IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received November 6, 2020, accepted December 4, 2020, date of publication December 10, 2020, date of current version December 24, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3043948

DSPBench: A Suite of Benchmark Applications for Distributed Data Stream Processing Systems

MAYCON VIANA BORDIN[1], DALVAN GRIEBLER[2,3], GABRIELE MENCAGLI[4], CLÁUDIO F. R. GEYER[1], AND LUIZ GUSTAVO L. FERNANDES[2]

[1]Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS), Porto Alegre 91509-900, Brazil
[2]School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre 90619-900, Brazil
[3]Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty, SETREM, Três de Maio 98910-000, Brazil
[4]Department of Computer Science, University of Pisa, 56127 Pisa, Italy

Corresponding author: Maycon Viana Bordin (mayconbordin@gmail.com)

This work was supported in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior (CAPES), Brazil, under Finance Code 001, in part by the FAPERGS 01/2017-ARD Project ParaElastic under Grant 17/2551-0000871-5, in part by the FAPERGS 05/2019-PQG Project ParAS under Grant 19/2551-0001895-9, in part by the Universal MCTIC/CNPq 28/2018 Project SParCloud under Grant 437693/2018-0, and in part by the European H2020 Project TEACHING under Grant 871385.

ABSTRACT Systems enabling the continuous processing of large *data streams* have recently attracted the attention of the scientific community and industrial stakeholders. *Data Stream Processing Systems* (DSPSs) are complex and powerful frameworks able to ease the development of streaming applications in distributed computing environments like clusters and clouds. Several systems of this kind have been released and currently maintained as open source projects, like Apache Storm and Spark Streaming. Some benchmark applications have often been used by the scientific community to test and evaluate new techniques to improve the performance and usability of DSPSs. However, the existing benchmark suites lack of representative workloads coming from the wide set of application domains that can leverage the benefits offered by the stream processing paradigm in terms of near real-time performance. The goal of this article is to present a new benchmark suite composed of 15 applications coming from areas like Finance, Telecommunications, Sensor Networks, Social Networks and others. This article describes in detail the nature of these applications, their full workload characterization in terms of selectivity, processing cost, input size and overall memory occupation. In addition, it exemplifies the usefulness of our benchmark suite to compare real DSPSs by selecting Apache Storm and Spark Streaming for this analysis.

INDEX TERMS Data stream processing, big data, benchmarking, apache storm, spark streaming.

I. INTRODUCTION

We are witnessing the exponential growth of data available from different kinds of sources (e.g., sensors, financial tickers, social media) often producing information in the form of *streams* [1], i.e. unbounded sequences of data items received at variable speed. Consequently, there has been a large scientific effort in the design and development of tools for processing streams in a near real-time fashion to extract actionable intelligence.

Although this recent interest, the Data Stream Processing (DSP) topic has been studied for years. First-generation Data Stream Processing Systems (DSPSs) like Aurora [2], Borealis [3], STREAM [4], and StreamIt [5] have been originated from the Database community, and are designed to execute relational algebra queries on data streams rather than on finite and permanent relations (tables). This experience has opened the domain of *streaming analytics*, with tools and languages (e.g., SQL dialects like CQL [6]) to process streams of structured records (often called *tuples*) and to process them on-the-fly to calculate online statistics.

As opposed to stream processing, batch processing assumes permanent static inputs available from distributed filesystems or in memory, and has been made available to programmers with well-known frameworks like MapReduce and Hadoop. Over the last years, to complement such set of programming tools for Big Data and data-intensive computing, some second-generation DSPSs. (e.g., Apache Storm [7], Apache Flink [8], Spark Streaming [9]) have been released as open source projects, often under the Apache umbrella.

The associate editor coordinating the review of this manuscript and approving it for publication was Eyhab Al-Masri.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

Such systems have two distinguishable features with respect to first-generation DSPSs. First, they are designed for distributed systems, and hide the complexity of designing a distributed application to the final users. Second, they target streaming applications not only in the domain of streaming analytics, but a broader range of streaming workloads dealing with structured or unstructured data (e.g., texts and images) can be modeled and developed using their high-level API. In most cases, this API allows building *data-flow graphs* of streaming *operators* (tasks), which can be instantiated with general user-defined code. Furthermore, each operator can be internally replicated (according to its *parallelism level*), in order to increase the processing throughput.

While the design of optimizations for DSPSs is a quite hot topic in recent research, there is still the need of finding representative streaming workloads to assess the full potential of new streaming techniques and optimization strategies. The existing applications commonly used in the research literature (e.g., LinearRoad [10] or the RIoTBench suite [11]) mainly focus on the domain of relational algebra queries and streaming analytics. However, they do not exhibit a high variety of features in terms of their workload characterization and complexity of the data-flow graphs. A more complete review of the existing suites is given in Sect. II.

