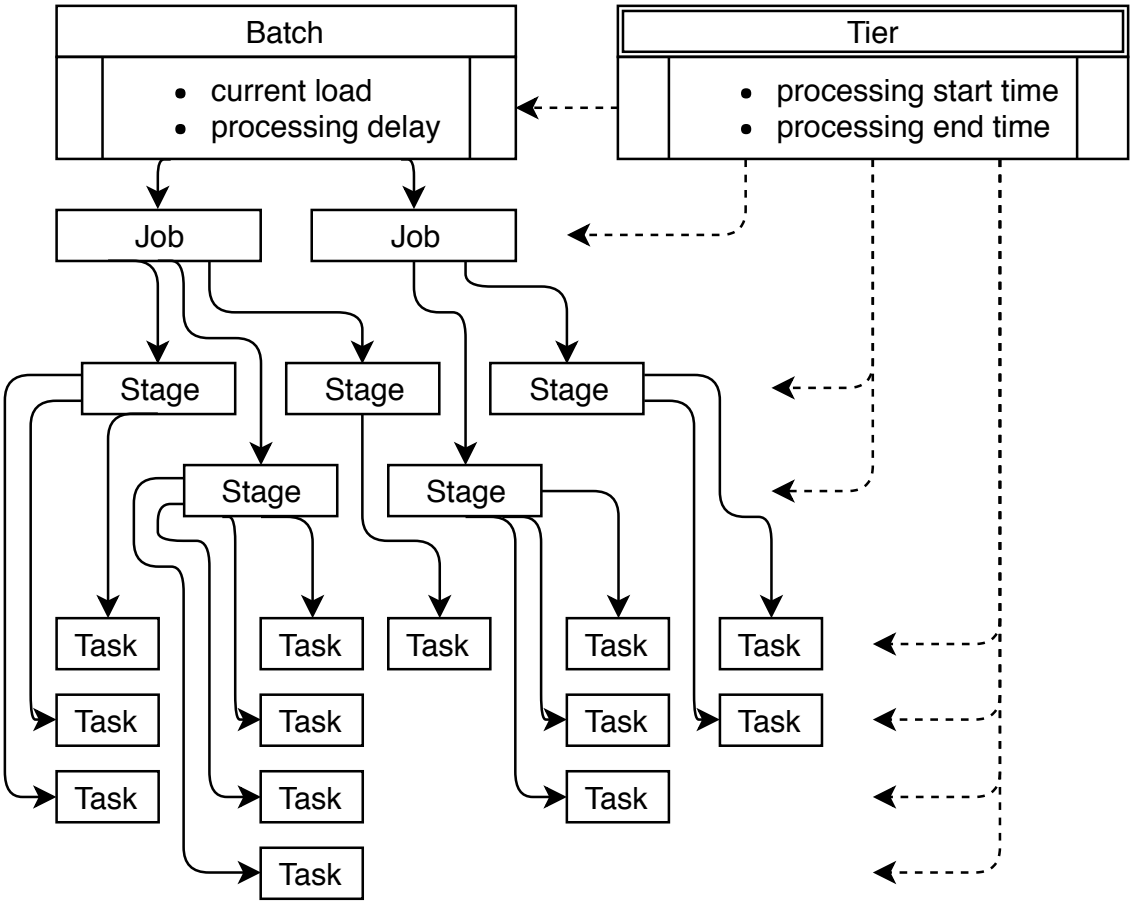


Figure 5.1: Tier tree

executor which finished processing is scheduled with queue's head. This step is repeated until the queue is empty.

The simulator component is divided into two parts. One is responsible for calculating all the time between jobs and stage levels that left cluster idle. This is done in order to include in simulation time spent on performing I/O and network operations which are dependable on batch's load. It is a simple algorithm returning cachable results and won't be discussed further. The other part's purpose is to simulate job execution. See Algorithms 1 - 3.

Algorithm 1 Job simulation

```
1: procedure SIMULATE(job: Tier, executorNo: Integer)
2:     stageCompletion ← Map[Stage, Time]()          ▷ Default value for any key is 0
3:     executors ← PriorityQueue[Time](executorNo)   ▷ Default value is 0
4:     for all stage ∈ job do
5:         startTime ← max(stageCompletion(parent ∈ stage.parents))
6:         stageCompletion(stage) ← simulate(stage, startTime, executors)
7:     end for
8:     return max(executors)
9: end procedure
```

Algorithm 2 Stage simulation

```
1: procedure SIMULATE(stage: Tier, startTime: Integer, executors: PriorityQueue[Time])
2:     stopTimes ← Array[Time](stage.tasks.length)   ▷ Default value is 0
3:     for all task ∈ stage do
4:         stopTimes(task.index) ← simulate(task, startTime, executors)
5:     end for
6:     return max(stopTimes)
7: end procedure
```

Algorithm 3 Task simulation

```
1: procedure SIMULATE(task: Tier, stageStartTime: Time, executors: PriorityQueue[Time])
2:     executorTime ← executors.dequeue()
3:     startTime ← max(executorTime, stageStartTime)
4:     stopTime ← startTime + task.duration
5:     executors.enqueue(stopTime)
6:     return stopTime
7: end procedure
```

Let s be number of stages within job, t_i be a number of tasks within i-th stage and n be a number of executors. Big $\mathcal{O}$ notation is used since each batch consists of nonempty set of jobs, stages and usually nonempty set of tasks. Thus the worst case scenario is the most common one.

