Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks

Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, María S. Pérez

Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks

Ovidiu-Cristian Marcu
Inria Rennes - Bretagne Atlantique
ovidiu-cristian.marcu@inria.fr

Alexandru Costan
IRISA / INSA Rennes
alexandru.costan@irisa.fr

Gabriel Antoniu
Inria Rennes - Bretagne Atlantique
gabriel.antoniu@inria.fr

María S. Pérez-Hernández
Universidad Politecnica de Madrid
mperez@fi.upm.es

***Abstract*—Big Data analytics has recently gained increasing popularity as a tool to process large amounts of data on-demand. Spark and Flink are two Apache-hosted data analytics frameworks that facilitate the development of multi-step data pipelines using directly acyclic graph patterns. Making the most out of these frameworks is challenging because efficient executions strongly rely on complex parameter configurations and on an in-depth understanding of the underlying architectural choices. Although extensive research has been devoted to improving and evaluating the performance of such analytics frameworks, most of them benchmark the platforms against Hadoop, as a baseline, a rather unfair comparison considering the fundamentally different design principles. This paper aims to bring some justice in this respect, by directly evaluating the performance of Spark and Flink. Our goal is to identify and explain the impact of the different architectural choices and the parameter configurations on the perceived end-to-end performance. To this end, we develop a methodology for correlating the parameter settings and the operators execution plan with the resource usage. We use this methodology to dissect the performance of Spark and Flink with several representative batch and iterative workloads on up to 100 nodes. Our key finding is that there none of the two framework outperforms the other for all data types, sizes and job patterns. This paper performs a fine characterization of the cases when each framework is superior, and we highlight how this performance correlates to operators, to resource usage and to the specifics of the internal framework design.**

***Index Terms*—Big Data, performance evaluation, Spark, Flink.**

I. Introduction

In the last decade, Big Data analytics has become an indispensable tool in transforming science, engineering, healthcare, finance and ultimately business itself, due to the unprecedented ability to extract new knowledge and automatically find correlations in massive datasets that naturally accumulate in our digital age [1]. In this context, the MapReduce [2] model and its open-source implementation, Hadoop [3], were widely adopted by both industry and academia, thanks to a simple yet powerful programming model that hides the complexity of parallel task execution and fault-tolerance from the users. This very simple API comes with the important caveat that it forces applications to be expressed in terms of map and reduce functions.

However, most applications do not fit this model and require a more general data orchestration, independent of any programming model. For instance, iterative algorithms used in graph analytics and machine learning, which perform several rounds of computation on the same data, are not well served by the original MapReduce model. To address these limitations, a second generation of analytics platforms emerged in an attempt to unify the landscape of Big Data processing. Spark [4] introduced Resilient Distributed Datasets (RDDs) [5], a set of in-memory data structures able to cache intermediate data across a set of nodes, in order to efficiently support *iterative* algorithms. With the same goal, Flink [6] proposed more recently native closed-loop iteration operators [7] and an automatic cost-based optimizer, that is able to reorder the operators and to better support *streaming*.

As a result of the widespread adoption of these frameworks, significant strides have been made towards improving their performance [8], [9], [10], [11], [12]. The vast majority of these works focus on some specific optimizations derived from the common knowledge about the performance of data analytics platforms. Network optimizations target end-to-end throughput [13] or parallel streams [14]. Storage optimizations leverage complex in-memory caches [5], [15]. Most of this research focuses on a particular aspect of the system in isolation, lagging behind in providing a comprehensive understanding on how all these factors impact the end-to-end performance.

This is precisely the goal targeted by this paper, as a first step in understanding how to optimize the runtime middleware for data analytics frameworks. To this end, we introduce a methodology for analyzing the performance of such platforms by means of correlations between the operators execution plan and the resource utilisation. We then use this methodology to study the performance of several batch and iterative processing benchmarks, with an eye on scalability and easiness of configuration. In particular, we focus on Spark and Flink as representative data analytics frameworks, due to the recent interest in their capabilities (both functional and non-functional) over Hadoop MapReduce and its ecosystem. We consider various workloads in our attempt to provide an in-depth performance comparison of Spark and Flink, as previous works typically benchmarked these frameworks against Hadoop, an unfair comparison for the latter considering its key design choices (*e.g.* use of disks, lack of optimizers etc.). With little information existing on Flink versus Spark, our second goal is to assess wether using a single engine for all data sources, workloads and environments [16] is feasible or not, and also to study how well frameworks that depend on smart optimizers work in real life [17].

We summarize our contributions as follows:

- We introduce a methodology to understand performance in Big Data analytics frameworks by *correlating the operators execution plan with the resource utilization and the parameter configuration*.

- We use this methodology to analyze the behaviour of Spark and Flink through a series of *extensive experiments* involving six representative workloads for *batch and iterative processing*. The experiments were conducted on the Grid'5000 [18] testbed, with various cluster sizes of up to 100 nodes.
- We analyze the various architectural choices and pinpoint their impact on performance, we illustrate the limitations and discuss the ease of configuration for both frameworks in a set of *take-aways for users and developers*.

The remainder of this paper is organized as follows. Section II presents Spark and Flink, highlighting their architectural differences. Section III gives the workloads description, while Section IV emphasizes the impact on performance of the right parameter configuration. Section V details our experimental methodology and Section VI presents the results alongside a detailed analysis. Finally, Section VII surveys the related work and Section VIII lists our insights and concludes this study.

II. Context and Background

Spark and Flink facilitate the development of multi-step data pipelines using directly acyclic graph (DAG) patterns. At a higher level, both engines implement a driver program that describes the high-level control flow of the application, which relies on two main parallel programming abstractions: (1) structures to describe the data and (2) parallel operations on these data. While the data representations differ, both Flink and Spark implement similar dataflow operators (*e.g. map*, *reduce*, *filter*, *distinct*, *collect*, *count*, *save*), or an API can be used to obtain the same result. For instance, Spark's *reduceByKey* operator (called on a dataset of key-value pairs to return a new dataset of key-value pairs where the value of each key is aggregated using the given reduce function) is equivalent to Flink's *groupBy* followed by the aggregate operator *sum* or *reduce*.

A. Apache Spark

Spark is built on top of RDDs (read-only, resilient collections of objects partitioned across multiple nodes) that hold provenance information (referred to as lineage) and can be rebuilt in case of failures by partial recomputation from ancestor RDDs. Each RDD is by default lazy (*i.e.* computed only when needed) and ephemeral (*i.e.* once it actually gets materialized, it will be discarded from memory after its use). However, since RDDs might be repeatedly needed during computations, the user can explicitly mark them as persistent, which moves them in a dedicated cache for persistent objects.

The operations available on RDDs seem to emulate the expressivity of the MapReduce paradigm overall, however, there are two important differences: (1) due to their lazy nature, maps will only be processed before a reduce, accumulating all computational steps in a single phase; (2) RDDs can be cached for later use, which greatly reduces the need to interact with the underlying distributed file system in more complex workflows that involve multiple reduce operations.

B. Apache Flink

Flink is built on top of DataSets (collections of elements of a specific type on which operations with an implicit type parameter are defined), Job Graphs and Parallelisation Contracts (PACTs) [19]. Job Graphs represent parallel data flows with arbitrary tasks, that consume and produce data streams. PACTs are second-order functions that define properties on the input/output data of their associated user defined (first-order) functions (UDFs); these properties are further used to parallelize the execution of UDFs and to apply optimization rules [8].

C. Zoom on the Differences between Flink and Spark

In contrast to Flink, Spark's users can control two very important aspects of the RDDs: the persistence (*i.e.* in memory or disk based) and the partition scheme across the nodes [5]. This fine-grained control over the storage approach of intermediate data proves to be very useful for applications with varying I/O requirements.

