Annales Univ. Sci. Budapest., Sect. Comp. 55 (2023) 19–37

PERFORMANCE IMPACT OF NETWORK SECURITY FEATURES ON LOG PROCESSING WITH SPARK

Attila Péter Boros, Péter Lehotay-Kéry and Attila Kiss

(Budapest, Hungary)

Communicated by András Benczúr

(Received December 1, 2022; accepted January 31, 2023)

Abstract. Various industries maintain a large number of machines to run their production lines and services. These types of systems process and produce massive amounts of data to provide high quality and availability for their customer services. Therefore, these systems should constantly be inspected, to not only provide continuously the standard levels achieved but also be upgraded to keep up with the market competition. Our aim is to examine Apache Spark and to find one of the most suitable configurations that perform best on our challenges and can be further applied in real, live scenarios. In addition, despite that several studies in this field were already done, none of them considers the security factor of Spark during computation when predicting run time.

The presented work entails testing Apache Spark for log processing in standalone cluster setups with a varying number of workers on different submitted tasks. We also examine the performance impact of enabling authentication in the network communication between cluster nodes with these setups. Our results show that increasing the number of executor nodes and simplifying the underlying algorithm does not always influence performance in a positive manner as expected. Furthermore, securing network communication between Spark processes increases the overall execution time of submitted jobs noticeably.

Key words and phrases: Big data, data analysis, distributed system, log processing, network, spark, telecommunication.

2010 Mathematics Subject Classification: 68R99.

https://doi.org/10.71352/ac.55.019

1. Introduction

As part of a wider project on analyzing log files generated by telecommunication systems, we investigate examining the performance impact of securing network communication between cluster nodes.

Distributed computation for analyzing data is applied increasingly by companies in various industries to extract information from large amounts of generated log files. For this reason, to monitor, analyze and detect anomalies inward states and a large amount of generated logging information about the behavior of these systems, there was introduced the difficulty of processing massive amounts of collected input data. Hence a large variety of tools were recently developed to provide high-performing solutions to such problems. Together with the increasing demands on completion time, several privacy concerns are involved when dealing with sensitive data, therefore the proposed solutions have to be not only efficient but also secure, according to the recently introduced privacy regulations.

Our aim in this work is to examine Apache Spark, one of the most prominent frameworks for our purposes. During the examination, we are going to focus on finding an optimal cluster configuration to run our tasks and elaborate on how different parameters affect the overall performance.

Similar studies and investigations in this field were already made by introducing high-level system enhancements for achieving better performance or proposing theoretically well-founded frameworks and statistically proved formulas [3, 6, 9]. These works not only estimate the time to run such computations but suggest a best-performing configuration with given allocatable resources and a deadline for the computation to complete. In addition, several works propose new frameworks aimed at securing such systems [2, 4, 10]. Unfortunately, none of them considers these features combined, such as estimating optimal cluster composition before a deadline, taking into account securing network communication between Spark compute nodes and execution processes. More generalized we have not yet found a such formula that takes into consideration communication overhead when estimating the run time of a job completed within a predetermined time limit.

Given the high number of processes and the complexity involved in the communication between nodes, this impact is a non-trivially determined parameter when considering execution time in calculation. Therefore, we experimented on various scenarios with a variable number of cluster nodes by examining the impact measure of increasing the amount of compute nodes on execution time and taking into consideration securing network communication during Spark job run time. In the long run, these experiments will serve as a first step in determining a formula for computation performance alongside a fixed cluster configuration, where additional overhead on network communication is taken into account.

Our work is structured as follows. In the next section, we review related works by summarizing the main ideas and newly introduced concepts in this field. After that, we present the framework of Apache Spark, the system configuration used for evaluations, and then give a short presentation about the test cases. Consequently, we show our experimental results and give a conclusion on evaluation outcomes. Finally, we shortly discuss the planned future works.

2. Related works

In this section, we look through similar works on estimating the processing time of Apache Spark jobs and on considering the effect of security features by proposing new frameworks for this purpose.

Authors of [6] introduce a new cloud-based middle-ware platform that supports on-demand composition and configuration of security mechanisms to ease regulatory compliance enablement. In the course of this work, the benefits of the approach are highlighted by evaluating the performance impact and examining the trade-offs of different security mechanisms.

Authors of [9] propose a new framework for examining and modifying Spark cluster resources taking into account the already available resources and computation deadline. This is done by analyzing the execution model of Spark and proposing a formula for estimating the execution time of jobs as an optimization problem.

In [3], the determination of the proper parameter values that meet the performance requirements of workloads had been studied. The paper also aims to minimize both re-source cost and resource utilization time. A sampling and simulation framework, a simulation-based cost model has been introduced to predict the performance of jobs accurately and to recommend the total amount of the vcore and memory resources toward the cost-effectiveness of Spark jobs. The research also includes empirical experiments showing the efficiency and effectiveness of the proposed algorithms.