Algorithm 3 has time complexity of $\mathcal{O}(\log n)$ due to operations of removing and adding elements to the priority queue of executors. Later in Algorithm 2 this step is repeated for each task in a stage. This gives time complexity $\mathcal{O}(1 + t_i \log n)$ since access to array costs $\mathcal{O}(1)$ and stage might be empty. Again this level of simulation process is repeated in Algorithm 1 for all stages belonging to

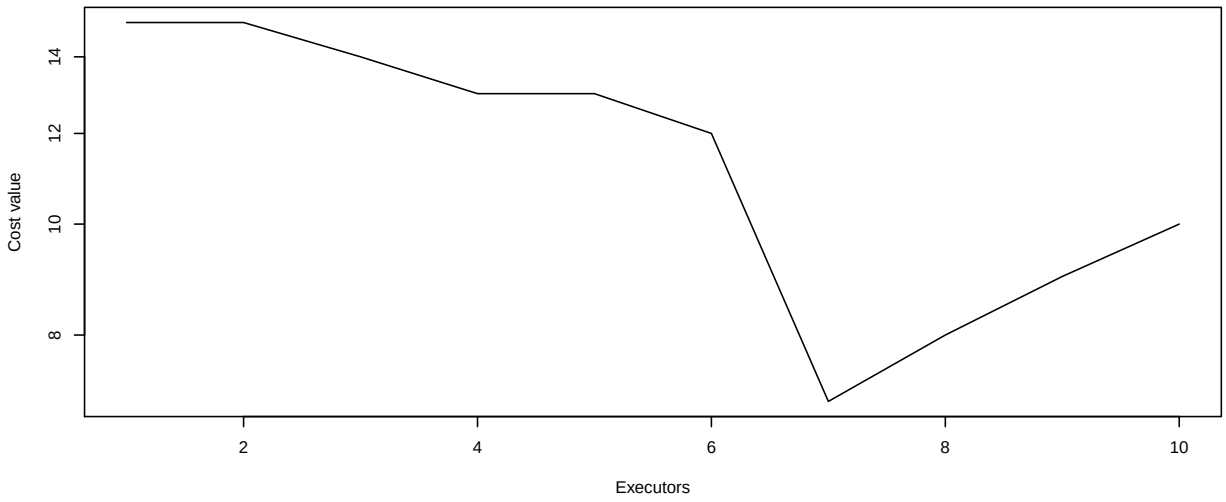


Figure 5.2: Sample cost function

the job. The result is maximal executor time since next job won't be processed until the previous one is finished. The queue is containing lowest element on the top and because of that it takes $\mathcal{O}(n)$ steps to find the higest value. In the end such simulation costs

$$\mathcal{O}(\sum_{i=1}^{s}(1 + t_i \log n) + n) = \mathcal{O}(n + s + t \log n),\ t = \sum_{i=1}^{s} t_i$$

under assumption that access to Map has $\mathcal{O}(1)$ complexity. It can be said that in regular case with no empty stages the complexity is equal to $\mathcal{O}(n + t \log n)$.

The simulator component is summing all the results for each job belonging to the batch since their processing is done in a consecutive way. Due to close resemblance to original Spark execution model the results are very close to real ones and it can be used in optimization part.

5.3.3 Cost function

Given number of executors and a batch window the cost function uses simulator to get expected times of completion. Then it calculates all the delays.

If processing a batch took longer than batch interval then the next batch is delayed by the difference between those times. If batch after that was processed within duration of batch interval then next delay will be equal to $max(0, time - interval + delay)$. This is going to affect next batches until delay is eliminated.

Each delay greater than 0 is treated as latency threshold breach. If total number of those breaches is lower than optimization parameter then number of executors is returned. Otherwise the function returns sum of latency violations and maximal number of executors.

This function may be split into two two consecutive parts (see example from Figure 5.2):

1. The beginning of the function is non-increasing due to the fact that adding another executor to simulator will never make the result worse.

2. The part following is a simple identity function.

Let m be a number of batches, t_j total number of tasks and s_j total number of stages that j-th batch consists of. Big $\mathcal{O}$ notation is used for the same reasons as before. Batches are evaluated

in consecutive manner, same as for jobs. Then the time complexity of calculating the cost is equal to:

$$\mathcal{O}(\sum_{j=1}^{m}(n + s_j + t_j \log n)) = \mathcal{O}(nm + S + T \log n),\ T = \sum_{j=1}^{m} t_j,\ S = \sum_{j=1}^{m} s_j$$

5.3.4 Search algorithm

The search algorithm always wants to determine the lowest number of executors that does not violate latency constraint. Because the cost function has exactly one local minimum that is also the global one, this can be done by finding the element of lowest value that satisfies desired conditions. For that reason a customized version of tail-recursive binary search is used, see Algorithm 4. The initial range is bounded by lowest and highest number of executors minus number of receivers.

Algorithm 4 Binary search

```
1:  procedure SEARCH(begin, end)
2:      if begin < end then
3:          middle ← ⌊(begin+end)/2⌋
4:          if cost(middle) > maxExecutors then
5:              search(middle + 1, end)
6:          else
7:              search(begin, middle)
8:          end if
9:      else
10:         return end
11:     end if
12: end procedure
```

Additionally it determines other dynamic parameters. The number of batches m analyzed by cost function is equal to the size of adaptive window after adding entire history from application's buffer. Then the upper bound on threshold breaches is dependent on ratio between size of window and length of the buffer.

5.3.5 Request handling

If there is a need for changing amount of cluster resources the manager tries to do this gradually in $\mathcal{O}(\log c)$ steps, where c is the executor change and each move depends on cluster granularity defined in the configuration. Moreover it takes into consideration requested number of backup executors by adding it to optimal number returned by the search algorithm. The formula can be found in Algorithm 5.

Algorithm 5 calculateExecutorChange

```
1: procedure CALCULATE(optimal: Integer, proposed: Integer)
2:     change ← min(optimal + backup, maximum) − workingExecutors
3:     softChange ← round((|change| + (granularity − 1) ∗ min(sgn(change), 0))/2.0)
4:     return sgn(change) ∗ ⌈softChange/granularity⌉ ∗ granularity
5: end procedure
```

In case some executors are to be killed they are picked at random. It would be better if this was done by choosing the worker node with lowest amount of active resources in order to preserve computation locality. This information is unfortunately unavailable.

When asking for executors the cluster manager decides where to spawn them. In Spark Standalone this is done by choosing the worker node with highest number of free CPU cores. This process is repeated until all executors are spawned.