The main contribution of this article is to provide a new benchmark suite (called *DSPBench*) of 15 applications coming from different areas and all needing the processing features offered by modern DSPSs. In some cases, such applications are derived from prior works, while in others we have designed the application structure from scratch. This results in a new rich suite of applications (the largest as far as we known) that we publicly provide to the community in a GitHub repository.[1] The second goal is to create a low-level API for the unified development of applications, enabling them to be written once and ran anywhere as long as the specific adapter components have been developed as well. Finally, we exemplify the methodology based on our suite to assess and compare the performance of two popular DSPSs (Apache Storm and Spark Streaming), providing insights into how DSPBench is capable of enabling a valuable comparison between systems.

We claim that DSPBench can become a representative benchmark suite for DSPSs in the future, and some of the applications of our suite have already been used by some existing research papers [12]–[14].[2]

This article is organized as follows. Sect. II reviews the existing benchmark suites in the DSP domain. Sect. III shows the architecture of our suite and its components. Sect. IV presents the applications in DSPBench, with the description of how we chose the different application areas, the description of each application and of their features. Sect. V presents a full workload characterization of all the 15 applications. Sect. VI exemplifies the use of our suite (with a subset of its applications) for the evaluation of two popular DSPSs: Apache Storm and Spark Streaming. Finally, Sect. VII draws the conclusion of this article.

II. RELATED WORKS

In this section, we review the literature closely related to this article. We introduce what are the main benchmark suites available to the DSP community and which are their features in terms of the presence of real-world applications, synthetic ones, and whether a full workload characterization has been proposed alongside the presentation of the suites. In this analysis, we highlight why a new suite is needed, and the reasons because our new DSPBench attempts to fill the existing gaps.

A. EXISTING BENCHMARKS

One of the first proposed applications was the Linear Road Benchmark [10] (LR). It simulates a tolling system in a fictitious city with the purpose of calculating the toll values based on traffic jams and accident proximity. The original suite consists of five queries where only one (named Toll Notification) is a continuous query whose results are constantly updated based on the contents of the data, while the others are historical queries based on events stored in a database (so no streaming). For benchmarking purposes, the LR queries have recently been fused in a single application [12], in order to compare Apache Storm and Flink with a more complex composition and interconnection of operators. A widely adopted application is the Yahoo Streaming Benchmark [15] (YSB). It is another relational algebra query simulating an advertisement campaign, where records grouped by the same campaign identifier are counted in time windows of 10 seconds. Although the simplicity of this application, which includes a filtering phase and a join with a static table, it is still used in recent papers [16], [17] to test and compare prototypes of new DSPSs.

BigDataBench [18] is a more recent suite of big data applications. One of its merits is to include applications beyond the set of relational algebra queries (search engines, e-commerce, and social networks). However, most of the applications are not for live streaming tasks, but they model offline workloads that belong to the batch processing domain (i.e. they are implemented in Spark and Hadoop). The suite is currently maintained [19], and its last version (v5.0) consists of 27 benchmarks from different domains (e.g., offline and graph analytics, NoSQL and data warehousing). Only one benchmark (grep) is provided in a streamed version with Spark Streaming.

StreamBench [20] defines seven programs to test Apache Storm and Spark Streaming. They have been designed after a workload characterization in three dimensions: data type, complexity, and use of historical data. The resulting applications still have a very simple form (four of them are based on a single operator, even stateless), which is the main downside

[1] DSPBench source code is available here: https://github.com/GMAP/DSPBench

[2] BriskStream repository available here https://github.com/Xtra-Computing/briskstream gives credit to our preliminary version of some DSPBench applications available here https://github.com/mayconbordin/storm-applications.

TABLE 1. Comparison between the existing benchmark suites of data stream processing applications. DSPBench is the new suite proposed in this article. For each suite, we considered only streaming applications (continuous queries), while in some cases the suites provide also applications and their characterization for non-streamed tasks.

Benchmark Suite	Real-world Applications	Synthetic Applications	Metrics	DSPSs	Unified API	Workload Charact.
Linear Road Benchmark	1	-	latency	Aurora, STREAM	No	No
Yahoo Streaming Benchmark	1	-	throughput, latency	Storm, Flink, Spark Streaming	No	No
BigDataBench	-	1	wall clock time, energy	Spark Streaming	No	No
StreamBench	-	7	throughput, latency, fault tolerance, durability	Storm, Spark Streaming	No	Yes
RIoTBench	4	-	throughput, latency, cpu/mem. usage, jitter	Storm	Yes	Yes
HiBench	-	4	latency	Storm, Flink, Spark Streaming	No	No
DSPBench	**13**	**2**	**throughput, latency, cpu/mem/net usage**	**Storm, Spark Streaming**	**Yes**	**Yes**

of this suite. Only three applications are graphs of multiple interconnected operators. However, they still perform simple computations (like calculating the maximum, minimum, sum, and average of streaming inputs) and graphs are only linear chains (pipelines) of operators. The suite is presented by highlighting that its applications can be implemented in any DSPSs provided that they are adapted to their specific programming interfaces. So, a unified API seems to be not provided. Furthermore, in their analysis, the authors propose specific metrics to evaluate the impact of fault tolerance support on the performance of DSPSs. We do not emphasize this aspect in our evaluation, although this can be done in the future starting from the same metrics used by StreamBench.