In [10], a new authentication mechanism is proposed, implemented, and evaluated, using certificates for data analytic tools, providing advantages over Kerberos and thus giving better data protection by providing improved authentication, meeting industry-level multi-factor authentication and scalability. To evaluate the possibility of replacing Kerberos, the mechanism is implemented in Spark.

Authors of [4] propose a general securing framework for end-to-end cloud environments to protect data-in-use while communicating between server nodes.

For this purpose, they use the Intel SGX SoC encryption, available in processors starting with the Skylake series, and the analytic engine of Apache Spark SQL. Despite the fact that this proposal addresses a suitable solution for the first security problem we presented in the introduction, it does not focus on the performance impact of securing these cloud systems. Nor do they examine a general heuristic on the effect of applying their proposed framework.

Authors of [2] propose SafeSpark, a new framework for securing data analysis by encrypting processed files and using an environment secured by using hardware-based protection. The proposed framework combines the cryptography processing engine of Apache Spark SQL with Intel's hardware-based SGX encryption. The performance impact of this newly proposed framework is measured by evaluating three official TPC-DS benchmark scenarios. Results show that run time performance overheads range from 10% up to 300% compared to an unsecured version of vanilla Apache Spark.

In [8], authors introduce three new security approaches into the Apache Spark framework with the aim to secure the exposure of data during processing, caching, and spilling data to disk. Securing the temporary written data is realized by using a combination of Shamir's Perfect Sharing and the Information Dispersal Algorithm. As a result, their evaluations show that encoding temporary chunk files during computation using the newly introduced framework, takes a penalty of between 10-25% on performance regarding the run time.

In [7] authors present a secure stream processing system based on Intel SGX aimed specifically for processing live medical biometric data. The implemented system prototype is evaluated on several realistic datasets, showing that the proposed system achieves a modest overhead compared to the vanilla Spark run time while assuring protection guarantees under powerful attackers and various threat models.

3. Technologies used

In this section, we present the technology of Apache Spark, show the cluster composition on which Spark is deployed, and finally, give a brief summary of the test cases and the motivation behind them. For our purposes, we use Apache Spark version 3.0.0, a unified analytic engine for processing large-scale data. This technology provides a wide variety of interoperability and compatibility with other systems. In addition, it supports several client interfaces by the help of which users can drive the system. These are all achieved by realizing a relatively simple and clear interface, through its built-in operations on a resilient distributed dataset (`RDD`) and `DataFrame` objects. `RDD` objects provide a functional interface for transformations that can be applied to the data they represent, while `DataFrame` objects provide support for optimizing and running queries on data using SQL syntax.

Furthermore, it supports client interfaces for various popular programming languages (i.e. Python, Java, Scala, R), from which we use *pyspark* as the Python client for driving the system. Spark also provides compatibility with underlying external systems by supporting other filesystems (e.g. HDFS) and being easily deployable with other virtualization technologies (e.g. Docker, Kubernetes) in cloud environments. For simplicity, we use the system in the standalone mode which means Spark is installed directly on each machine of the cluster and reads data directly from disk without involving other third-party software to reach the data we want to analyze. This feature involves that all processed data needs to be replicated on each of the involved machines. Furthermore, to keep up with today's fast evolution of technologies and traffic magnitude, it supports processing streamed data, graph processing, and machine learning by providing built-in libraries for these purposes. [5]

Besides Apache Spark, we considered finding other suitable solutions and frameworks. Technologies examined were Apache Flink and Elasticsearch[1]. Despite the fact that they were providing satisfactory solutions for our goals, the interface provided in Python by the former candidate was too rigid and would require the rewriting of our whole codebase to Java or Scala. Meanwhile, the latter alternative would involve higher unnecessary system complexity by involving a pipe of several auxiliary APIs (Filebeat with Logstash, or Fluent Bit with Fluentd Forwarder) for delivering data to the main search engine components.

4. Test cases

On evaluating our test scenarios we use a cluster consisting of ten machines, each equipped with an Intel Core, Haswell series twelve-core processor, running at 2.5Ghz, 16Gb of RAM memory, and 250Gb of disk storage. These nodes are strongly interconnected, such as each machine is reachable from any other through the network with a bandwidth of 10Gbps.

Our work is part of a wider project aiming at monitoring and analyzing node configuration, state descriptor, and log files, generated during the execution of automatic incremental software deployments on production customer telecommunication network server nodes. These processes are executed on data collected continuously on a daily basis. This collection of data is done with the aim of getting more insights into the impact of these software upgrades by comparing states from before and after the execution of an upgrade. In addition, these are performed to be able to treat system malfunctions and anomalies faster, or in case to call back the upgrade.

Hence, we are focusing on processing these generated log files by searching for restart events, and the reasons behind them, calculating logging intensity and boot time and comparing results from before and after the upgrades.