Another important difference relates to iterations handling. Spark implements iterations as regular for-loops and executes them by loop unrolling. This means that for each iteration a new set of tasks/operators is scheduled and executed. Each iteration operates on the result of the previous iteration which is held in memory. Flink executes iterations as cyclic data flows. This means that a data flow program (and all its operators) is scheduled just once and the data is fed back from the tail of an iteration to its head. Basically, data is flowing in cycles around the operators within an iteration. Since operators are just scheduled once, they can maintain a state over all iterations. Flink's API offers two dedicated iteration operators to specify iterations: 1) *bulk iterations*, which are conceptually similar to loop unrolling, and 2) *delta iterations*, a special case of incremental iterations in which the solution set is modified by the step function instead of a full recomputation. Delta iterations can significantly speed up certain algorithms because the work in each iteration decreases as the number of iterations goes on.

In this paper we aim for a comprehensive understanding of Flink and Spark that would eventually answer the question: *how are these different architectural choices impacting performance?*

III. Workloads

Although they were initially developed to enhance the batch-oriented Hadoop with efficient iterative support, currently Spark and Flink are used conversely for both *batch* and *iterative* processing. Recent extensions brought SQL [20], [21] and streaming [22], [23] support, but their evaluation is out of the scope of this paper and planned for future work. For the batch category we have selected three benchmarks implementing the one-pass processing: Word Count, Grep and Tera Sort. These are representative workloads used in several real-life applications, either scientific (*e.g.* indexing the monitoring data at the Large Hadron Collider [24]) or Internet-based (*e.g.* search algorithms at Google, Amazon [25], [26]). For the iterative category we have opted for three benchmarks that are mainly used to evaluate the effectiveness of the loop-caching: K-Means, Page Rank and Connected Components. These workloads are frequent in machine learning algorithms [27] and social graphs processing (at Facebook [28] or Twitter [29]). Table I lists the use of the most important operators

Operators	Batch (one pass)			Iterative (caching)		
	WC	G	TS	KM	PR	CC
map			✓	✓	✓	✓
flatMap	✓				✓	✓
mapToPair (S)	✓					
groupBy → sum (F)	✓					
reduceByKey (S)	✓			✓		
collectAsMap (S)				✓		
filter → count		✓				
distinct					✓	✓
repartitionAndSort - WithinPartitions (S)			✓			
partitionCustom → sortPartition (F)			✓			
Graph specific operators					✓	✓
coalesce, mapPartitionsWithIndex (S)					✓	✓
DeltaIteration, join, groupBy, aggregate (F)						✓
BulkIteration, groupBy, reduce, withBroadcastSet(F)				✓		
save	✓		✓	✓	✓	✓

TABLE I: Operators used in each workload: Word Count (WC), Grep (G), Tera Sort (TS), K-Means (KM), Page Rank (PR), Connected Components (CC). Operators annotated with F or S are specific only to Flink or Spark respectively, the other ones are common for both frameworks.

by each workload, including basic core operators and specific ones implemented by the graph libraries of each framework.

Word Count is a simple metric for measuring article quality by counting the total number of occurences of each word. It is a good fit for evaluating the *aggregation component* in each framework, since both Spark and Flink use a map side combiner to reduce the intermediate data. In Flink, the following sequence of operators is applied to the DataSets: *flatMap* (map phase) → *groupBy* → *sum* (reduce phase) → *writeAsText*. In Spark, the following sequence is applied to RDDs: *flatMap* → *mapToPair* (map phase) → *reduceByKey* (reduce phase) → *saveAsTextFile*.

Grep is a common command for searching plain-text data sets. Here, we use it to evaluate the *filter* transformation and the *count* action. Both Flink and Spark implement the following sequence of operators applied on their specific datasets: *filter* → *count*. For both Word Count and Grep, Spark's RDDs and Flink's DataSets are built by reading Wikipedia text files from HDFS.

Tera Sort is a sorting algorithm suitable for measuring the performance of the two engines' core capabilities. We have chosen the implementation described in [30] on 100-byte records, with the first 10-bytes representing the sort key. The input data is generated using the *TeraGen* [31] program with Hadoop and the same range partitioner has been used in order to provide a fair comparison. A number of equally sized partitions is generated and a custom partitioner is used based on Hadoop's *TotalOrderPartitioner*. Spark is creating two RDDs: the first one by reading from HDFS and performing a local sort (*newAPIHadoopFile*) and the second one by repartitioning the first RDD according to the custom partitioner (*repartitionAndSortWithinPartition*). Flink first creates a DataSet from the given HDFS input and then applies a *map* to create key-value tuples. These are stored using an *OptimizedText* binary format in order to avoid deserialization when comparing two keys. Next, Flink's algorithm partitions the tuple DataSet (*partitionCustom*) on the specified keys using a custom partitioner and applies a *sortPartition* to locally sort each partition of the dataset. Finally, the results are saved using the same Hadoop output format.

K-Means is an unsupervised method used in data mining to group data elements with a high similarity. The input is generated using the *HiBench* suite [32] (training records with 2 dimensions). In each iteration, a data point is assigned to its nearest cluster center, using a map function. Data points are grouped to their center to further obtain a new cluster center at the end of each iteration. This workload evaluates the effectiveness of the caching mechanism and the basic transformations: *map*, *reduceByKey* (for Flink: *groupBy* → *reduce*), and Flink's *bulk iterate* operator.

Page Rank is a graph algorithm which ranks a set of elements according to their references. For Flink we evaluated the vertex-centric iteration implementation from its *Gelly* [7] library (iteration operators: *outDegrees*, *joinWithEdgesOnSource*, *withEdges*), while for Spark we evaluated the standalone implementation provided by its *GraphX* [33] library (iteration operators: *outerJoinVertices*, *mapTriplets*, *mapVertices*, *joinVertices*, *foreachPartition*).

Connected Components gives an important topological invariant of a graph. For Flink we evaluated the vertex-centric iteration implementation (iteration operators: *mapEdges*, *withEdges*), while for Spark we evaluated the *ConnectedComponents* implementation (iteration operators: *mapVertices*, *mapReduceTriplets*, *joinVertices*) as provided by their respective libraries. In Flink's case, we evaluated a second algorithm expressed using *delta iterations* in order to assess their speedup over classic bulk iterations. Both Page Rank and Connected Components are useful to evaluate the caching and the data pipelining performance. For these two workloads, we have used three real datasets (small, medium and large graphs) to validate different cache sizes.

IV. The Importance of Parameter Configuration

Both frameworks expose various execution parameters, preconfigured with default values and allow a further customization. For every workload, we found that different parameter settings were necessary to provide an optimal performance. There are significant differences in configuring Flink and Spark, in terms of ease of tuning and the control that is granted over the framework and the underlying resources. We have identified a set of 4 most important parameters having a major influence on the overall execution time, scalability and resource consumption. They manage the task parallelism, the network behaviour during the shuffle phase, the memory and the data serialization.

A. Task Parallelism

The meaning and the default values of the parallelism setting are different in the two frameworks. Nevertheless, this is a mandatory configuration for each dataflow operator in order to efficiently use all the available resources. Spark's default parallelism parameter (*spark.def.parallelism*) refers to the default number of partitions in the RDDs returned by various transformations. We set this parameter to a value proportional to the number of cores per number of nodes

multiplied by a factor of 2 to 6 in order to experience with a various number of partitions in RDDs for distributed shuffle operations like *reduceByKey* and *join*. Flink's default parallelism parameter (*flink.def.parallelism*) allows to use all the available execution resources (*Task Slots*). We set this parameter to a value proportional to the number of cores per number of nodes. Therefore, in Flink the partitioning of data is hidden from the user and the parallelism setting can be automatically initialized to the total number of available cores.