An interesting streaming suite is RIoTBench [11], which defines 27 basic Internet of Things (IoT) tasks (e.g., filtering, pattern matching, statistical analysis) that can be combined to create micro-benchmarks. The authors propose four applications composed of a subset of those tasks. This suite is powerful and interesting since the task API has been designed to be implemented in any DSPS. However, the suite evaluation is performed only on Apache Storm, and the task selection has been made in the field of IoT only, which was the main goal of that work. Furthermore, although the nature of the proposed tasks is well discussed in the paper, their workload characterization is done in terms of operator selectivity (number of outputs per input) and presence of an internal state, while it could be extended with a more precise analysis in terms of input size and processing time per input that is missing.

HiBench [21] is a suite originally developed to test batch processing computing platforms like MapReduce and Hadoop. The applications are designed to stress various components like HDFS throughput based on the different data-access patterns exhibited by the proposed workloads. Only recently the suite has been extended to cover DSPSs, with four micro-benchmarks (wordcount, fixed window, identity, and repartition) used to evaluate specific aspects of streaming engines like changing the parallelism of a streaming task. The micro-benchmarks have been implemented in Apache Storm, Flink, and Spark Streaming. Although the suite is dense of applications for the batch processing part, the streaming suite is quite small and does not contain complex applications involving the interconnection of several streaming tasks.

B. COMPARISON OF BENCHMARK SUITES

The suites described before partially cover the full potential and expressive power of modern DSPSs. In most cases, they either focus on a single real-world application or they use a set of synthetic/micro- benchmark applications. Furthermore, only a few suites propose a precise workload characterization. Indeed, the relevance of a benchmark suite depends on the applications selected, as they need to represent the most significant use cases as well as a broad and diverse range of behaviors. In the specific domain of DSP, the workload characterization should take into account different aspects that we will describe in more detail in Sect. V.

Table 1 reports the features of the existing suites The reader can derive the structure of each suite in terms of real-world applications (if any) and micro-benchmarks, the DSPSs used in the analysis, the presence of an API that unifies the development of the applications in various systems, and the metrics used in the analysis. The last row in the table reports the features of our DSPBench, which enriches the panorama of existing benchmark suites both in terms of the number of real-world applications available, and in terms of their workload characterization and presence of a unified API to ease the development.

III. DSPBench ARCHITECTURE

In this section, we describe the architecture of DSPBench based on the components shown in Figure 1. One of the main goals of our suite is to enable the comparison among different DSPSs, not only existing ones but also next-generation

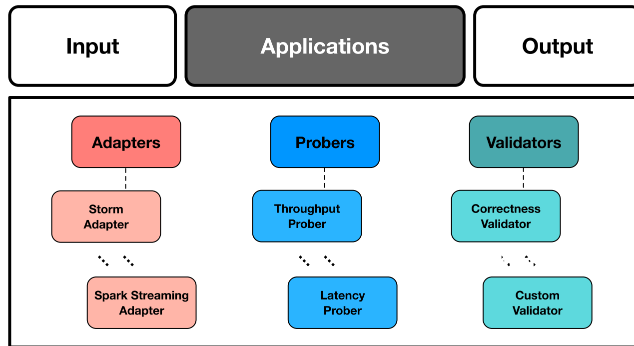


FIGURE 1. General architecture of DSPBench.

DSPSs that will be developed in the future. Ensuring that each application has the same behavior across systems requires some effort. This is because each DSPS exposes features in a quite different way. For example, the API to create operators and to connect them in complex data-flow graphs with different distribution policies is different between DSPSs. Analogously, the parallelism level of each operator can be specified in different ways, e.g., in the declaration of operators in Storm.

The translation of the application-agnostic code to the DSPS-specific code is done by components called *adapters*. An adapter is responsible for calling the API of the specific DSPS and adjusting the application logic accordingly. By standardizing the development of applications, it becomes possible to insert *probes* in the operators and tuples for monitoring purposes. These *probers* can inject timestamps in the tuples to track the latency, they can count the number of received and sent tuples of an operator, as well as how much time is required for processing one tuple in an operator and subsequently calculating its instant throughput. The probers used by default in the suite are for collecting throughput, latency, and resource (CPU, memory, and network) usage. Furthermore, specific components (*validators*) verify the correctness of the results.

Two other components of the architecture are the *Input* and *Output* ones. The first is responsible for feeding the applications with data streams while, the second one, for storing the results so that they can be verified. The Input component has been implemented using Apache Kafka [22], a publish-subscribe distributed middleware. However, this does not prevent others from using the benchmark suite with a different messaging system (e.g., RabbitMQ or ActiveMQ) by re-implementing the Input component. Similar considerations can be made for the Output component, which should be based on a storage system able to cope with the throughput provided by a DSPS. We decided to use Cassandra [23] to store outputs arriving at the sinks of our applications. Also in this case this choice can be reconsidered by changing the implementation of the Output component only.

IV. DSPBench APPLICATIONS

This section describes the applications in DSPBench. We first outline the criteria chosen for the application selection. Then, we describe each application in terms of its operator graph and features (communication and role).