5.4 Implementation

Before being able to come up with any ideas it was vital to get familiar with Spark's design and code. Each of later steps was taken after making an analysis whether it fits into general execution model. Due to lack of any official documentation of system's architecture it was necessary to browse through original codebase. Most of the findings were already discussed in Sector 3. Since the project was written as an extension to the original system, the tools we use aim to follow original solutions.

5.4.1 Programming language

The language of choice for this Thesis was **Scala**. It is a fairly modern programming language developed on EPFL by Martin Odersky. By design it merges majority of concepts from both object oriented and functional programming. Currently it is used by top companies in projects that target distributed and concurrent applications[10][16]. It also serves as the main implementation language for Apache Spark. Because of those facts it is one of the top tools used in Big Data industry.

There are also few chunks of code written in **Java** imported from FUGU. Since both those languages run natively on JVM there was no need to rewrite them. The entire build process was handled by SBT.

5.4.2 Development

Several popular tools were used for checking code validity. Almost 70% of entire application's logic is covered with solid tests. Upon committing new changes the code style was checked using scalafmt formatting library and, if successful and sent to continuous integration server where an automated build process occurred. Moreover the coverage was monitored all the time with another tool tracking any outdated dependencies. Git version control system was used at all times.

5.4.3 Spark compatibility

Several steps were made in order to maximize compatibility with upstream code and guarantee lowest maintenance costs. First of all the solution was made based on Spark Standalone cluster manager. It is the default solution that is the easiest to set up. Apart from that the configuration of new optimization component is overriding the existing one. Scala version matches the officially supported one.

5.4.4 System architecture

The system design scheme can be found on Figure 5.3. It is composed of elements that are cleanly interconnected. The major component that is exposed to user interaction is StreamingContext

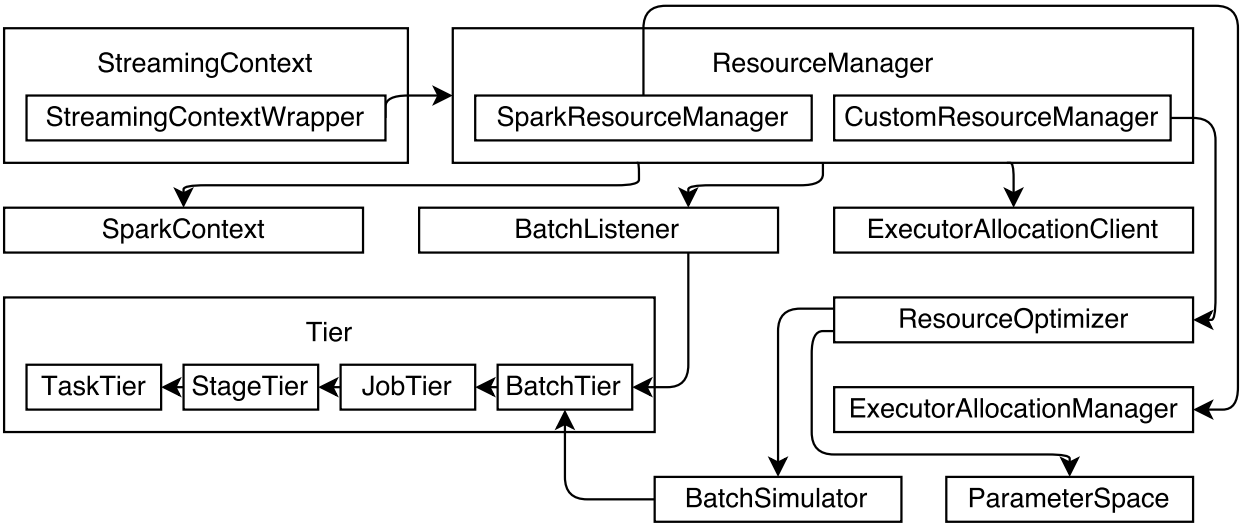


Figure 5.3: System architecture

discussed earlier in Section 3. Descriptions for further elements can be found below:

- StreamingContextWrapper
  Inherits from original StreamingContext and adds functionality of handling ResourceManager of user's choice. Overrides default start and stop routines in order to hook up the optimization part. Prevents activation of ExecutorAllocationManager.
- ResourceManager
  Exposes internal interface for tracking and managing executors through ExecutorAllocationClient. Moreover reads configuration parameters.
- SparkResourceManager
  Encapsulation of default upstream solution. Added mainly for testing purposes.
- CustomResourceManager
  Manager of optimization model discussed in this chapter. Whenever a change is proposed it checks whether it is valid and, if so, asks the cluster for expected amount of resources.
- BatchListener
  Registers entire batch composition with it's jobs, stages and tasks. Utilizes RPC event listeners. Allows other components to get detailed history of stream processing.
- Tier
  Interface containing common logic. Using F-bounded polymorphism it enables sorting for all tiers. All classes implementing it must containg an id, processing start time and processing stop time. Every tier except TaskTier has references to lower ones.
- ResourceOptimizer
  Determines optimal number of working executors that promises to preserve latency constraint while lowering resource usage costs. It utilizes simple binary-search-like algorithm for calculating the result. For a proposed amount of resources it uses BatchSimulator to get predicted processing delay.

- ParameterSpace
  Plain structure determining dynamic parameters.

- BatchSimulator
  Given number of executors it simulates requested BatchTier. Takes into consideration missing time between tiers from the same level by adding it into general result. All such calculations are cached for a limited time.

The whole project is fully relying on stable elements of original Spark project. It is not changing the way the cluster evaluates information. Moreover it can be attached to a project with a simple library that is small in size - which reduces both the build time for user's streaming application and serialization process.

When an executor is killed it is done in a smooth way - it means that resources are released only when they are not needed any more via a soft kill. In that way cluster behaviour is not interrupted by changes made by optimizer and it does not inflict running jobs. The only waiting time occurs when spawning new JVM instances for fresh executors. Because of that configuration of the parameters should be done in a preservative manner, as frequent changes might destabilize the application.