Since most of these customer networks contain thousands of nodes and produce large amounts of several kinds of log files, it is required to use a suitable platform, in our case Apache Spark, that supports distributed algorithms and technologies in an effective way.

Even though realizing the processing of these files involves technical complexity in itself, this paper focuses on evaluating the impact of securing network communication between cluster nodes and processes during computation. For this reason, we have chosen two test cases to run with different setups and evaluate the obtained results. These two test cases are collecting the intensity of local logs (counting the number of log lines generated in a given time frame), and collecting restart counts on the different nodes, obtained from examining two different types of error log files, generated daily on the telecommunication server nodes.

These two separate kinds of files are used in the production environment too for restart counts and for log intensities. The restart counts are extracted from files containing filtered entries that are known to contain the restart events, but contain fewer entries for a given time frame but also cover larger time frames. The relevant lines from these logs are as in the example below:

```
12: 2019-01-11T17:07:23+0000#NOOP#eh#APU warm restart 0,
  slot 9, error 193
13: 2019-01-11T17:07:26+0000#WARM#eh#Init NPU/node warm restart
14: 2019-01-11T17:08:17+0000#NOOP#eh#Node Warm Restart (Management)
```

Log intensities are extracted from log files containing unfiltered log messages. One sample line from these files is:

```
Jan 14 14:00:32 ML66-10-41-103-4 cli: .200
  ../platform/cmo/cm/src/cli-util.c(556)SYSLOG:NOOP: Backup ordered NOW!
```

The algorithms presented by these test cases can be sketched by the following pseudocode:

```
init( result )
for path in input_files:
     file = spark.read.text( path ).rdd.map()
     result += file.filter( relevant lines )
                    .map( extract relevant parts of log lines )
                    .map( convert date format )
rows = spark.union( result )
```

```
rows.collect()
table = spark.createDataFrame( rows )
return spark.sql( table )
```

Differences between these two cases lie in the parameterized operations presented in the pseudocode structure of the test cases. First, files are read by `spark.read.text` into `RDD`. Filter and map operations on log lines differ in the way relevant data extraction and format conversion are done. In the case of the restart analysis, `filter` searches for the elements of the log line triples that can be seen above. In the case of intensity count, since there are entries containing multiple lines and we only want to count each entry once, it filters for the first lines of the entries, containing the date. In the case of the restart analysis, `extract relevant parts of log lines` extracts the necessary information for further analyses like the kind of restart triggered or the node module that requested the restart. In the case of the log intensity count, it collects the data that we want to use in the later grouping. After this, `convert date format` just saves the date in a common format. In the end, the `union` of the results are created and `collect`ed. The main difference between the two algorithms is held by the conversion of `RDD`s to `DataFrame` objects and querying the resulting table.

In each of these cases, conversion of `RDD` objects to `DataFrame` consists of simple correspondence of objects to table rows, but in each test case applying different schema. Among these test cases, our codebase contains also analyzing scripts for measuring restart intensity on nodes. In this case, this conversion is a switching of the table rows to columns in order to calculate the number of restart counts on each node for every restart type. Here the burden of complexity is taken on creating the `DataFrame` instead of querying the data.

In the case of determining log intensities, the emphasis is taken on executing the final SQL query. Here a `GROUP BY` operation is used on the result line for counting the intensity of logs on the given day and node:

```
SELECT TimeStamp, NodeIp, COUNT(*) as Count FROM rows
        GROUP BY TimeStamp, NodeIp ORDER BY TimeStamp ASC, NodeIp ASC;
```

In the case of collecting restarts the SQL part of it is equivalent to a standard `SELECT * FROM table;` - the essence here being the preprocessing of files with `RDD` transformations.

Besides the original version of the test cases, we evaluate also the simplified versions of them. These differ from the structure of their original versions in the body of the loop, such as simplified versions' `filter` and two `map` operations are joined into a single `flatMap` operation. Other parts of the test cases are left intact. Equivalence of the original and simplified versions was tested by using black box tests on results in several critical and general cases. This optimization causes in some cases significant performance increase and in general a noticeable improvement compared to the original versions. These results are further presented in the next section.

5. Results

In this section, we present the results of run-time performance evaluations executed on the two test cases presented in the previous section.

According to the official documentation [11] securing RPC network communication between processes in Apache Spark involves two steps in setting the configuration. The first step is enabling authentication of its internal connections and consequently specifying a secret key by setting `spark.authenticate` and `spark.authenticate.secret`. The second step is applying an AES-based encryption protocol for RPC connections by setting

```
spark.network.crypto.enabled
spark.network.crypto.keyLength
spark.network.crypto.keyFactoryAlgorithm
```

properties on top of the previous authentication settings.