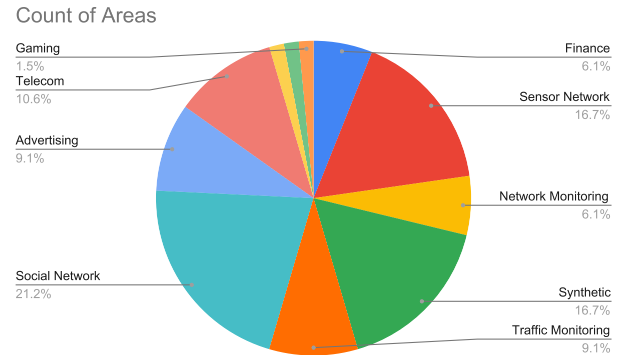


FIGURE 2. Identified areas of interests in our bibliography search over the last eleven years.

TABLE 2. Papers used in our applications selection.

Area	Papers
Finance	[25], [26], [27], [28]
Network Monitoring	[29], [30], [31], [32]
Synthetic	[21], [33], [34], [35], [36], [37]
Traffic Monitoring	[38], [39], [40], [41], [42]
Advertising	[29], [30], [15], [43], [44], [45], [46]
Sensor Network	[47], [48], [49], [50], [51], [52], [11], [53]
Social Network	[54], [55], [56], [57], [58], [59], [60], [61], [62], [18]
Telecom	[63], [64], [65], [66], [67]
Gaming	[68]

A. APPLICATION DOMAINS

The relevance of a benchmark suite is strictly dependent on how its applications have been chosen. The goal in DSPBench was to select applications covering a wide spectrum of areas in the DSP domain. To this end, we searched for papers describing new DSPSs, performance comparisons between systems, and use cases. We applied an in-depth search of the literature over the last eleven years based on both analytical and browsing approaches [24]. Several query terms were formulated as well as synonyms and variations in order to retrieve papers matching those queries. Furthermore, an initial set of papers were selected, and starting from them a path was followed through their bibliography, until the point where no more new papers were found. The main areas identified from the set of 50 resulting papers were: sensor networks, advertising, finance, telecommunication, social networks, synthetic applications, and network monitoring. The percentage of papers in our set belonging to the different areas is depicted in Figure 2 and the complete list can be found in Table 2.

The choice of the applications requires the use of a suitable *workload characterization* in order to discard those with very similar behavior. In past works [69], two techniques are employed to collect information enabling such characterization: performance measurement instrumentation and source code analysis. While the first requires a modification of

the code to allow the collection of statistics at run time, the second one is fully static. We adopt both approaches in DSPBench.

In addition, we need to identify the relevant classification dimensions of DSP applications. Past results are the ones from the topic of scheduling algorithms for DSPSs, as they usually estimated the cost of applications/operators to build an execution plan accomplishing a target Quality of Service (QoS). In [70] the authors propose a scheduling algorithm able to reduce the latency and memory consumption with results very close to the optimal. The importance of memory usage was also acknowledged by the early schedulers for first-generation DSPSs [71]. Thus, *memory usage* is an aspect that has to take part in the characterization criteria.

Early works like [72] characterized the parameters affecting the operator cost. They are: *i)* the input size, *ii)* the selectivity, *iii)* the average time to execute the user-function on each input, *iv)* the time to send outputs to destination operators. Such information can be obtained by instrumenting the code to collect such statistics.

The other two application-wise aspects need to be included in the approach. The first is the identification of the *operator types* involved in an application. A classification of operators can be easily done based on relational algebra for stream analytics (e.g., selection, projection, join, aggregates, group-by). For streaming applications in other areas, we classify operators based on their selectivity (filter, map, and flat-map) and the presence of an internal state. The second is related to the *number of operators* per application and their *communication pattern*, i.e. the form of the data-flow graph. At least two categories can be identified: applications based on linear chains (pipelines) of operators, and applications with more complex acyclic graphs, where an operator can apply several distribution strategies to different subscribers. Furthermore, each distribution between operators can have different semantics. Three are the most used: *i) shuffle* means that outputs of an operator are delivered to the next operator, and internally to one of its replicas, randomly; *ii) group-by field* means that all the inputs with the same key attribute(s) are delivered to the same replica of the next operator; *iii) broadcast* means that each input is delivered to all the replicas of the next operator. The identification of these aspects can be made by a source code analysis as well as through code instrumentation.

Table 3 summarizes the different aspects of our workload characterization. We will describe the results later in the next section. We point out that our approach extends past works about benchmark suites for streaming applications. Indeed, the characterization proposed in StreamBench [20] was limited to three dimensions (data type, presence of multiple operators, and use of historical data), while RIoTBench [11] focuses on applications in the IoT domain only.

B. THE BENCHMARK SUITE

We describe the 15 applications in DSPBench with a description for each one, a figure showing the data-flow graph and communication/operator type. Over the connection between consecutive operators, we report the distribution type (no label is shown for shuffle distributions). Furthermore, we will show for each operator a unique identifier (ID) that will be used in the results of the workload characterization (in Section V).

TABLE 3. Workload characterization aspects in our approach.