Each major step is fully logged by default utility. This gives user ability to browse through history of project's behaviour and make an attempt at further tailoring it to processed dataset. It would require pausing the application. Nevertheless it is possible to modify the code with dynamic read of input parameters. However for business needs this is not advised since such approach would result in higher volatility.

Chapter 6

Evaluation

6.1 Testing application

A simple stream based on Apache Kafka capabilities was created in order to simulate real-world example. Such tool has two parts - a producer sending the stream and a consumer receiving it.

6.1.1 Producer

With Kafka each stream is represented by a *topic*. Moreover each topic can be represented by multiple *partitions*. It means simply that when a consumer hooks up with a producer, it can receive a single stream via multiple sockets in parallel.

It was not necessary to create a complex solution here. A simple utility reading comments from file and streaming them under a single topic with a single partition was enough for needs of the evaluation process. There is a loop spawning a new JVM thread that sends specified number of comments written within some constant time interval. Here the application was streaming 5 minutes of data per second which was large enough for triggering optimization software and small enough to fit network bandwidth.

6.1.2 Consumer

A Spark application utilizing Kafka stream receiver was used. The running query used a time window of 72 seconds with batch duration lasting 3 seconds. That means the application was performing an analysis on 6 hours of comments with 15 minutes step. The complete query can be found on Listing 6.1.

Listing 6.1: Test query

```
val t1 = allComments map { comment =>
  (comment.author, comment)
}

val t2 = t1 reduceByKey { (c1, c2) =>
  val words = (c1.words ++ c2.words)
    .filter(_.nonEmpty)
    .groupBy(identity)
    .mapValues(_.length)
```

#	Month	Size	Number of comments
1.	January	27GB	48342747
2.	February	30GB	53851542

Table 6.1: Datasets

```
      .toSeq
      .sortBy(-_._2)
      .map(_._1)

    Comment(
      c1.author,
      words,
      (c1.subs ++ c2.subs),
      c1.upsSum + c2.downsSum,
      c1.downsSum + c2.downsSum
    )
}

val t3 = t2 map {
  case (_, aggregatedComment) => aggregatedComment
} filter { c =>
  c.upsSum > c.downsSum
} map { c =>
  c.toString
}

t3.print()
```

6.2 Dataset

The tests were performed based on two separate, single-filed datasets. They were containing Reddit comments from January and February 2015 respectively[5]. Each comment was represented by a JSON structure with information about author, comment content, time of writing, number of upvotes and downvotes, etc. Specification of both files can be found on Table 6.1. It is assumed that such source is randomized enough for purpose of performing the tests.

Moreover when looking upon Figure 6.1 it might be noticed that the stream in time changes like a sine wave. This produces enough variation for the optimization model to make some shifts in resource usage over application's lifecycle.

6.3 Environment

The solution was tested on 6 Amazon AWS EC2 m4-xlarge instances. Each such instance consists of 4 CPUs, 16 GB of RAM and runs Ubuntu 16.04 LTS. Expected network throughput is 93.75 MB/s. One of those instances served as a master node together with a Kafka producer. The other ones

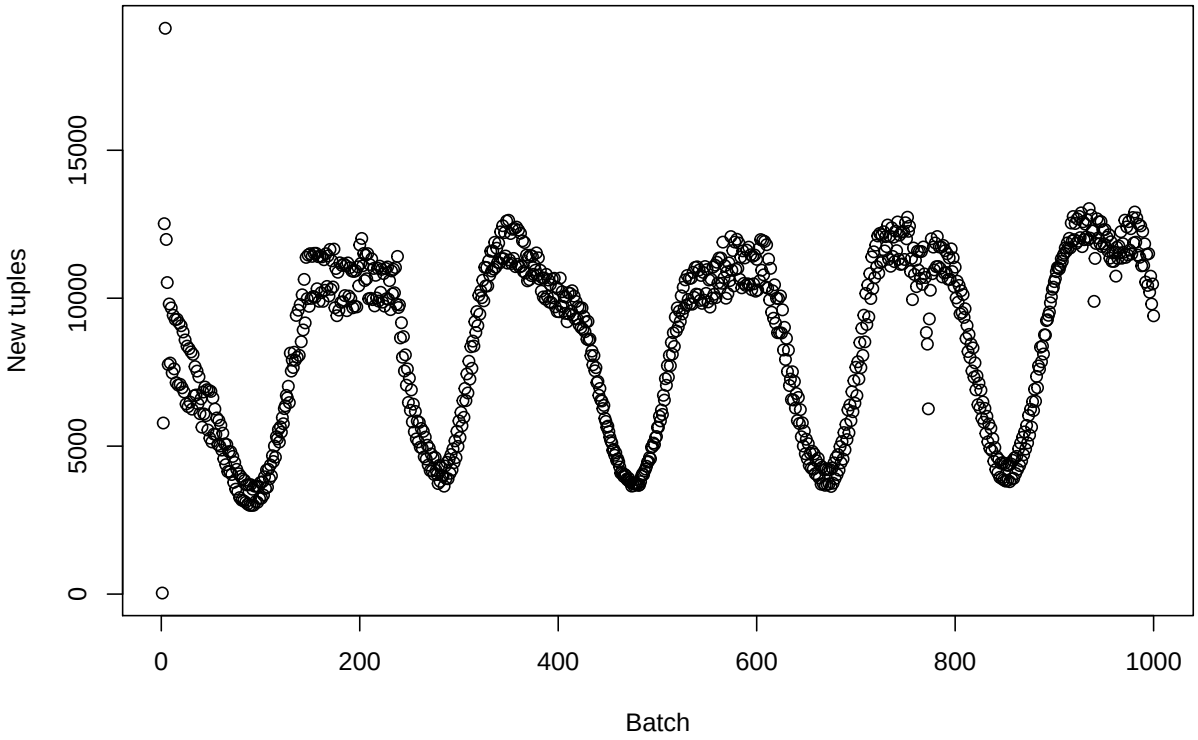


Figure 6.1: Stream load from January 2015

were used as workers. Each test scenario lasted 50 minutes, was repeated twice and average result was taken. The results consist of data from 1000 batches.