B. Shuffle Tuning

One difference in configuring Flink and Spark lies in the mandatory settings for the network buffers, used to store records or incoming data before transmitting or respectively receiving over a network. In Flink these are the *flink.nw.buffers* and represent logical connections between mappers and reducers. In Spark, they are called *shuffle.file.buffers*. We enabled Spark's shuffle file consolidation property in order to improve filesystem performance for shuffles with large numbers of reduce tasks. The default buffer size is pre-configured in both frameworks to 32 KB, but it can be increased on systems with more memory, leading to less spilling to disk and better results. In all our experiments we initialize the Spark shuffle manager implementation to *tungsten-sort*, a memory efficient sort-based shuffle. This is to provide a fair comparison to Flink, which is using a sort-based aggregation component.

C. Memory Management

In Spark, all the memory of an executor is allocated to the Java heap (*spark.executor.memory*). Flink allows a hybrid approach, combining on- and off-heap memory allocations. When the *flink.off-heap* parameter is set to true, this hybrid memory management is enabled, allowing the task manager to allocate memory for sorting, hash tables and caching of intermediate results outside the Java heap. The total allocated memory is controlled in Flink by the *flink.taskmanager.memory* parameter, while the *flink.taskmanager.memory.fraction* indicates the portion used by the JVM heap. In Spark, the fractions of the JVM heap used for storage and shuffle are statically initialized by setting the *spark.storage.fraction* and *spark.shuffle.fraction* parameters to different values so that the computed RDDs can fit into memory and ensure enough shuffling space.

D. Data Serialization

Flink peeks into the user data types (by means of the *TypeInformation* base class for type descriptors) and exploits this information for better internal serialization; hence, no configuration is needed. In Spark, the serialization (*spark.serializer*) is done by default using the Java approach but this can be changed to the *Kryo* serialization library [34], which can be more efficient, trading speed for CPU cycles.

E. Other Settings

We plan to explore, in a future work, other configurations as well: 1) For both Flink and Spark to configure each node's instance to implement more than one worker, each with equal shares of the resources (*e.g.* CPU, Memory) and 2) For each workload to carefully set the parallelism of each operator.

V. Methodology and Experimental Setup

In order to understand the impact of the previous configurations on performance and to quantify the differences in the design choices, we devised the following approach. For both Flink and Spark we plot the execution plan with different parameter settings and correlate it with the resource utilisation. As far as performance is concerned, for this study, we focus on the end-to-end execution time, which we collect using both timers added to the frameworks' source code and by parsing the available logs, and analyze it in the context of strong and weak scalability.

We deploy Flink and Spark on Grid'5000 [18], a large-scale versatile testbed, in a cluster with up to 100 nodes. Each node has 2 CPUs Intel Xeon E5-2630 v3 with 8 cores per CPU and 128 GB RAM. All experiments use a single disk drive with a capacity of 558 GB. The nodes are connected using a 10 Gbps ethernet. We use Flink version 0.10.2 and Spark version 1.5.3, working with HDFS version 2.7.

For every experiment we follow the same cycle. We install Hadoop (HDFS) and we configure a standalone setup of Flink and Spark. We import the analyzed dataset and we execute on average 5 runs for each experiment. For each run we measure the time necessary to finish the execution excluding the time to start and stop the cluster and at the end of each experiment we collect the logs that describe the results. We make sure to clear the OS buffer cache and temporary generated data or logs before a new execution starts. We plot the mean and standard deviation for aggregated values of all nodes for multiple trials of each experiment. We dissect the resource usage metrics (CPU, memory, disk I/O, disk utilization, network) in the operators plan execution.

VI. Results and Discussion

In this section we describe our experience with both frameworks and interpret the results taking into account the configuration settings and the tracked resource usage. For the batch workloads, our goal was to validate strong and weak scalability. For the iterative workloads, we focused on scalability, caching and pipelining performance.

A. Evaluating the Aggregation Component (Word Count)

We first run the benchmark with a fixed problem size per node (Figure 1) with the parameter configuration detailed in Table II and then with a fixed number of nodes and increased datasets (Figure 2).

Number of nodes	**2**	**4**	**8**	**16**	**32**
spark.def.parallelism	192	384	768	1536	1024
flink.def.parallelism	32	64	128	256	512
spark.executor.memory (GB)	22	22	22	22	22
flink.taskmanager.memory (GB)	4	4	4	4	11

TABLE II: Word Count and Grep configuration settings for the fixed problem size per node (24GB). Other parameters: *HDFS.block.size* = 256MB, *flink.nw.buffers* = Nodes*2048, *buffer.size* = 64KB.

Weak Scalability. We observe in Figure 1 that both frameworks scale well when adding nodes, sharing a similar performance for a small number of nodes (2 to 8), but for a larger number (16, and 32), Flink performs slightly better.

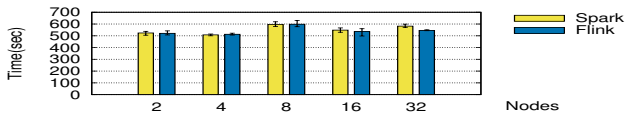


Fig. 1: Word Count - fixed problem size per node (24GB).

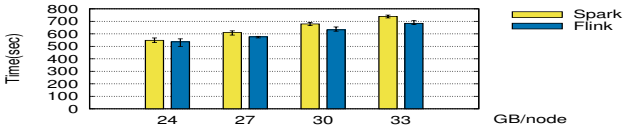
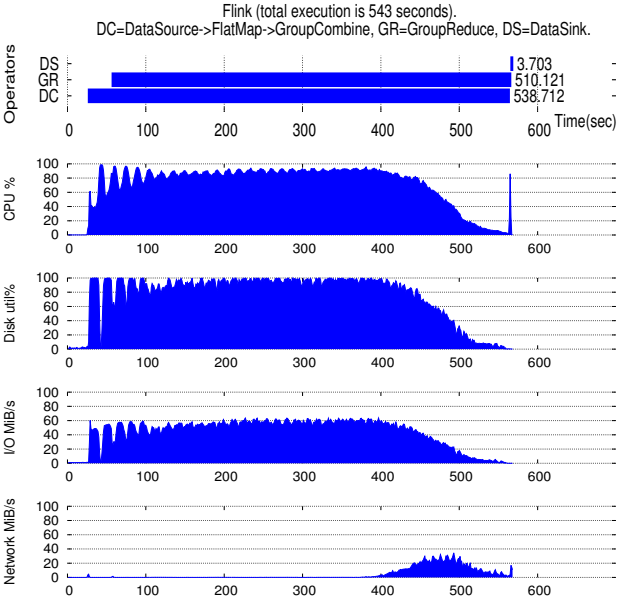


Fig. 2: Word Count - 16 nodes, different datasets.



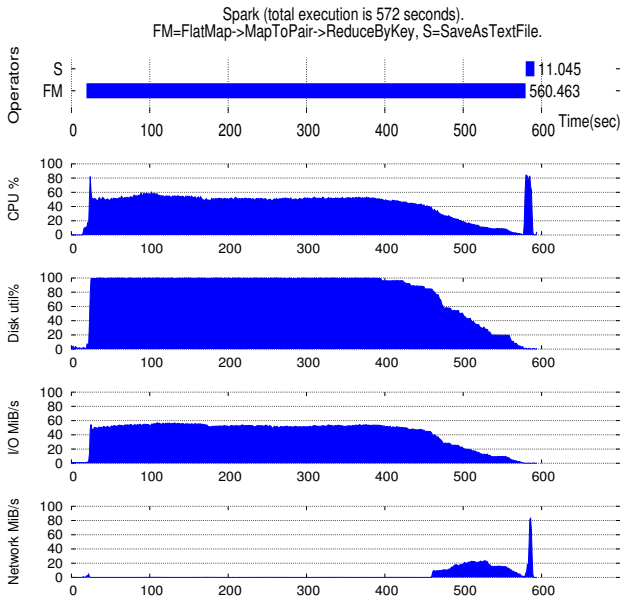


Fig. 3: Word Count resource usage. Flink (left) versus Spark (right), 32 nodes and 768 GB dataset. Similar memory usage, growing linearly up to 30%.