However, our initial aim was only to (a) test the job execution performance of our test cases regarding security features, during test plan creation the idea came up of (b) experimenting with alternating numbers of executors to suggest an optimal cluster composition on these specific cases by considering the actual hardware configurations. In addition, another two ideas came up and became part of the test plan. One was (c) examining how the simplified versions of these test cases perform with a different number of executors where only one `RDD` operation is executed on each file, and the other one was (d) how the reduced number of `RDD` operations influences the performance of secured communication between processes.

Unfortunately, we managed to execute our encrypted test cases only without enabling AES-based encryption algorithms, since some still unknown technical reasons[12] this system feature causes an inability for Spark workers to establish a connection with the master node. Therefore the presented *secured* benchmark results involve only *authentication* between execution processes. Furthermore, test cases were planned to run on up to ten thousand input files, but despite physical limits regarding system memory, with an exception in some cases tests by only up to one thousand input files could have been completed in some cases.

In spite of the aforementioned technical difficulties, we ran our test cases using a varying number of input file counts - in each test case with 10, 30, 60, 100, 300, 500, 1000, 3000, 5000, and 10000 input files with sizes from 204Kb up to 438Mb - and different input file sorts (separate kind of files for restart counts and for log intensities), where the files used for the restart calculations consisted of 29 up to 1943 log lines in each file and the files used for the intensity calculations consisted of 643 up to 2010 log lines in each file. These input files were generated, and their structure is based on a sample of original log files. Furthermore, we distinct measurements on enabling authentication of communication between processes. Performance results are calculated from the average of *three* different measurements to avoid unexpected accidental results.

In the following tables, our measurement results will be presented as follows. Tables 1, 2, 3 and 4 contain run time results in *seconds*, obtained from running the restart counts test case. Tables 5, 6, 7 and 8 contain run time results also in *seconds*, obtained from running the log intensity test case with varying even count of cluster nodes and file counts, abbreviated as *FC* in table headers.

Furthermore, simplified versions of these test cases were created to evaluate run times using less `RDD` operations, hence less communication between processes being involved. Matching of results considering the simplified versions was proven with black box unit tests, taking care also on corner cases i.e. no input files, or input files without relevant log lines.

5.1. Restart counts

Table 1 contains benchmark results from running the original version of the restart counts test case *without* using RPC authentication between execution processes. Here we can notice that distributing job execution on multiple nodes returns its effect only when there are more than five hundred files (sum of file sizes: 15Mb) to process.

FC	1	2	4	6	8	10
10	11.63	15.59	14.67	14.56	14.71	14.78
30	13.42	16.91	16.78	16.93	16.76	16.87
60	15.76	18.93	19.03	18.77	19.00	18.87
100	19.06	21.93	21.82	21.77	21.78	21.98
300	35.17	35.12	34.99	35.09	35.48	35.18
500	54.45	48.40	49.13	49.23	49.28	48.90
1000	130.14	92.88	94.04	93.15	92.91	93.40
3000	551.22	403.82	406.88	404.58	404.17	401.54
5000	1306.10	1152.87	1194.75	1214.81	1222.72	1207.24
10000	4939.18	4228.67	4326.44	4109.65	4076.14	4069.45

Table 1. Restart count processing times in seconds without authentication using different number of files and cluster nodes

We can further notice that processing times in the last row are much higher than expected. This effect is caused by the lack of a sufficient amount of memory, therefore the system has to write temporary computation chunks on the disk instead of utilizing only in-memory calculations. Furthermore, run times until three hundred files (below 15Mb of input size) performs best when applying only a single node for computation. This phenomenon can be caused by the reason that the system does not have to distribute the task and schedule workers to execute new incoming processes, thus saving the overhead derived from distributing execution.

FC	1	2	4	6	8	10
10	11.33	15.01	14.97	14.99	14.71	15.00
30	13.46	16.93	16.83	16.92	17.12	16.85
60	15.94	19.11	19.12	19.29	19.07	19.27
100	19.28	21.89	22.20	22.03	21.93	22.26
300	35.15	35.82	35.64	35.22	35.20	35.81
500	53.64	49.93	49.12	49.17	49.17	48.72
1000	111.34	94.05	93.61	93.58	93.60	93.52
3000	542.62	405.36	400.85	403.71	403.29	400.63

Table 2. Restart count processing times in seconds with authentication enabled using different number of files and cluster nodes

Table 2 presents benchmark results obtained from run times of the original restart counts script with enabled authentication between Spark execution processes. The same observation on these results holds here as before, and we can now complete them by correlating with run times obtained from disabled authentication measurements. Minimal deviations in benchmark results are noticeable, between -14.45% and 3.6% in standalone mode and -3.77% and 3.17% in cluster node, meaning that enabling authentication between processes had the impact of increasing run time in standalone setup by slightly more than three and a half percent and cluster mode setups were completing faster with a maximum decrease of fourteen and a half percent, and an average of around half percent. Looking deeper into the insights we can further notice that the most dominant value differences appear when running the test case on three thousand files. In these cases, run time deviations vary only between -1.48% and 0.38% which is not a significant performance impact, considering the overall time requirement of the test case in this setup.