Aspect	Technique
Memory Usage	source code analysis, instrumentation
Input Size	instrumentation
Operator Selectivity	instrumentation
Operation Cost	instrumentation
Operator Type	source code analysis
Communication Pattern	source code analysis, instrumentation

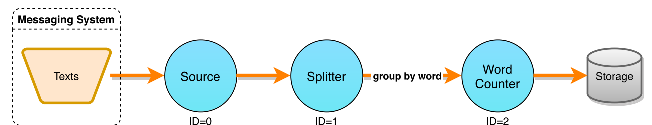


FIGURE 3. Word Count application (WC).

1) WORD COUNT (WC)

WC is a popular synthetic application splitting the sentences of a long text in words, and counting the number of occurrences of each word in the whole corpus. While the operator (Splitter) doing the splitting of words is stateless, words are sent to the instances of the operator doing the counting (WordCounter) in such a way that all the instances of the same word are delivered to the same instance of the counting operator. The graph (see Figure 3) is a pipeline with a group-by distribution in the middle.

2) MACHINE OUTLIER (MO)

MO receives resource usage readings from computers in a network, calculates the Euclidean distance of reading from the cluster center of a set of readings in a given time period, and applies the BFPRT algorithm to detect abnormal readings [73]. This application has been cleaned and integrated into our suite from an old GitHub repository.[3] The data-flow graph in Figure 4 shows a pipeline of operators with a group-by distribution in the middle.

3) LOG PROCESSING (LP)

The LP application[4] receives a stream of logs coming from HTTP web servers. They are in the Common Log Format and need to be parsed in order to extract the relevant data fields, such as the timestamp, request verb, resource name, IP

[3] https://github.com/yxjiang/stream-outlier.

[4] We adapted the code available (only for Apache Storm) in the GitHub repository https://github.com/domenicosolazzo/click-topology.

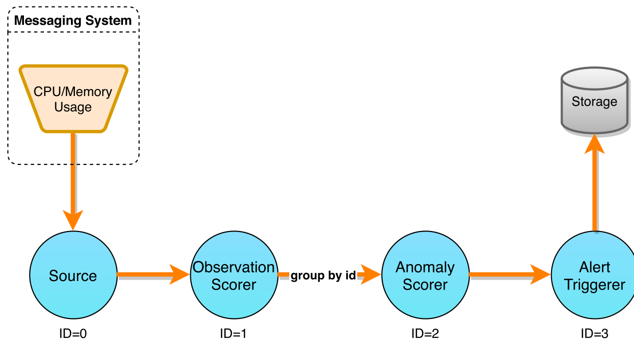


FIGURE 4. Machine Outlier application (MO).

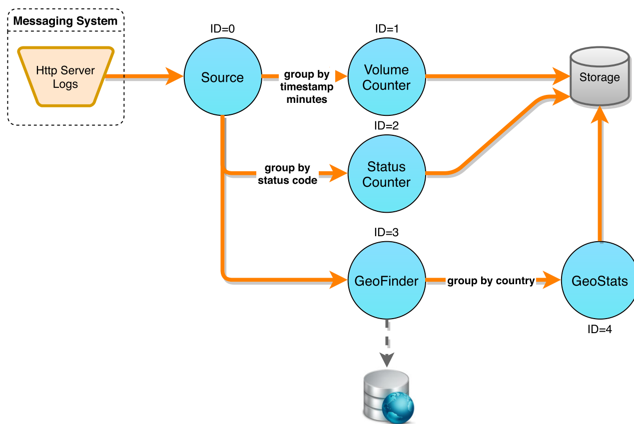


FIGURE 5. Log Processing application (LP).

address of the user, and the status code. The data-flow graph is shown in Figure 5.

The parsed stream is duplicated to three operators. The VolumeCounter operator counts the number of visits per minute, with each event representing a single visit. The StatusCounter operator stores the number of occurrences of each status code in an associative array. The GeoFinder operator finds the location of the user using its IP address and an IP location database (MaxMind or GeoIP), and emits a new event with the name of the country and city of the user if found. The subsequent operator, GeoStats, receives the location information and updates the counter per country and city, emitting the new values.

4) SENTIMENT ANALYSIS (SA)

The SA application[5] uses a simple NLP (Natural Language Processing) technique to calculate the sentiment of sentences, consisting of counting positive and negative words and using the difference to indicate the polarity of the sentence. The application is designed to receive a stream of tweets in JSON format, where each tweet has to be preliminary parsed in order to extract the relevant fields (identifier of the tweet, language, and text content). The graph is the pipeline in Figure 6.

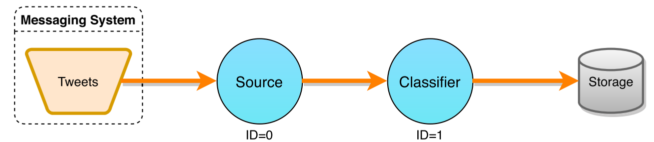


FIGURE 6. Sentiment Analysis application (SA).