Each executor was running on a single CPU core with each task running on a single executor. There was also a need for changing the locality constrain. By default, if a task can't be ran on the same locality as other tasks, the cluster manager waits 3 seconds before allowing it to be processed somewhere else. Since the cluster is fast enough with enterprise-level support from AWS, it was acceptable to forego this constraint. Spark configuration can be found on Table 6.2. Some parameters however were modified with each scenario. Those are corresponding to the rest of static factors of the optimization model.

The default solution used in upstream Spark codebase was tested using two configuration sets: the default one and another adopted to stream's distribution model with more flexibility (see Table

Parameter	Value
CPUs per task	1
Maximal number of CPUs	20
CPUs per executor	1
Memory per executor	2 GB
Miminal number of executors	2
Maximal number of executors	20
Backup executors	2
Time before loosing locality constraint	1 ms

Table 6.2: Spark configuration

Parameter	Default	Custom
Scaling interval	60	30
Batch interval LB	0.3	0.45
Batch interval UB	0.9	0.85

Table 6.3: Spark optimization model configuration

Parameter	Default	Underfitting	Overfitting
Granularity	2	4	1
Threshold breaches	$[2, 6, \ldots, 254]$	$[0, 4, \ldots, 128]$	$[4, 12, \ldots, 252]$
Adaptive Window delta	0.5	0.25	0.75

Table 6.4: Thesis optimization model configuration

6.3). The latter simply narrows the utilization bounds by 15% from each side with scaling interval being reduced in half.

The custom solution created in this thesis was tested using three configuration sets: the default, the underfitting and the overfitting configuration (see Table 6.4). The default one was established with more intuitive parameters - the granularity is equal to 10% of cluster resources and Adaptive Window delta is the same as in the FUGU Stream Processing Engine. On the other hand the underfitting one is using values that are two times more restrictive and the overfitting is twice more permissive. This ensures getting resulsts from some spectrum of possible approaches.

6.4 Results

Each of the results contains general information about statistics such as mean or median together with number of times latency constraint was breached. For duration it says how often it took longer than batch interval to process the data. For delay it sums all the values greater than 0.

6.4.1 January dataset

Spark

The default configuration (see Table 6.5) resulted in a static behaviour where there is nearly no change made upon cluster resources. The difference between two attempts may be explained by tiny shifts in stream information of each batch. Only 1% of all batches was subjected to latency

	Duration [ms]	Delay [ms]	Executors
Minimum	384.5	0	19.5
Lower quartile	1243.9	0	19.50
Median	1739.8	0.5	19.5
Mean	1633.1	23.79	19.55
Upper quartile	1928.8	0.5	19.5
Maximum	8306	6381	20
Breaches	2	10	N/A

Table 6.5: Default configuration (Spark, January)

	Duration [ms]	Delay [ms]	Executors
Minimum	385	0	13
Lower quartile	1441	0	14.5
Median	1936	0.5	15
Mean	1964	564.9	15.77
Upper quartile	2203	1.5	17
Maximum	8798	8924.5	20
Breaches	21	145	N/A

Table 6.6: Custom configuration (Spark, January)

	Duration [ms]	Delay [ms]	Executors
Minimum	419.5	0	10
Lower quartile	1468.5	0	12
Median	1971.8	0	13
Mean	1953.6	164.8	14.33
Upper quartile	2311.1	0.5	18
Maximum	4049	3754	20
Breaches	40	131	N/A

Table 6.7: Default configuration (Thesis, January)

violation. The reason for that is the way the optimization components kills executors. It sometimes does that in the middle of processing so that cluster must wait for all the resources to be flushed from designated executors. This results in latency spikes that can be noticed on Figure 6.2. It is observable that the outliers in duration time occur whenever the stream intensity gets low and manager starts killing executors. In result upper quartile is significantly lower than mean. For reference see Figure 6.3.

The custom configuration (see Table 6.6) resulted in a much better resource cost. Only 79% in relation to utilizing the cluster in full at all times. However 14.5% of all batches were late due to delay constraint violation. This happens for the same reason as with default configuration.

Thesis

In terms of latency the default configuration (see Table 6.7) is performing more or less similar to Spark's custom configuration with 14 less delay violations. However it uses only 71% of available

	Duration [ms]	Delay [ms]	Executors
Minimum	679	0	12
Lower quartile	1412	0	12
Median	1904	0	14
Mean	1851	56.28	15.09
Upper quartile	2209	0.5	16
Maximum	3972	2354	20
Breaches	14	61	N/A

Table 6.8: Underfitting configuration (Thesis, January)