Table 3 presents performance results obtained from running the simplified version of the restart counts test case. Simplified versions, as discussed before involve an equivalent implementation of the same algorithm but using only a single `flatMap RDD` operation for each input file, obtained by joining operations.

FC	1	2	4	6	8	10
10	11.18	16.39	14.87	14.72	14.62	14.43
30	13.39	16.82	16.79	16.65	16.64	16.78
60	15.98	19.05	18.81	19.14	18.80	18.94
100	19.18	21.75	21.81	22.22	21.72	21.50
300	34.33	35.13	35.40	34.57	34.83	35.06
500	52.29	48.30	48.37	48.01	48.07	48.01
1000	108.61	90.49	90.77	90.57	91.13	91.62
3000	548.88	393.12	393.68	393.77	391.38	394.80
5000	1312.48	1166.55	1169.19	1160.15	1173.18	1203.14
10000	4653.05	4120.01	4148.37	4129.37	4103.91	4119.52

Table 3. Restart count processing times in seconds without authentication using a single operation, different number of files and cluster nodes

Observations from the original version of this test case also hold here. Compared to the results of the original version, run time result deviations are between -16.55% and 5.08%, where the run time decrease is achieved by using the standalone setup and the highest increase is when executing the test case on ten files with two executors. In addition, the highest run time decrease considering only the distributed measurements can be noticed when executing the test case on five thousand input files using six nodes for computation, achieving a decrease of 4.5%, which percentage can be a significant improvement. An average of 1.35% of run time decrease was measured during test runs of this case.

FC	1	2	4	6	8	10
10	11.95	15.11	14.87	14.84	14.96	14.83
30	13.88	16.82	16.84	16.96	17.27	16.96
60	16.42	19.16	19.16	19.10	19.08	19.17
100	20.23	21.67	21.83	22.00	22.11	21.95
300	36.00	34.85	35.24	34.64	35.21	35.00
500	54.28	48.28	48.42	48.31	48.71	48.64
1000	114.21	91.66	91.36	91.71	92.20	91.38
3000	586.25	404.62	406.87	401.77	394.26	393.18

Table 4. Restart count processing times in seconds with authentication enabled using a single operation, different number of files and cluster nodes

Table 4 contains run time results obtained from running the simplified version of the restart counts test case with enabled authentication between Spark execution processes. Observations from the non-authenticated benchmark results hold here too. Interestingly, authentication impacts more consistently compared to the unsecured version, having a maximum run time penalty of

7.13% in standalone and 3.78% increase in cluster mode against run times of the non-authenticated and simplified test runs. Furthermore, there was measured a run time decrease of at most 1% and an outstanding value of 7.82% when comparing the test case executing with two executors on ten files.

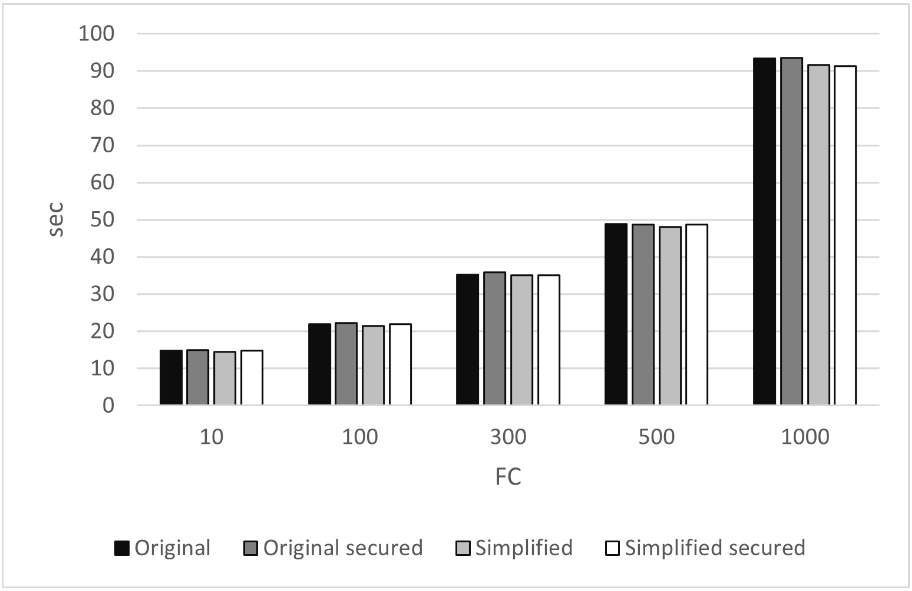

Figure 1. Restart data calculation on 10 nodes

Compared to the secured version of the original test case, run time performance deviations of the simplified versions were measured between -3.31% and 8.04%, where the highest improvement was detected when running the test case with a single node on three thousand files. The highest improvement between distributed versions was only 1.74% when running the test case on ten files with eight executors. In conclusion for this test case, simplified versions did not improve run times significantly when enabling authentication between processes, only by an average of 0.82% run time decrease.