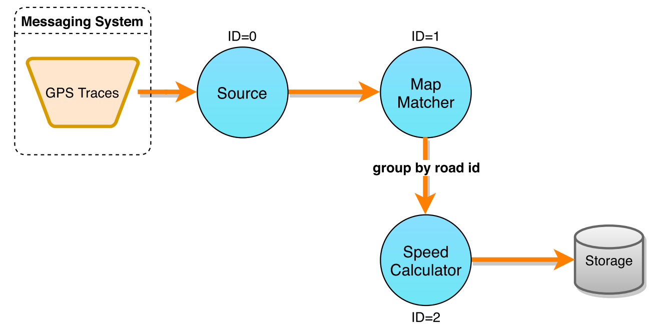


FIGURE 7. Traffic Monitoring application (TM).

After being parsed, the tweets are filtered to remove those that have been written in a language that is not supported by the application. By default, only English is supported, although our code has been designed in order to easily switch between languages by loading the right list of negative/positive words that must be available to support a new language. The Classifier operator is in charge of removing so-called *stop words*, i.e. words that usually do not carry sentiment and thus are irrelevant. Then, it counts the positive and negative words in the message. Finally, the *sentiment* of the tweet is produced outside: it is positive if the number of occurrences of positive words is greater than the negative ones, or negative otherwise.

5) TRAFFIC MONITORING (TM)

The TM application[6] is a chain of operators as in Figure 7. It receives events emitted from vehicles containing their identifier, location (latitude and longitude from a GPS), direction, current speed, and timestamp. The MapMatcher operator receives these events and identifies the road that vehicles are riding. To do so, this operator is initialized with a bounding box that corresponds to the borders of the city being monitored, enabling the component to eliminate the events that occurred outside the city limits. It also loads a shapefile with all the roads of the city, which is used to lookup the road that vehicles are riding based on their current location. Geospatial operations are done using the GeoTools library [74].

After finding the road, the MapMatcher operator appends the road ID to the event and forwards it to the SpeedCalculator operator, which calculates the average speed of the vehicles for each road creating a new event with the timestamp, road ID, average speed and the number of vehicles on the road.

[5] Our code has been inspired by the code in the repository `https://github.com/voltas/real-time-sentiment-analytic`.

[6] Our TM code extends the source code available for Storm at `https://github.com/whughchen/RealTimeTraffic`.

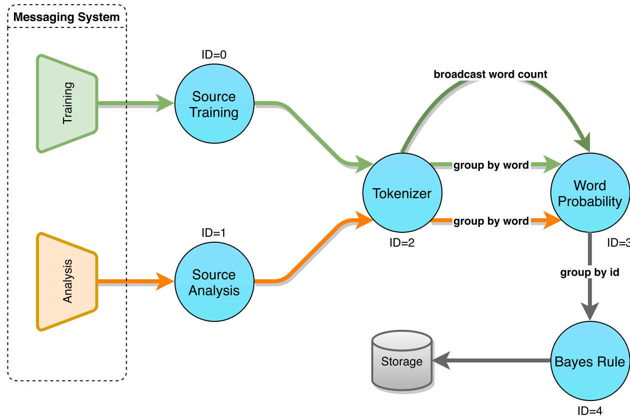


FIGURE 8. Spam Filter application (SF).

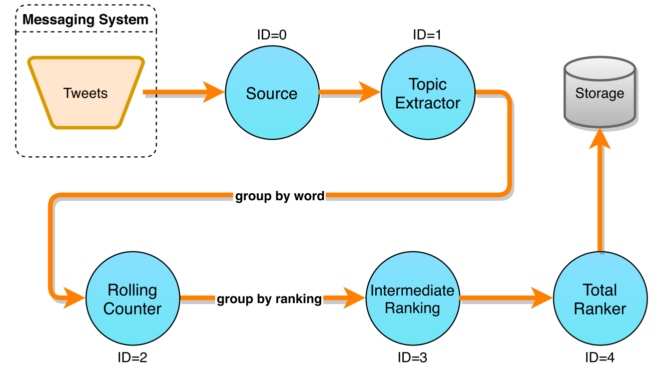


FIGURE 9. Trending Topic application (TT).

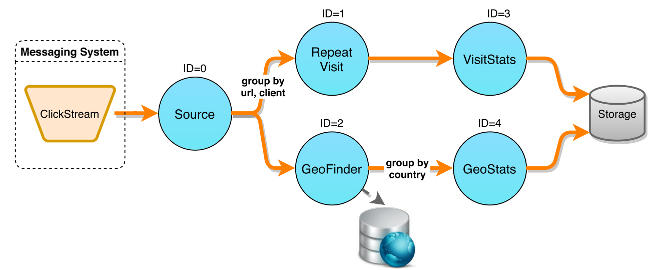


FIGURE 10. Click Analytics application (CA).

6) SPAM FILTER (SF)

The SF application uses Naive Bayes [75] to analyze if email messages are spam (or ham). As opposed to other applications that require an offline training phase, in this case, there is a training stream that enables the application to be trained in real-time. The data-flow graph (see Figure 8) is composed of several operators. This application processes two logically separated streams (for online training and for the analysis), which is a feature not used by previous applications.