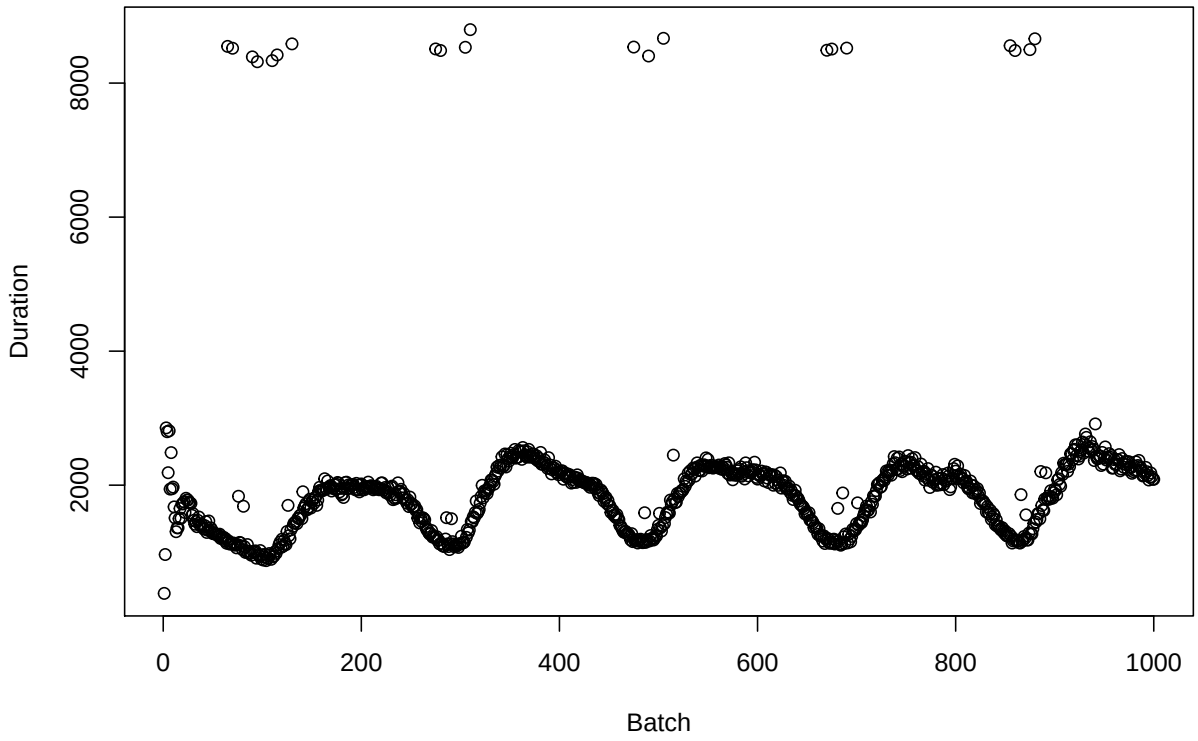


Figure 6.2: Duration in milliseconds for custom configuration

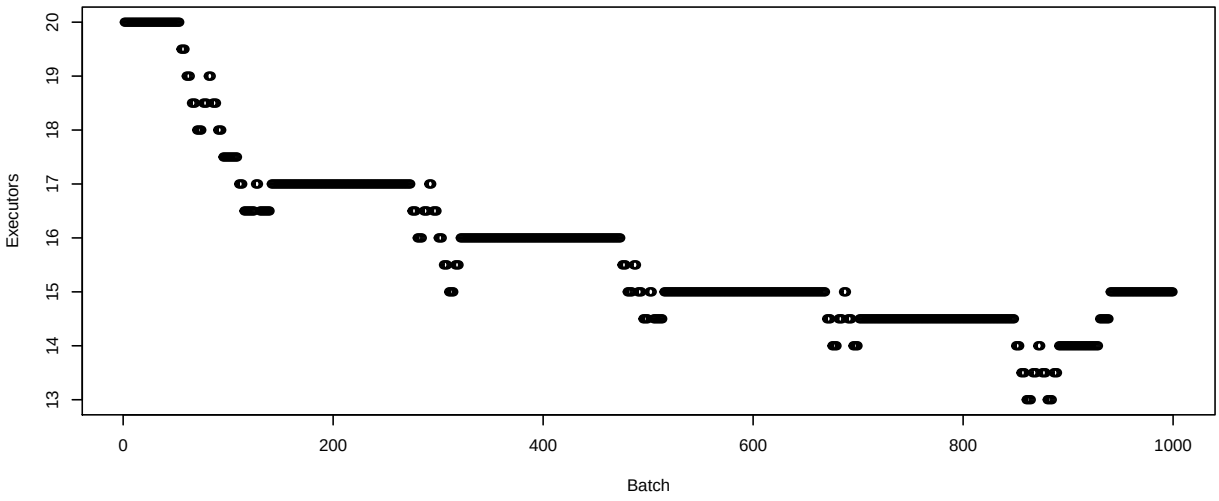


Figure 6.3: Executor changes for custom configuration

	Duration [ms]	Delay [ms]	Executors
Minimum	364.5	0	9
Lower quartile	1431.4	0	11.5
Median	1872.8	0.5	14.5
Mean	1942.2	346.3	14.81
Upper quartile	2302	1	17
Maximum	8970	6619	20
Breaches	59	189	N/A

Table 6.9: Overfitting configuration (Thesis, January)

	Duration [ms]	Delay [ms]	Executors
Minimum	962.5	0	20
Lower quartile	1349.6	0	20
Median	1843.5	0	20
Mean	1771.8	6.016	20
Upper quartile	2143.5	0.5	20
Maximum	3620	1203	20
Breaches	3	8	N/A

Table 6.10: Default configuration (Spark, February)

resources making it 10% more cost efficient.

The underfitting configuration (see Table 6.8 works well. It breaches delay in only 6.1% of all cases. Yet it also works better than any configuration of the original solution in terms of resources with a 75% outcome. Such good results come from more preservative behaviour. Each change kills or adds at least 4 executors. This together with lower threshold breach constraint and milder Adaptive Window delta causes the optimization module to reconfigure the cluster in a less intensive manner. For more visual representation of resource changes see Figure 6.4.

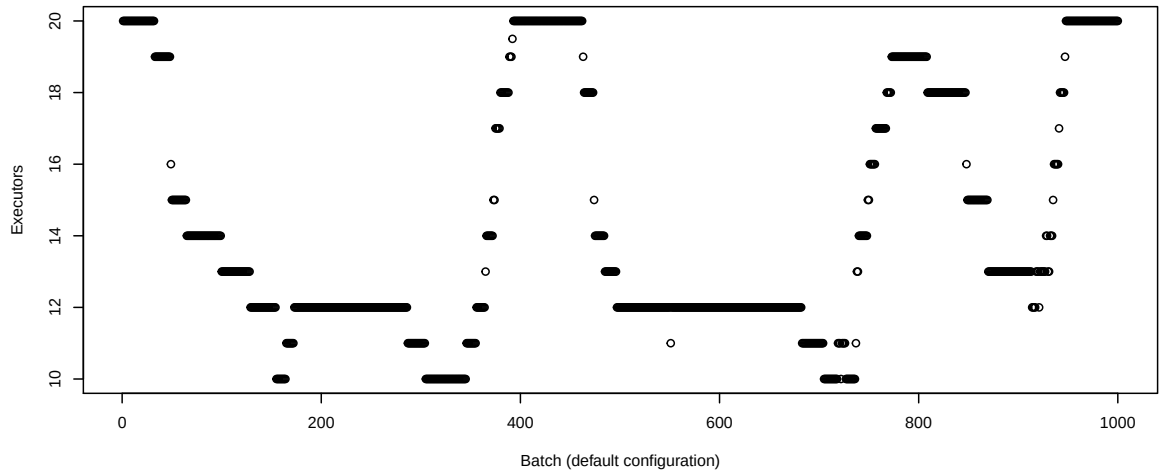
The overfitting configuration (see Table 6.9) produces the worst results. Not only it is worse than default one in terms of latency constraint violations - it also uses more resources. This is caused by much more frequent changes that lead to cluster becoming unstable. With frequent executor killing and adding there is a constant flush of stored information. This is clearly not a proper way to handle the stream. It is worth noticing that the resource usage in time differs a lot from the rest. However in comparison to custom configuration of Spark solution (see Table 6.6) it does slightly increase delay breaches but at the same time it slightly decreases resource usage costs.