This happens even though, for fairness, we configured Spark with more memory because of its use of the Java serializer.

Strong Scalability. This observation is further confirmed for large datasets and a fixed number of nodes (Figure 2) with Flink constantly outperforming Spark by 10%. Spark's behaviour is influenced by the configured parallelism: the transformation *reduceByKey* that merges the values for each key (locally on each mapper before sending the results to the reducer) hash-partitions the output with the number of partitions (*i.e.* the default parallelism). In fact, for a similar cluster setup (8 nodes) we experimented with a decreased parallelism for Spark (double the number of cores) and obtained an execution time increased by 10%. Flink showed an improved execution when configured with 2 *Task Slots* for each available core. We further analyze the resource usage in order to understand this gap in performance.

Resource Usage. Figure 3 presents the correlation between the operators execution plan and the resource usage for 32 nodes. For this workload both Flink and Spark are CPU and disk-bound. For Flink, we notice an anti-cyclic disk utilization (*i.e.* correlated to the CPU usage: the CPU increases to 100% while the disk goes down to 0%), which is explained by the use of a sort-based combiner for grouping, collecting records in a memory buffer and sorting the buffer when it is filled. CPU-wise, Flink seems more efficient than Spark and also takes less time to save the results with the corresponding action, contributing to the reduced end-to-end execution time.

Discussion. Flink's aggregation component (sort-based combiner) appears more efficient than Spark's, building on its improved custom managed memory and its type oriented data serialization. With its recent *DataSet* API [35] for structured data, Spark seems to aim for a similar approach. Flink is currently investigating the introduction of a hash-based strategy for the *combine* and *reduce* functions, that could yield further improvements (to overcome the anti-cycling effect). Because of Flink's pipeline nature, we had to increase the number of buffers in order to avoid failed executions. There is a correlation between the number of buffers, the parallelism and the workflow's operators. As such, users need to pay attention to the correct configuration.

B. Evaluating Text Search (Grep)

The next benchmark is evaluated in the same scenario: a fixed problem size per node (Figure 4) and a fixed number of nodes for increasing datasets (Figure 5), with the same parameters from Table II.

Weak and Strong Scalability. When increasing the number of nodes, we notice an improved execution for Spark, with up to 20% smaller times for large datasets (16, and 32 nodes). Spark's advantage is preserved over larger datasets as well.

Resource usage. In order to understand this performance gap, we zoom on the main differences in network and disk usage observed in Figure 6. Flink's current implementation of the *filter* $\rightarrow$ *count* operators is leading to inefficient use of the resources in the latter phase.

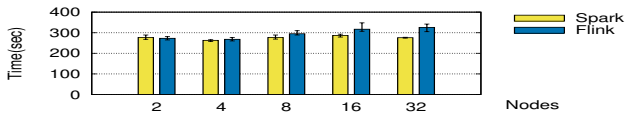


Fig. 4: Grep - fixed problem size per node (24GB).

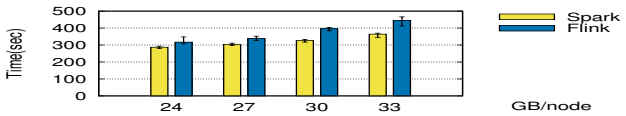
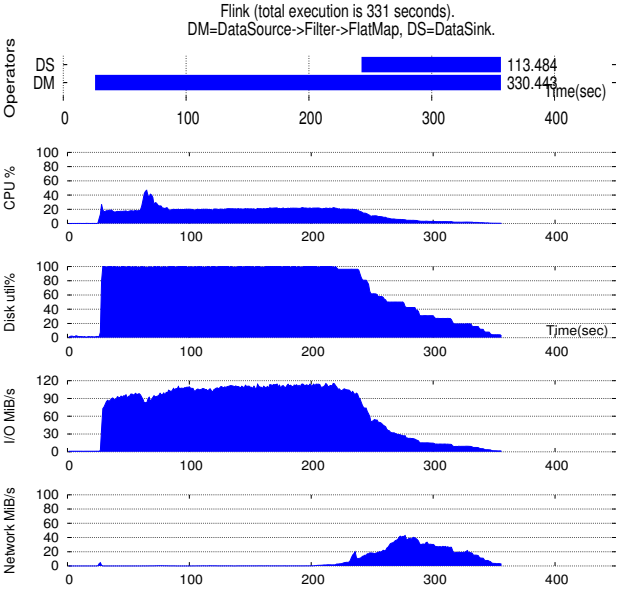


Fig. 5: Grep - 16 nodes, different datasets.



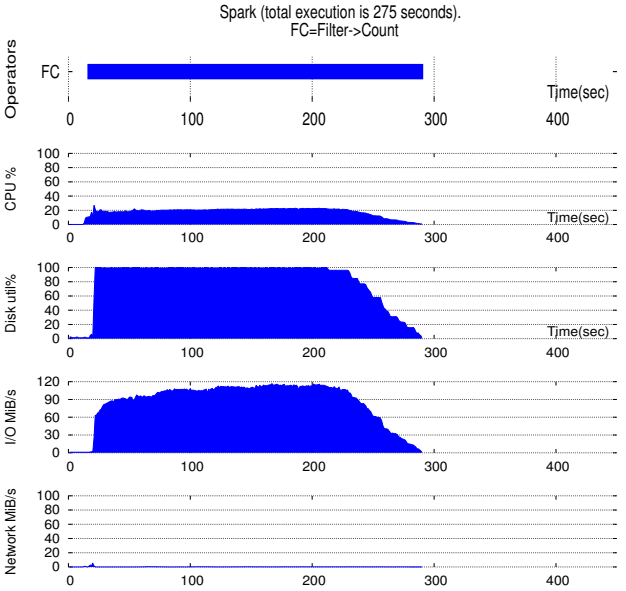


Fig. 6: Grep resource usage. Flink (left) versus Spark (right), 32 nodes and 768 GB dataset. Similar memory usage, growing linearly up to 30%.

Discussion. Although for this workload Spark overperformed Flink, for more complex workflows with multiple filter layers applied on the same dataset, Spark can take more advantage of its persistence control over the RDDs (disk or memory) and further reduce the execution times. This important feature is missing in the current implementation of Flink.

C. Shuffle, Caching and the Execution Pipeline (Tera Sort)

We ran this benchmark with a fixed data size per node (32GB) up to 64 nodes and then for a fixed dataset (3.5TB) with up to 100 nodes (parameter settings in Table III).

Number of nodes	17	34	63	55	73	97
spark.def.parallelism	544	1088	1984	1760	2336	3104
flink.def.parallelism	134	270	500	475	580	750

TABLE III: Tera Sort configuration settings. Both Flink and Spark use 62 GB memory. The number of partitions is equal to the Flink parallelism number. Other parameters: *HDFS.block.size* = 1024MB, *flink.nw.buffers* = Nodes*1024, *buffer.size* =128KB.

Weak Scalability. In Figure 7 we notice that although Flink is performing on average better than Spark, it also shows a high variance between each of the experiments' results, when compared to Spark. This variance may be explained by the I/O interference in Flink's execution due to its pipeline nature. We wanted to check whether Flink's speedup is preserved for a larger dataset processed at each node and we experimented with sorting 75 GB per node using a cluster of 27 nodes (432 cores) and increasing the memory quota for both up to 102 GB). Again, Flink showed 15% smaller execution times.

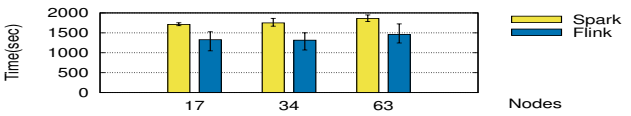


Fig. 7: Tera Sort - fixed problem size per node (32 GB).