The application also supports offline training, which means that the probabilities of words are pre-loaded by the Word-Probability operator. In this case, the events from the Tokenizer operator do not need to be grouped by word since all the instances of WordProbability have a static read-only table with all the probabilities of the words. The advantage of offline training is that the recovery of an operator after a failure is very quick since it only needs to load the probability file again instead of having the be trained from scratch. The downside is that instances of this operator consume memory to load the large probability file.

7) TRENDING TOPICS (TT)

The goal of TT is to extract topics from a stream of *tweets*, count the occurrences for each topic in a window of events (limited size), and emits only the popular topics (i.e. the trending topics).

The occurrences of topics are tracked by a sliding-window operator (RollingCounter), which advances by a fixed interval of time. The IntermediateRanking operator is used to rank a subset of topics, and in a fixed interval of time, these intermediate scores are sent to the TotalRanker operator, which merges the intermediate scores and emits the final ranking scores of topics. The graph is a pipeline of operators shown in Figure 9.

An example of such an application is the TwitterMonitor [76], a system that detects trends in real-time from Twitter. A similar application has also been used to compare the performance of a traffic monitoring and analysis tool called BlockMon [67].

8) CLICK ANALYTICS (CA)

CA receives events from users accessing a website. These inputs are logs from the web server, usually in the Common Log Format, which means they have to be parsed in order to extract the relevant data fields. The most common fields are the timestamp, URL, IP address of the user, identifier of the user (the IP address is used if the ID is not available). The graph is depicted in Figure 10.

After the Source operator, the stream is broadcast to two destination streams using a different distribution strategy (shuffle and group-by). In the RepeatVisit operator, events are grouped based on the URL and ID of the user. These two fields are used as a key in an associative array to verify if the user has already visited the URL or not. The downstream operator VisitStats counts the total number of visits and unique visits.

On the other stream, the events are randomly distributed among the instances of the Geography operator. During the initialization, it creates a connection to a remote database (both MaxMind or GeoIP can be used). Upon receiving an event, the operator queries the database with the user IP address and receives as a result the location of the user. The operator extracts from the location the name of the city and the country and forwards them as a new event to the GeoStats operator. This latter stores some information fields for each country in an associative array. One field is a counter of visits. The other is an associative array with counters per city within the country. Upon the arrival of events, this operator updates the content of its associative array and emits the new values.

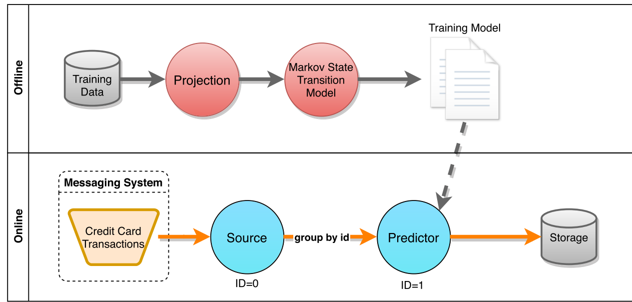


FIGURE 11. Fraud Detection application (FD).

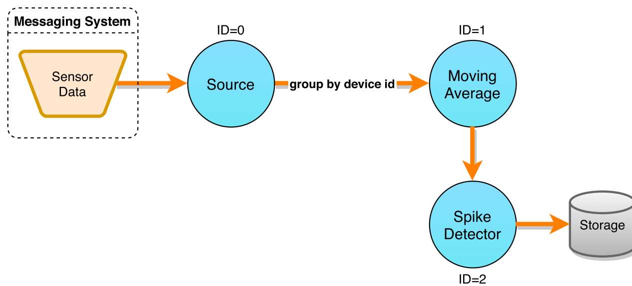


FIGURE 12. Spike Detection application (SD).

9) FRAUD DETECTION (FD)

FD uses a Markov model [77], created during an offline phase, to calculate the probability of a credit card transaction is a fraud. The source operator is in charge of cleaning the raw input stream of credit card transactions that are delivered to the Predictor operator grouped by identifier. The predictor uses the model to emit transactions that are considered a fraud with a minimum threshold probability. The graph is shown in Figure 11.

10) SPIKE DETECTION (SD)

The SD application (see Figure 12) receives a stream of readings from sensors in order to monitor spikes. The MovingAverage operator receives these events grouped by the sensor identifier and maintains for each identifier a moving window. When a new event is received, the operator adds the new value to the corresponding window and emits a new event with the identifier of the device, the current value, and the moving average. The operator SpikeDetector receives these events and based on a threshold value specified at initialization, checks whether the current event is a spike or not, by computing the relative difference between the current value and the average of the last window and emitting those values that exceed the threshold.

11) BARGAIN INDEX (BI)

BI is used in past research papers [25], [26], [78] to evaluate IBM InfoSphere Streams [79] (we translated the code to be included in our suite, making it available to be run in any DSPS). The application analyzes stocks available for selling in quantity and price below the mean observed in the last time window. To this end, the application computes a so-called *bargain index*, a scalar value representing the magnitude of the bargain. The data-flow graph is shown in Figure 13. The system is fed by a stream of trades and quotes from a financial market. A trade is a completed transaction while a quote is a selling/buying proposal (called the bid/ask in the financial jargon). The Source operator receives both types of inputs and splits them in two separate streams analyzed in parallel.