At the end of examining the results of this test case, Figure 1 shows the summary of the four kinds of algorithms: the original, the original secured (with authentication), the simplified, and the simplified secured (with authentication). The processing time of the calculation can be seen in seconds for 10, 100, 300, 500, and 1000 files on 10 nodes. As it is detailed above, enabling the authentication and the algorithm simplification have little effect on the processing time.

5.2. Log intensities

To ensure that our results are not limited to a specific test case we involved a second one with a similar execution structure and with a more complex logic involved, also using different input sources with the purpose of being able to compare the outcomes of these two cases. In addition, we implement and compare simplified versions of these test cases by the different setups.

FC	1	2	4	6	8	10
10	12.29	18.51	22.65	22.60	22.23	19.82
30	14.13	20.21	20.62	20.70	20.49	20.48
60	16.22	22.20	22.32	22.42	22.45	22.20
100	19.10	24.49	24.94	24.77	24.72	25.13
300	34.71	38.93	39.19	39.36	38.85	39.25
500	53.88	57.36	57.43	57.39	56.84	57.37
1000	120.00	118.01	118.36	118.50	118.91	118.73
3000	1739.80		546.14	814.14	945.00	1010.76

Table 5. Log intensity processing times in seconds without authentication using different number of files and cluster nodes

Table 5 contains run-time results from executing the original log intensity test case *without* enabling authentication between Spark processes. One can observe, that the efficiency of using more than one machine takes its effect by analyzing more than one thousand input files with a total size of 146Mb data. The non-linear proportion in the increase of run times above one thousand files can be caused by the behavior of the system, spilling to disk temporary files during computation that can not fit into the memory. We also note that blank cells in these tables indicate execution failure during running the test case, caused by exceeding the available amount of memory on cluster nodes.

FC	1	2	4	6	8	10
10	13.58	18.42	19.91	20.07	22.24	22.46
30	15.43	20.81	20.73	20.87	20.94	20.74
60	17.76	22.53	22.56	22.49	22.64	22.52
100	20.69	24.85	24.78	25.05	25.11	25.02
300	37.25	38.79	39.34	39.30	39.82	39.27
500	58.44	57.14	57.80	58.14	57.26	57.38
1000	126.15	118.64	119.37	119.19	120.99	119.55
3000		980.55				1282.12

Table 6. Log intensity processing times in seconds with authentication enabled using different number of files and cluster nodes

Table 6 contains run time results of the log intensity test case with authentication enabled between execution processes. Observations regarding the increase rate between test cases from the non-authenticated version hold here also, but with more noticeable differences. In addition has to be mentioned that in most cases above one thousand input files results could not be obtained due to the memory limit exceeding.

Compared to the unsecured version, the largest run time increase of 26.85% was measured when running the test case on 3000 files using ten nodes for execution. Furthermore, without considering this outlier value, the maximum increase in standalone mode was 10.45% when computing ten files, and in cluster mode was 13.98%, when computing 10 files using 10 nodes. When considering cases only with one thousand files, where run times were measured around almost two minutes, run time deviations were showing an increase of 5.12% in standalone mode, meanwhile between 0.54% and 1.75% in cluster mode, meaning that in general applying authentication between processes adds a minimal but noticeable overhead of more than a half percent, compared to the unsecured setup.

FC	1	2	4	6	8	10
10	13.10	18.96	21.98	19.65	21.96	21.41
30	14.98	19.59	19.74	20.24	19.97	19.96
60	17.15	21.44	21.42	21.63	21.49	21.73
100	197.85	23.74	23.42	23.23	23.52	23.79
300	35.50	36.72	36.06	36.45	36.21	36.82
500	55.01	52.87	52.21	52.46	52.27	51.91
1000	121.90	105.21	105.96	106.08	105.92	106.08
3000	1800.12	435.58				

Table 7. Log intensity processing times in seconds without authentication using a single operation, different number of files and cluster nodes

Table 7 presents test results of the simplified log intensity test case without securing network communication. Same observation on run time increase from the original test case results also apply here. In addition, compared to the original version, run time result differences vary from -13.05% up to 8% in both the standalone and cluster mode setups. The maximum decrease was achieved by running the test case on ten executors computing its result from ten files, while the maximum increase in performance was measured by using four executors and processing ten files. The average benefit of simplifying these processes was a decrease of 3.98% in run time.