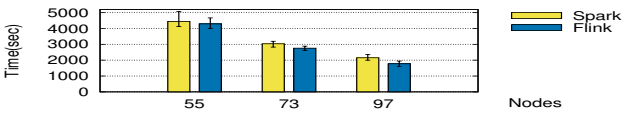


Fig. 8: Tera Sort - adding nodes, same dataset (3.5TB).

Strong Scalability. As seen in Figure 8, Flink's advantage is increasing with larger clusters, which can be explained by less I/O interference caused by a reduced dataset to sort by each node.

Resource usage: Figure 9 presents the correlation between the operators execution plan and the resource usage for sorting 3.5 TB of data on a cluster with 55 nodes. Flink's and Spark's default parallelism settings were initialized to 1760 (twice the number of cores, following Spark's recommendation) and 475 respectively (half the number of cores in order to match the number of custom partitions, otherwise Flink fails due to insufficient task slots), and the number of custom partitions to 475 (*i.e.* Flink's parallelism, for a fair comparison). A few important observations differentiate Flink and Spark executions. First, Flink pipelines the execution, hence it is visualized in a

single stage, while in Spark the separation between stages is very clear. Next, a virtual second stage is observable in both sides which is triggered by the separation of disk I/O read and write processes defined by one process domination along the time axis. Finally, Spark uses less network in this case due to the map output compression.

Discussion. The importance of the execution pipeline implemented by the smart optimizer in Flink is clearly illustrated by this workload. Reordering the operators enables more efficient resource usage and drastically reduces the execution time.

D. Bulk Iterations Performance (K-Means)

We evaluated the loop support in both platforms (Figure 11) for a fixed dataset of 51 GB (1.2 billion edges) needing 10 iterations to stabilise. While both Spark and Flink scale gracefully when adding nodes (up to 24), we notice that Flink's *bulk iterate* operator and its pipeline mechanism outperform by more than 10% the loop unrolling execution of iterations implemented in Spark. As seen in Figure 10, both frameworks have a similar resource usage, CPU-bound when loading the data points and processing the iterations.

E. Graph Processing (Page Rank and Connected Components)

We have selected 3 representative graph datasets (Small [36] and Medium [37] social graphs from Twitter and Friendster respectively, and a Large [38] one, the largest hyperlink graph available to the public) as detailed in Table IV. We executed Page Rank and Connected Components with each graph type and specific parameter configuration, as discussed below.

Graph type	Small [36]	Medium [37]	Large [38]
Nodes / Edges	24.7 M / 0.8 B	65.6 M / 1.8 B	1.7 B / 64B
Size	13.7 GB	30.1 GB	1.2 TB

TABLE IV: Graph datasets characteristics.

Small Graph. With the settings detailed in Table V, we notice a slightly better performance of Flink both for Page Rank (Figure 12) and Connected Components (Figure 13). For Page Rank this was rather surprising, considering that Flink's implementation will first execute a job to count the vertices, reading the dataset one more time in order to load the graph. We experimented with various values of the Spark's *spark.edge.partition* parameter and we obtained a drop in performance when this value is increased (more files to handle) or decreased (inefficient resource usage) for both algorithms. Flink's better performance is mainly backed by its bulk iteration operator and its pipeline nature.

Medium Graph: With the configuration in Table VI and a larger graph we executed Page Rank (Figure 14) and Connected Components (Figure 15). For Flink we experimented with a decreased parallelism setting in order to test the pipeline execution implementation and we observed that during the iteration computation we can obtain a similar performance, but in the load graph phase (including vertices count for Page Rank) the performance drops due to inefficient resource usage. For Spark with 27 nodes and more we had to decrease the number of edge partitions because we experimented with larger values (proportional to the number of cores per number of nodes) for a configuration of 24 nodes and we found a large drop in performance (up to 50%). Flink's Connected Components outperforms Spark by a much larger factor than in the case of Small Graphs (up to 30%) mainly because of its efficient *delta iteration* operator.

Parameter	Formula
spark.def.parallelism	[nodes * cores * 6]
flink.def.parallelism	[nodes * cores]
spark.edge.partition	[nodes * cores]
flink.nw.buffers	[cores * cores * nodes * 16]

TABLE V: Configuration settings for the Small Graph.

Nodes	24	27	34	55
spark.def.parallelism	1440	1620	1632	2640
flink.def.parallelism	288	297	442	715
spark.executor.memory(GB)	22	96	62	62
flink.taskmanager.memory(GB)	18	18	62	62
spark.edge.partition	1440	256	320	480

TABLE VI: Configuration settings for the Medium Graph.

Large Graph. We experimented with the large graph dataset on 3 clusters of 27, 44 and 97 nodes respectively, as shown in Table VII.

Nodes	27		44		97	
Large Graph	**Load**	**Iter.**	**Load**	**Iter.**	**Load**	**Iter.**
Flink PR	no	no	no	no	1096s	645s
Spark PR	3977s	no	667s	no	418s	596s
Flink CC	no	no	no	no	580s	1268s
Spark CC	3717s	3948s	798s	978s	357s	529s

TABLE VII: Page Rank (PR) and Connected Components (CC) with 5 and 10 iterations (Iter.) respectively. Flink's load graph (Load) stage includes the vertices count.

Flink's execution with 27 and 44 nodes failed because of the *CoGroup* operator's internal implementation which computes the solution set in memory. For Spark's Page Rank and Connected Components with 27 and 44 nodes we were able to process correctly the graph load stage only when we doubled the number of edge partitions from a value equal to the total number of cores. The execution of Page Rank and Connected Components with 97 nodes was successful for both frameworks. Flink is less efficient because the parallelism is reduced. In Flink's case, we set the parallelism to three quarters of the total number of cores in order to allocate more memory to each *CoGroup* operator so that the execution does not crash. Setting the parallelism to the total number of cores causes a failure. We suspect two problems: one with the pipeline execution and the I/O interference and second with the inefficient management of a very large number of network buffers.

Resource Usage. As seen in Figure 16 we identified two processing *stages* for Page Rank: the first one is necessary to load the edges and prepare the graph representation, while the second one consists of the iterative processing. In the first stage both Flink and Spark are CPU- and disk-bound, while in the second stage they are CPU- and network-bound. Spark is using disks during iterations in order to materialize intermediate ranks and we observe that the memory increases from one iteration to another, while for iterations in Flink there is no disk usage in case of Page Rank and some disk usage for the first iterations in the case of Connected Components, while