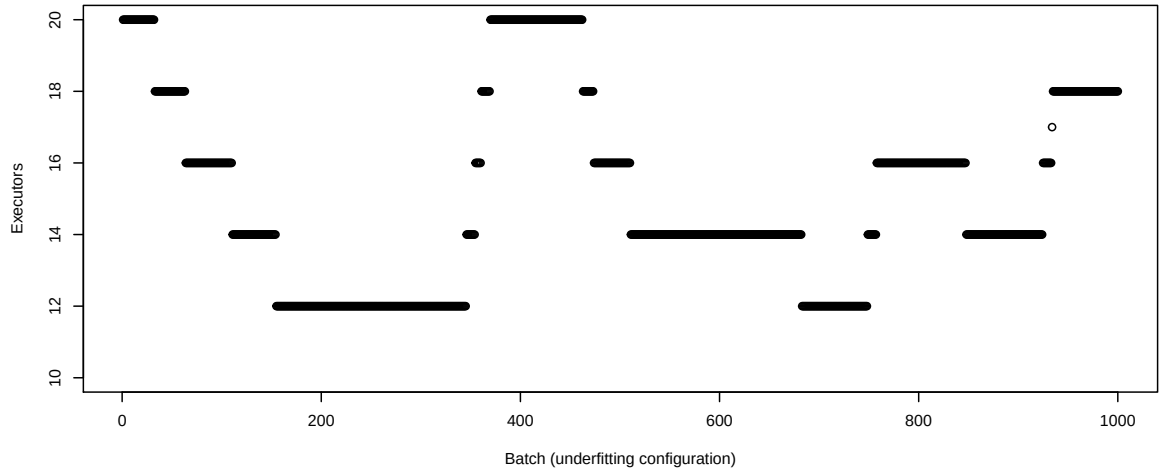
6.4.2 February dataset

Spark

For February dataset the default configuration (see Table 6.10) doesn't inflict any change upon resource usage. All of the executors are used all the time without interruptions. This results in excellent Quality of Service (QoS), yet doesn't bring any possible monetary savings.

The custom configuration however (see Table 6.11) breaches latency constraint almost as often as in case of January dataset. Nevertheless the cost efficiency is much lower. This can be explained with the new stream of information being more demanding than the previous one - a fact observable in Table 6.1.





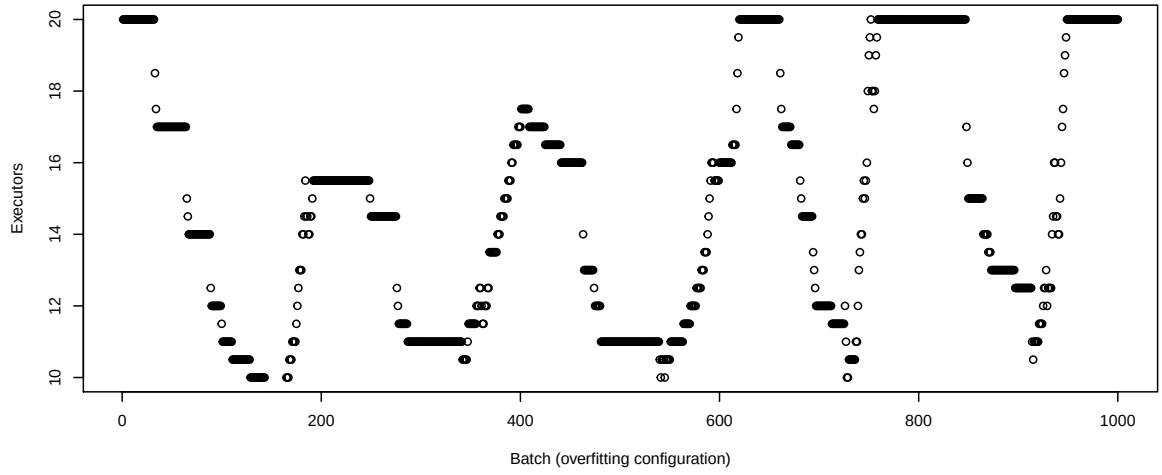


Figure 6.4: Executor changes for respective configurations of Thesis optimization model

	Duration [ms]	Delay [ms]	Executors
Minimum	337	0	15
Lower quartile	1486	0	17
Median	1922	0.5	18
Mean	1992	435.1	18.19
Upper quartile	2284	1.5	19.5
Maximum	8601	10006.5	20
Breaches	24	147	N/A

Table 6.11: Custom configuration (Spark, February)

	Duration [ms]	Delay [ms]	Executors
Minimum	468.5	0	11
Lower quartile	1508.8	0	14
Median	1987.2	0.5	16
Mean	2000	163.2	16.51
Upper quartile	2392.1	1.0	20
Maximum	4320.5	4857	20
Breaches	37	141	N/A

Table 6.12: Default configuration (Thesis, February)

Thesis

The results that can be seen on Tables 6.12, 6.13 and 6.14 are comparable to the ones from January dataset. All of them are more resource efficient with underfitting solution being the best out of all when it comes to $cost/QoS$ ratio. However the overfitting configuration brought many more latency violations - an effect that might be expected for the same reasons as in Section 6.4.1.