Table 8 contains run time results from running simplified versions of the log intensity test case with enabled authentication between Spark processes. Observation from the original version results cannot be applied here, since

FC	1	2	4	6	8	10
10	13.75	19.58	22.67	22.87	21.89	22.26
30	15.81	20.16	20.22	20.25	20.43	20.24
60	18.02	21.79	21.81	21.52	22.11	21.53
100	20.87	23.81	23.74	23.69	23.59	23.88
300	37.26	36.73	36.38	36.49	36.70	36.67
500	58.62	52.65	52.84	53.09	52.77	52.61
1000	135.16	106.08	105.72	104.94	106.17	105.51

Table 8. Log intensity processing times in seconds with authentication enabled using a single operation, different number of files and cluster nodes

measurements above one thousand files were failing due to memory constraints.

Compared to the unsecured version results, the largest run time increase of 16.38% was measured when running the test case on 10 files with six executors. In addition, without considering this outlier value the maximum increase in standalone mode was 10.89% when computing one thousand files. In cluster mode, the largest increase was 3.98%, when computing 10 files using 10 nodes, showing an average increase of 1.44% in run times compared to the unsecured results.

In addition, compared to the results on the original variant with enabled authentication, run time results deviate from -12.24% to 13.91%, the highest increase among results being measured using six executors when analyzing 10 files. Meanwhile, a decrease of more than 12% shows up when running the test case with eight executors and one thousand files, showing an average decrease of 3.43% of run time in the authenticated cluster setup.

At the end of examining this test case, Figure 2 shows similar results as Figure 1 in the means of correlation of the different dimensions. However, simplifying the algorithm decreased the processing time by a greater amount.

5.3. Summary

Summarizing the outcomes of our test cases, we measured the execution times of two different test cases with similar algorithm structures. A simplified version was created from each of them. Having the equivalent original and simplified versions of each test case we examined their run times on the different number of worker nodes and amount of input files, each execution being tested with both the unsecured and secured setup. To validate and avoid accidental outcomes of our results we ran each case three times.

Therefore, results have shown that enabling authentication between processes with a significant amount of files increased run times by up to 3.35% running our test cases. In addition, between original and simplified versions we measured a run time decrease of up to 3.98% on average, where processing

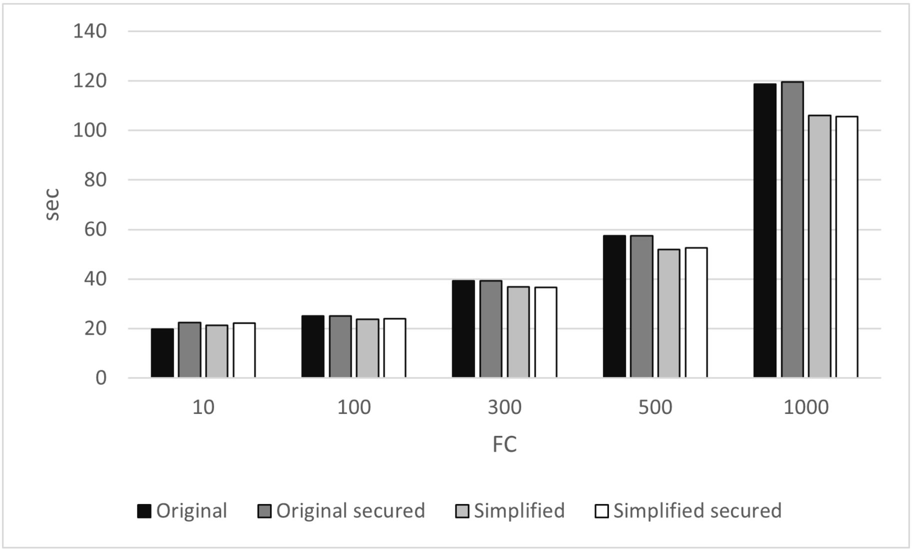

Figure 2. Log intensity calculated on 10 nodes

each input file consisted of only one `flatMap RDD` transformation.

Summarizing the number of used cluster nodes, based on our tests, it can be concluded, that using a smaller test set, up to 100 files, using a single cluster node resulted in lower processing time. From 1000 files, the processing times were always lower using multiple cluster nodes.

6. Conclusion and future work

Constantly accelerating development of technologies and the evolution of industries require nowadays innovative, stable, efficient, and most importantly secure solutions for their products and services to be able to keep up with the market competition. To constantly provide and improve serving these continuously evolving market facilities, organizations have to monitor their systems with the purpose of developing efficient ways for maintaining and upgrading them. This involves examining massive amounts of machine-generated data to provide efficient ways on supporting distributed algorithms. The security aspect of these solutions also becomes more and more important.