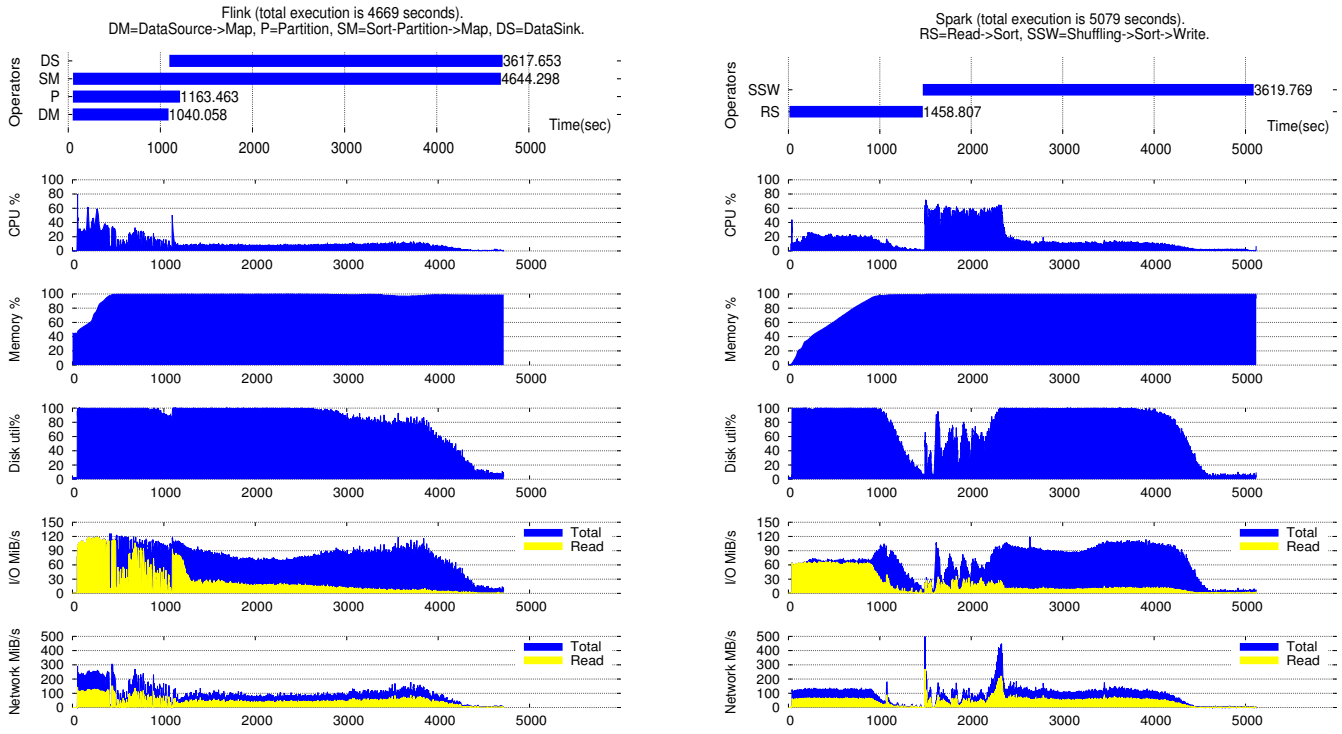


Fig. 9: Tera Sort resource usage of Flink and Spark for 55 nodes and 3.5 TB dataset.

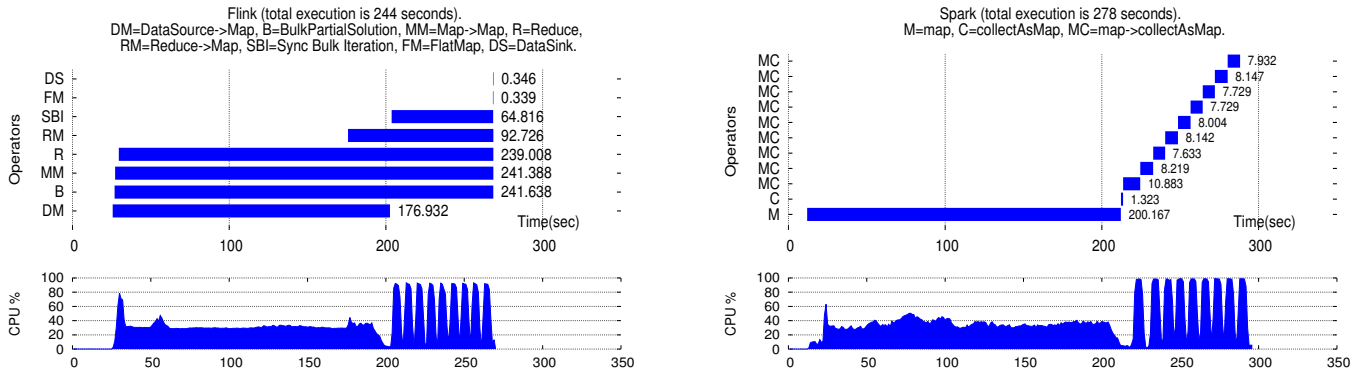


Fig. 10: K-Means resource usage of Flink and Spark for 24 nodes, 10 iterations and 1.2 billion samples dataset. Resource usage not shown for similarity reasons: memory and disk utilization are less than 10%, total disk I/O (r+w) is less than 20MB/s, and total network (r+w) is less than 30MB/s.

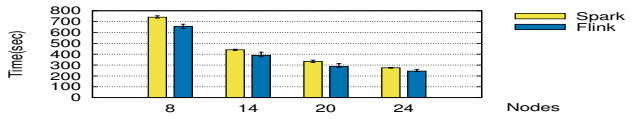


Fig. 11: K-Means - increasing cluster size, same dataset (1.2 billion edges).

the memory remains constant. Flink uses more network during the iterative process and, because of its pipeline mechanism and its bulk iterator operator, it is able to reduce the execution time. Overall, for Connected Components we observe a similar resource usage (Figure 17). However, although Flink's delta iterate operator makes a more efficient use of CPU, it is still forced to rely on disk for iterations on large graphs, hence Spark's better results.

Discussion. Spark needs a careful parameter configuration (for parallelism, partitions etc.), which is highly dependent on the dataset, in order to obtain an optimal performance. In Flink's case, one needs to make sure that enough memory is allocated so that its *CoGroup* operator that builds the solution set in memory could be successfully executed. Applications handling a solution set built with delta iterations should consider the development of a spillable hash table in order to avoid a crash, trading performance for fault tolerance. For such workloads, in which the execution consists of one stage to prepare the graph (load edges) and another one to execute a number of iterations, an optimal performance can be obtained by configuring the parallelism setting of the operators separately for each stage.

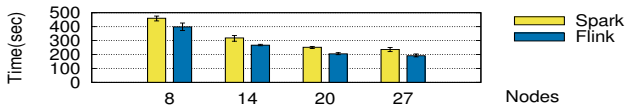


Fig. 12: Page Rank - Small Graph (increasing cluster size).

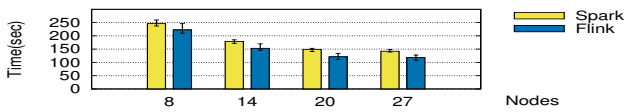


Fig. 13: Connected Components - Small Graph (increasing cluster size).

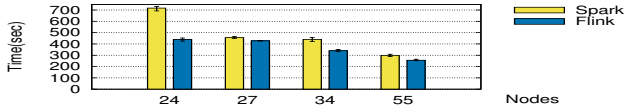


Fig. 14: Page Rank - Medium Graph (increasing cluster size).

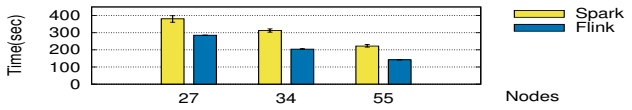
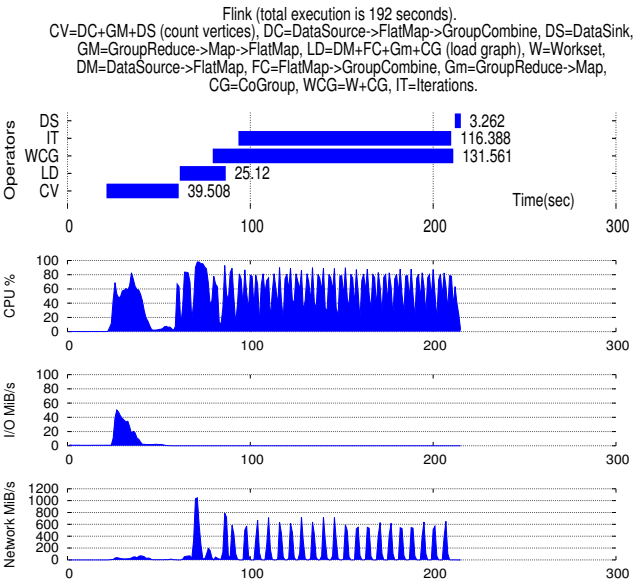


Fig. 15: Connected Components - Medium Graph (increasing cluster size).