6.5 Summary

As we can see both solutions are comparable in terms of latency constraint violation. However in each case our custom optimization model outperformed significantly upstream one.

Spark's solution to resource management is much more preservative. It doesn't adhere fast to rapid changes in stream intensiveness. This has it's benefits. First of all there is much lower chance that there won't be enough executors to process spike in data income. Also fewer changes

	Duration [ms]	Delay [ms]	Executors
Minimum	458.5	0	12
Lower quartile	1411.2	0	16
Median	1920.5	0	16
Mean	1877.3	88.57	17.32
Upper quartile	2236.9	0.5	20
Maximum	4924.0	5069	20
Breaches	15	53	N/A

Table 6.13: Underfitting configuration (Thesis, February)

	Duration [ms]	Delay [ms]	Executors
Minimum	369	0	10.5
Lower quartile	1516	0	14
Median	1962	0.5	16.5
Mean	2037	1160.963	16.45
Upper quartile	2357	3.625	20
Maximum	9594	12145.5	20
Breaches	80	243	N/A

Table 6.14: Overfitting configuration (Thesis, February)

in executor number results in less JVM threads being spawned and that overall leads to greater system stability.

Nevertheless there is 5% to 10% difference when it comes to cost value. Moreover our solution utilizes available executors in much more adherent way. The simulator part is very exact when it comes to assessing the amount of resources required for processing batches from Adaptive Window scope. This turns out to be a major economic advantage that also doesn't cause a lot of disturbance on user end.

Chapter 7

Conclusion

The purpose of this study was to invent a new optimization tool for Spark Streaming data processing system. It involved investigating modern approaches applied in various research project. Furthermore it was mandatory to dive deep inside Spark execution model which lacks any official documentation in order to understand how the problem can be approached. The resulting solution was targeted at satisfying business needs of companies that deal with heavy information workflow.

In this work a novel online parameter optimization system was introduced. It merges ideas from FUGU's resource profiling with Spark's software architecture. In the end it resulted in creation of very precise simulation algorithm that is capable of predicting latency given an arbitrary number of executors. It was later used together with dynamic parameters and search algorithm using cost function to adapt cluster for significant shifts in stream intensiveness.

This resulted in a software library that is highly independent of upstream Spark code and may be used in cases when it is not possible to make any changes in the cluster manager. It adheres to the best professional standards when it comes to project quality. There are no external dependencies used and internally the software doesn't reuse any already existing code.

Results of the developed solution are very promising in comparison with experimental one that has been already implemented in existing codebase. It significantly surpasses original solution in terms of resource usage while keeping Quality of Service on similar level. When tuned well for desired stream of information, it gets better in both terms.

There is however a great area of improvement for this approach. First of all it is highly probable that making cluster granularity a dynamic parameter could improve the way the system adapts to any changes. It was showed that overfitting solution tends to get closer to minimal required number of executors while the opposite one prevented the cluster from becoming unstable. If a middle ground could be obtained in a more online manner it could lead to finding more optimal solutions. Another idea is to expand parameter space with more factors and implement more sophisticated search algorithm such as Recursive Random Search heuristic.

Lastly it is worth mentioning that Big Data market becomes stronger every year. All of us are producing more and more information that is later gathered and processed by different companies. In perspective this is a great field that encourages scientists all over the world to think about new solutions when it comes to preserving privacy, making the computation process more efficient or minimizing resource usage costs. This Thesis tried to make one step further in two of those subjects. Hopefully the results will encourage some future work that improves this solution or creates a much better one.

Bibliography

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 277–289, 2005.

[2] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, 2017. USENIX Association.

[3] Apache Foundation. Apache Spark. `http://spark.apache.org/`.

[4] Apache Foundation. Apache Storm. `http://storm.apache.org/`.

[5] J. M. Baumgartner. I have every publicly available reddit comment for research. 1.7 billion comments @ 250 gb compressed. any interest in this? `https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/`.

[6] A. Bifet and R. Gavaldà. Adaptive learning from evolving data streams. In *Proceedings of the 8th International Symposium on Intelligent Data Analysis: Advances in Intelligent Data Analysis VIII*, IDA '09, pages 249–260, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] T. C. Burak Yavuz. Running streaming jobs once a day for 10x cost savings. `https://databricks.com/blog/2017/05/22/running-streaming-jobs-day-10x-cost-savings.html`.

[8] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. R. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 725–736, 2013.

[9] L. Fischer, S. Gao, and A. Bernstein. Machines tuning machines: Configuring distributed stream processors with bayesian optimization. In *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*, pages 22–31, 2015.

[10] K. Greene. The secret behind twitter's growth. `https://www.technologyreview.com/s/412834/the-secret-behind-twitters-growth/`.

[11] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2351–2365, 2012.

[12] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer. Auto-scaling techniques for elastic data stream processing. In *The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, Mumbai, India, May 26-29, 2014*, pages 318–321, 2014.

[13] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer. Online parameter optimization for elastic data stream processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*, pages 276–287, 2015.

[14] Internet Live Stats. Google Search statistics. `http://www.internetlivestats.com/google-search-statistics/`.

[15] B. Lohrmann, P. Janacik, and O. Kao. Elastic stream processing with latency guarantees. In *35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015, Columbus, OH, USA, June 29 - July 2, 2015*, pages 399–410, 2015.

[16] B. Saeta. Why we love scala at coursera. `https://building.coursera.org/blog/2014/02/18/why-we-love-scala-at-coursera/`.

[17] M. Stonebraker, U. Çetintemel, and S. B. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47, 2005.

[18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, 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 Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28, 2012.

[19] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 423–438, 2013.

[20] S. B. Zdonik, M. Stonebraker, M. Cherniack, U. Çetintemel, M. Balazinska, and H. Balakrishnan. The aurora and medusa projects. *IEEE Data Eng. Bull.*, 26(1):3–10, 2003.