In our work, we examined the run time results of two different test scenarios, then evaluated the performance impact of securing network communication between execution processes, using the unified analytic engine of Apache Spark. As result, we found that enabling authentication between execution processes can impact run time by decreasing it up to 3.5%. Furthermore, as a side increment of these examinations, we found that simplifying the algorithm by rationally decreasing the number of Spark operations to only one on each input file can positively influence execution time by decreasing it by up to 8.04% using different system configurations. In addition, based on the presented test cases our recommendation is to use a single machine for analyzing low amounts of input files. For higher counts of input files fewer executors performed better in general, hence based on our results we suggest using cluster mode with two nodes for inputs between 500 and 3000 files and four nodes above 3000 input files.

In the following, we are going to look into overcoming the arisen technical issues to be able to perform similar benchmarks with encrypted communication between execution processes by using advanced encryption algorithms for this purpose. In addition, we would like to extend our measurements on larger amounts of files. Further investigations have to be conducted about the memory constraint failures when experimenting with more than 1000 files. It would be also interesting to run them on horizontally scaled machines in meanings of memory. Furthermore, we intend to design a framework, with the help of which one can determine the execution time of tasks by considering overhead on network communication between nodes and processes of the distributed Apache Spark system.

Acknowledgement. The project was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).
Supported by the ÚNKP-22-4 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.
This publication is the partial result of the Research & Development Operational Programme for the project "Modernisation and Improvement of Technical Infrastructure for Research and Development of J. Selye University in the Fields of Nanotechnology and Intelligent Space", ITMS 26210120042, co-funded by the European Regional Development Fund.
The project was also supported by the Ericsson-ELTE Software Technology Lab.

References

[1] **Boros, A.P., P. Lehotay-Kéry and A. Kiss,** A Comparative Evaluation of Big Data Frameworks for Log Processing, in: Kovásznai G., I., Fazekas, T., Tómács (Ed.) *Proceedings of the 11th International Conference on Applied Informatics (ICAI 2020)* (Eger, 2020), CEUR Workshop Proceedings **2650**, 2020, pp. 57–64.

[2] **Carvalho, H., D. Cruz, R. Pontes, J. Paulo and R. Oliveira** On the Trade-Offs of Combining Multiple Secure Processing Primitives for Data Analytics, in: D. Eyers, Voulgaris, S. (Ed.) *Distributed Applications and Interoperable Systems* (Valletta, 2020), Lecture Notes in Computer Science **13272**, Heidelberg, Berlin, 2020, 3–20.

[3] **Chen, Y., J. Lu, C. Chen, M. Hoque and S. Tarkoma,** Cost-effective Resource Provisioning for Spark Workloads, Proceedings of the 28th ACM International Conference on Information and Knowledge Management (Beijing, 2019), Association for Computing Machinery, New York, USA, 2019, 2477—2480.

[4] **Feder, O., Gershinsky G. and Tsfadia E.,** 2018. RestAssured: Securing Cloud Analytics, Proceedings of the 11th ACM International Systems and Storage Conference (Haifa, 2018), Association for Computing Machinery, New York, USA, 2018, 120.

[5] **Frampton, M.,** *Mastering apache spark*, Packt Publishing Ltd, Birmingham, UK, 2015.

[6] **Le, M., K.R. Jayaram, Y. Weinsberg, D.J. Dean and S. Tao,** Agile Composition of Compliant Data Analytics Platforms, IEEE International Conference on Cloud Engineering (IC2E) (Vancouver, 2017), IEEE Press, Washington D.C., 2017, 51–58.

[7] **Segarra, C., R. Delgado-Gonzalo, M. Lemay, P.L. Aublin, P. Pietzuch and V. Schiavoni,** Using Trusted Execution Environments for Secure Stream Processing of Medical Data, arXiv preprint (2019) https://arxiv.org/pdf/1906.07072.pdf

[8] **Shah S.Y., B. Paulovicks and P. Zerfos,** Data-at-rest security for spark, IEEE International Conference on Big Data (Washington D. C., 2016), IEEE Press, Washington D.C., USA, 2016, pp. 1464-1473.

[9] **Sidhanta, S., W. Golab and S. Mukhopadhyay,** OptEx: A Deadline-Aware Cost Optimization Model for Spark, CCGRID '16: Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (Cartagena, 2016), IEEE Press, Washington D.C., USA, 2016, 193–202.

[10] **Velthuis, P.J., M. Schäfer and M. Steinebach,** New authentication concept using certificates for big data analytic tools, Proceedings of the 13th International Conference on Availability, Reliability and Security (Hamburg, 2018), Association for Computing Machinery, New York, USA, 2018, 1–7.

[11] https://spark.apache.org/docs/latest/security.html (accessed: 2020. 08. 30.)

[12] https://stackoverflow.com/questions/64050573/ (accessed: 2020. 11. 02.)

A. P. Boros, P. Lehotay-Kéry and A. Kiss
Eötvös Loránd University (ELTE)
Department of Information Systems
Budapest
Hungary
attila9778@inf.elte.hu
lepuaai@inf.elte.hu
kiss@inf.elte.hu