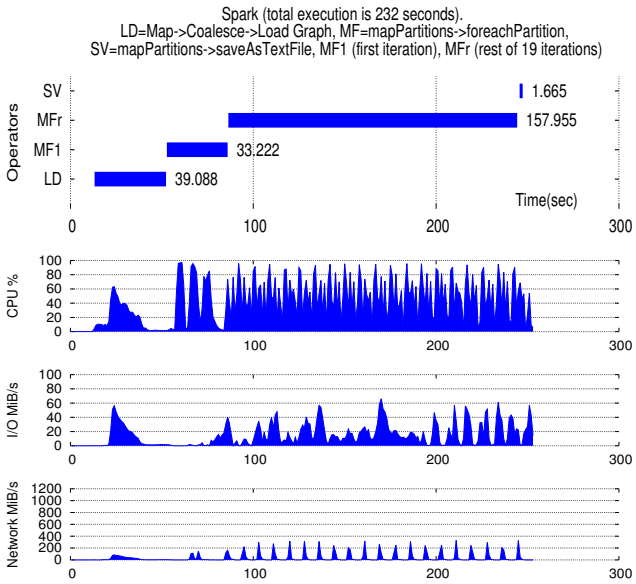


Fig. 16: Page Rank resource usage of Flink and Spark for 27 nodes, 20 iterations, and small graph. Disk utilization is similar to disk I/O, memory is 40%.

VII. Related Work

While extensive research efforts have been dedicated to optimize the execution of MapReduce based frameworks, there has been relatively less progress on identifying, analyzing and understanding the performance issues of more recent data analytics frameworks like Spark and Flink.

Execution optimization. Since the targeted applications are mostly data-intensive, a first approach to improving their performance is to make *network optimizations*. In [13] the authors provide the best parameter combination (*i.e.* parallel stream, disk, and CPU numbers) in order to achieve the highest end-to-end throughput. *Storage optimizations* try either to better exploit disk locality [39] or simply to eliminate the costly disk accesses by complex in-memory caches [5], [15]. In both cases, the resulting aggregated uniform storage spaces will lag behind in widely distributed environments due to the huge access latencies. In [7] the authors analyze the changes needed by the optimizer and the execution engine of Flink in order to support bulk and incremental (delta) *iterations*. Similarly to us, they consider graph processing algorithms like Page Rank when comparing to Spark, but the cluster size is small (hence no intuition about scalability) and they ignore recent improvements in Flink, like the memory management.

Performance evaluation. The vast majority of research in this field focuses on the Hadoop framework, since, for more than a decade, this has become the de-facto industry standard. The problem of how to predict completion time and optimal resource configuration for a MapReduce job was proposed in [40]. To this end, the work introduces a methodology that combines analytical modelling with micro-benchmarking to estimate the time-to-solution in a given configuration. The problem of disproportionately long-running tasks, also called stragglers, has received considerable attention, with many mitigation techniques being designed around *speculative execution* [41]. Other studies focus on the *partitioning skew* [42] which causes huge data transfers during the shuffle phases, leading to significant unfairness between nodes. More recent performance studies specifically target Spark [9]. The authors analyze three major architectural components (shuffle, execution model and caching) in Hadoop and Spark. Similarly to us, they use a *visual tool* to correlate resource utilization with the task execution; however, they do not evaluate the operator parallelism and do not consider Flink with its own cost-based optimizer. *Blocked time analysis* has been introduced in [43]

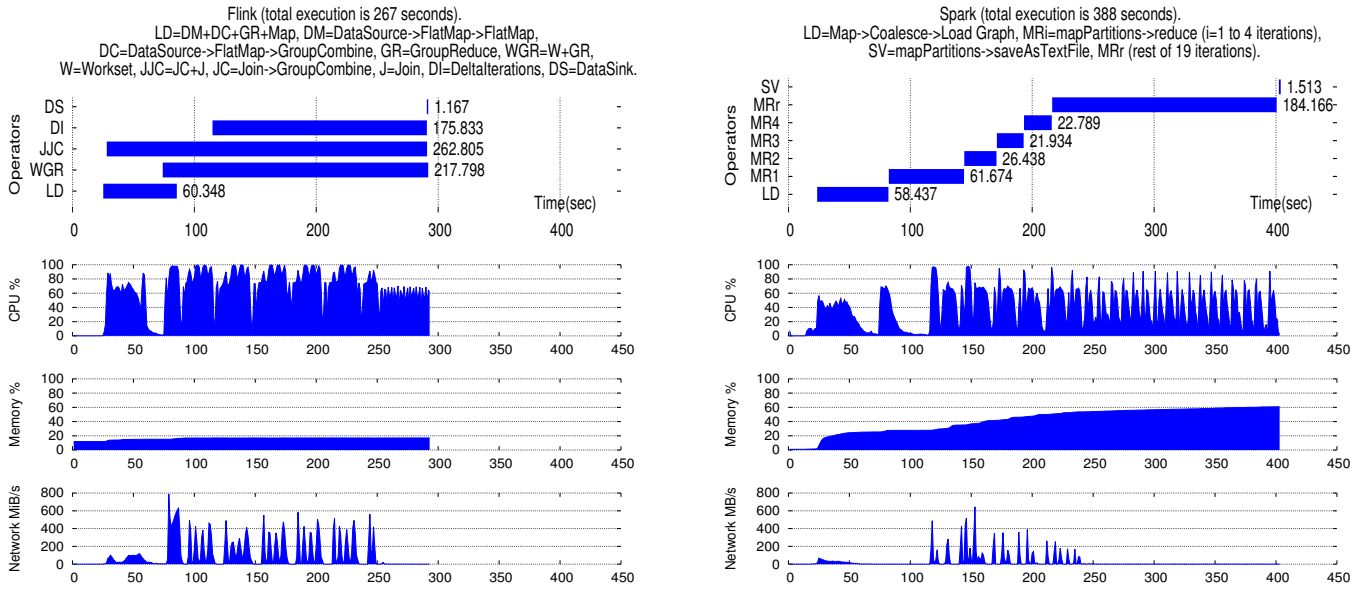


Fig. 17: Connected Components resource usage of Flink and Spark for 27 nodes, 23 iterations, and medium graph.

in order to understand the impact of disk and network and to identify the cause of stragglers. The authors show that in Spark SQL this is due to the Java Garbage Collector and the time to transfer data to and from the disk. This technique could be applied to Flink as well, where stragglers are caused by the I/O interference in the execution pipelines, as seen in the Tera Sort workload from our study.

Overall, most of the previous work typically focuses on some specific low-level issues of big data frameworks that are not necessarily well correlated with the higher level design. It is precisely this gap that we aim to address in this work by linking bottlenecks observed through parameter configuration and low level resource utilization with high-level behavior in order to better understand performance.

VIII. Summary of Insights and Conclusion

Our key finding shows that *there is not a single framework for all data types, sizes and job patterns*: Spark is about 1.7x faster than Flink for large graph processing, while the latter outperforms Spark up to 1.5x for batch and small graph workloads using sensitively less resources and being less tedious to configure. This behaviour is explained by different design choices that we recall below.

Memory management plays a crucial role in the execution of a workload, particularly for huge datasets. While common wisdom on processing lots of data in a JVM means storing them as objects on the heap, this approach has a few notable drawbacks. First, memory overallocation instantly kills the JVM (as seen in the Large graph case from Section VI-E). The overhead of garbage collection on multi-GB JVMs which are flooded with new objects can easily reach 50% and more. Finally, Java objects come with a certain space overhead depending on the JVM and platform. For data sets with many small objects this can significantly reduce the effectively usable amount of memory. Given proficient system design and careful, use-case specific system parameter tuning, heap memory usage can be more or less controlled in order to avoid overallocation. However, such setups are rather fragile especially if data characteristics or the execution environment change. During our experiments we noticed that, as opposed to Spark, Flink does not accumulate lots of objects on the heap but stores them in a dedicated memory region, to avoid overallocation and the garbage collection issue. All operators are implemented in such a way that they can cope with very little memory and can spill to disk. We also observed that although Spark can serialize data to disk, it requires that (significant) parts of the data to be on the JVM's heap for several operations; if the size of the heap is not sufficient, the job dies. Recently, Spark has started to catch up on these memory issues with its Tungsten [44] project, highly inspired from the Flink model, for the explicit custom memory management aiming to eliminate the overhead of the JVM object model and garbage collection.

The pipelined execution brings important benefits to Flink, compared to the staged one in Spark. There are several issues related to the pipeline fault tolerance, but Flink is currently working in this direction [45].

Optimizations are automatically built-in Flink. Spark batch and iterative jobs have to be manually optimized and adapted to specific datasets through fine grain control of partitioning and caching. For SQL jobs however, SparkSQL [21] uses an optimizer that supports both rule- and cost-based optimization.

Parameter configuration proves tedious in Spark, with various mandatory settings related to the management of the RDDs (*e.g.* partitioning, persistence). Flink requires less configuration for the memory thresholds, parallelism and network buffers, and none for its serialization (as it handles its own type extraction and data representation).

Identifying and understanding the impact of the different architectural components and parameter settings on the resource usage and, ultimately, performance, could trigger a new

wave of research on *self-configuring* and *auto-optimizing* data analytics frameworks. By necessity, our study did not look at a vast range of workloads. In our investigation we increased the scale as needed in order to highlight the differences in execution of the two engines. These results can bring further motivation to the new Apache Beam [46] initiative which is proposing an unified programming model based on [47], that can be used in order to execute data processing pipelines on separated distributed engines like Spark and Flink.

As future work, we plan to extend the evaluation with SQL and streaming benchmarks, and examine in this context whether treating batches as finite sets of streamed data pays off.

ACKNOWLEDGMENT

This work is part of the BigStorage project, funded by the European Union under the Marie Sklodowska-Curie Actions (H2020-MSCA-ITN-2014-642963). The experiments presented in this paper were carried out on the Grid'5000 testbed [48].

REFERENCES

[1] K. M. Tolle *et al.*, "The fourth paradigm: Data-intensive scientific discovery," *Proceedings of the IEEE*, vol. 99, pp. 1334–1337, 2011.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on OSDI*. USENIX Association, 2004.

[3] "Apache Hadoop." http://hadoop.apache.org.

[4] "Apache Spark." http://spark.apache.org.

[5] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on NSDI*. USENIX Association, 2012.

[6] "Apache Flink." http://flink.apache.org.

[7] S. Ewen *et al.*, "Spinning fast iterative data flows," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1268–1279, Jul. 2012.

[8] A. Alexandrov *et al.*, "MapReduce and PACT - comparing data parallel programming models," in *Proceedings of the 14th Conference on Database Systems for BTW*. Bonn, Germany: GI, 2011, pp. 25–44.

[9] J. Shi *et al.*, "Clash of the titans: Mapreduce vs. spark for large scale data analytics," *Proc. VLDB Endow.*, vol. 8, pp. 2110–2121, Sep. 2015.

[10] D. Warneke and O. Kao, "Exploiting dynamic resource allocation for efficient parallel data processing in the cloud," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 6, pp. 985–997, Jun. 2011.

[11] M. Armbrust *et al.*, "Scaling spark in the real world: Performance and usability," *Proc. VLDB Endow.*, vol. 8, pp. 1840–1843, Aug. 2015.

[12] A. Alexandrov *et al.*, "The stratosphere platform for big data analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, Dec. 2014.

[13] E. Yildirim and T. Kosar, "Network-aware end-to-end data throughput optimization," in *Proceedings of the first international workshop on Network-aware data management*, NY, USA, 2011, pp. 21–30.

[14] T. J. Hacker *et al.*, "Adaptive data block scheduling for parallel tcp streams," in *HPDC-14. Proceedings. 14th IEEE International Symposium on HPDC.*, 2005, pp. 265–275.

[15] H. Li *et al.*, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC. New York: ACM, 2014, pp. 6:1–6:15.

[16] "New directions for Apache Spark in 2015," http://www.slideshare.net/databricks/new-directions-for-apache-spark-in-2015.

[17] "Big Data Digest: How many Hadoops do we really need?" http://www.computerworld.com/article/2871760/big-data-digest-how-many-hadoops-do-we-really-need.html.

[18] "Grid'5000." www.grid5000.fr.

[19] D. Warneke and O. Kao, "Nephele: Efficient parallel data processing in the cloud," in *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*. NY, USA: ACM, 2009.

[20] A. Heise *et al.*, "Meteor/sopremo: An extensible query language and operator model," in *Proceedings of the Int. Workshop on End-to-End Management of Big Data (BigData) in conjunction with VLDB*, 2012.

[21] M. Armbrust *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD Int. Conf. on Management of Data*. NY, USA: ACM, 2015, pp. 1383–1394.

[22] B. Lohrmann *et al.*, "Nephele streaming: Stream processing under qos constraints at scale," *Cluster Computing*, vol. 17, pp. 61–78, 2014.

[23] M. Zaharia *et al.*, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM SOSP*. NY, USA: ACM, 2013, pp. 423–438.

[24] "Large Hadron Holider." http://home.cern/topics/large-hadron-collider.

[25] "Google Algorithms and Theory," http://research.google.com/pubs/AlgorithmsandTheory.html.

[26] "Keys to Understanding Amazons Algorithms," http://www.thebookdesigner.com/2013/07/amazon-algorithms/.

[27] "Algorithm notes," https://azure.microsoft.com/en-us/documentation/articles/machine-learning-algorithm-choice/#algorithm-notes.

[28] "Facebook." https://www.facebook.com/.

[29] "Twitter." https://twitter.com/.

[30] "Tera Sort." http://eastcirclek.blogspot.fr/2015/06/terasort-for-spark-and-flink-with-range.html.

[31] "Hadoop TeraGen for TeraSort," https://hadoop.apache.org/docs/r2.7.1/api/org/apache/hadoop/examples/terasort/package-summary.html.

[32] "Hibench suite." https://github.com/intel-hadoop/HiBench.

[33] J. E. Gonzalez *et al.*, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Conference OSDI*. Berkeley: USENIX Association, 2014, pp. 599–613.

[34] "Kryo," https://github.com/EsotericSoftware/kryo.

[35] "Introducing Spark Datasets." https://databricks.com/blog/2016/01/04/introducing-spark-datasets.html.

[36] "Small graph." http://an.kaist.ac.kr/~haewoon/release/twitter_social.

[37] "Medium graph." http://snap.stanford.edu/data/com-Friendster.html.

[38] "Large graph." http://webdatacommons.org/hyperlinkgraph.

[39] R. Tudoran *et al.*, "Tomusblobs: Towards communication-efficient storage for mapreduce applications in azure," in *CCGrid, 12th IEEE/ACM International Symposium*, May 2012, pp. 427–434.

[40] F. Clemente *et al.*, "Enabling Big Data Analytics in the Hybrid Cloud using Iterative MapReduce," in *UCC'15: The 8th IEEE/ACM Intl. Conf. on Utility and Cloud Computing*, Limassol, Cyprus, 2015.

[41] G. Ananthanarayanan *et al.*, "Grass: Trimming stragglers in approximation analytics," in *Proceedings of the 11th USENIX Conf. NSDI*. Berkeley, CA, USA: USENIX Association, 2014, pp. 289–302.

[42] Y. Kwon *et al.*, "Skewtune: Mitigating skew in mapreduce applications," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. NY, USA: ACM, 2012, pp. 25–36.

[43] K. Ousterhout *et al.*, "Making sense of performance in data analytics frameworks," in *Proceedings of the 12th USENIX Conf. NSDI*. Berkeley, CA, USA: USENIX Association, 2015, pp. 293–307.

[44] "Project Tungsten." https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[45] "Flink-2250." https://issues.apache.org/jira/browse/FLINK-2250.

[46] "Apache Beam." http://beam.apache.org.

[47] T. Akidau *et al.*, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1792–1803, Aug. 2015.

[48] R. Bolze *et al.*, "Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed," *International Journal of High Performance Computing Applications*, pp. 481–494, 2006.