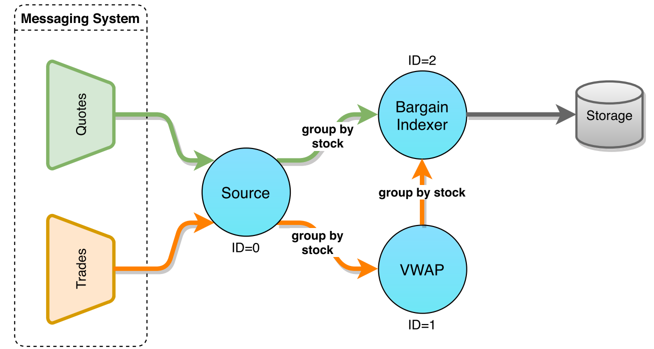


FIGURE 13. Bargain Index application (BI).

The VWAP operator computes the volume-weighted average price of trades for the same stock symbol over a window of the last 15 trades received. The BargainIndexer operator receives quotes and calculates the bargain index (which requires the comparison between the price and volume of the quote with the last VWAP computed). Only quotes with a bargain index greater than a threshold are emitted to the sink. Further information about the definition of the bargain index, and of past uses of this application, can be found in [25], [26], [78].

12) REINFORCEMENT LEARNER (RL)

We developed this application based on an existing repository of predictive and exploratory machine learning tools.[7] The application has only one operator doing the processing (ReinforcementLearner). It uses the interval estimate algorithm [80], and to choose the action to take, it uses second-order statistics of the reward distribution. The graph is shown in Figure 14.

The input stream conveys PRLs (Page Request Logs) consisting of a session identifier, and a counter incremented at each new request. The ReinforcementLearner operator decides which page to display from a pre-configured list and sends a new event with the session and page identifiers. The source also receives CTRs (Click Through Rates) for each one of the pre-configured pages, which can be translated as the performance of each page displayed to the users. The CTR data are also fed into the ReinforcementLearner for building a reward histogram of each page. This is used for the learning algorithm to improve its decision making.

The algorithm is characterized by two phases: exploration and exploitation. In the beginning, the reward distribution

[7] https://github.com/pranab/avenir

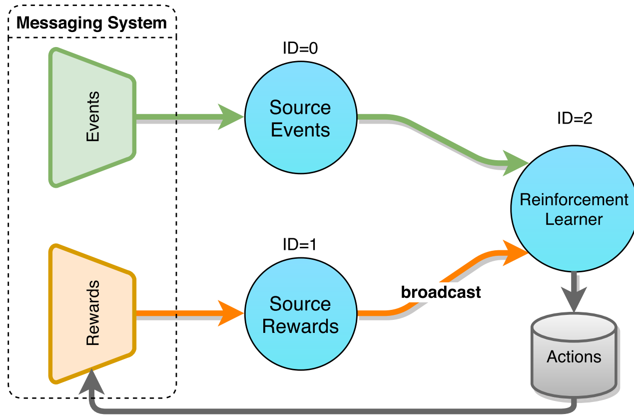


FIGURE 14. Reinforcement Learner application (RL).

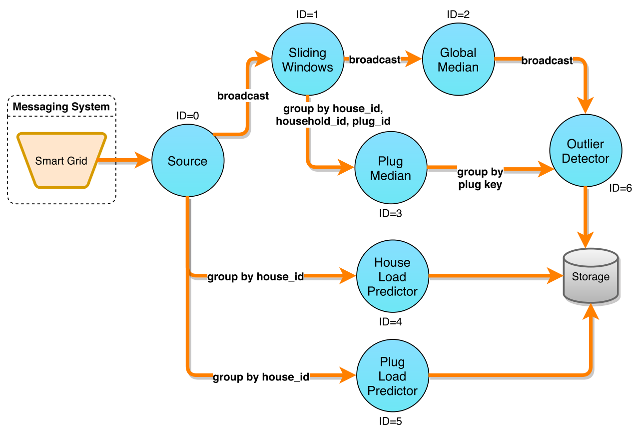


FIGURE 15. Smart Grid Monitoring application (SM).

does not have enough data, so the algorithm chooses actions randomly. When enough data are available, the exploitation phase begins, with the algorithm choosing actions with the highest mean reward.

13) SMART GRID MONITORING (SM)

This application was proposed for the DEBS 2014 Grand Challenge.[8] It monitors the energy consumption of a smart electricity grid in order to allow load predictions and detection of outliers. It is based on a graph of operators as depicted in Figure 15.

The application produces two results: outliers per house and house/plug load predictions. The outlier detection is done by first calculating the global median of all houses and then comparing it with the median of each house plug (values above the global median are considered outliers). The prediction uses the current average and median to predict future loads.

14) TELECOM SPAM DETECTION (VS)

The application (called VoipStream or simply VS) detects telemarketing users by analyzing Call Detail Records (CDRs)

[8] http://www.cse.iitb.ac.in/debs2014/?page_id=42

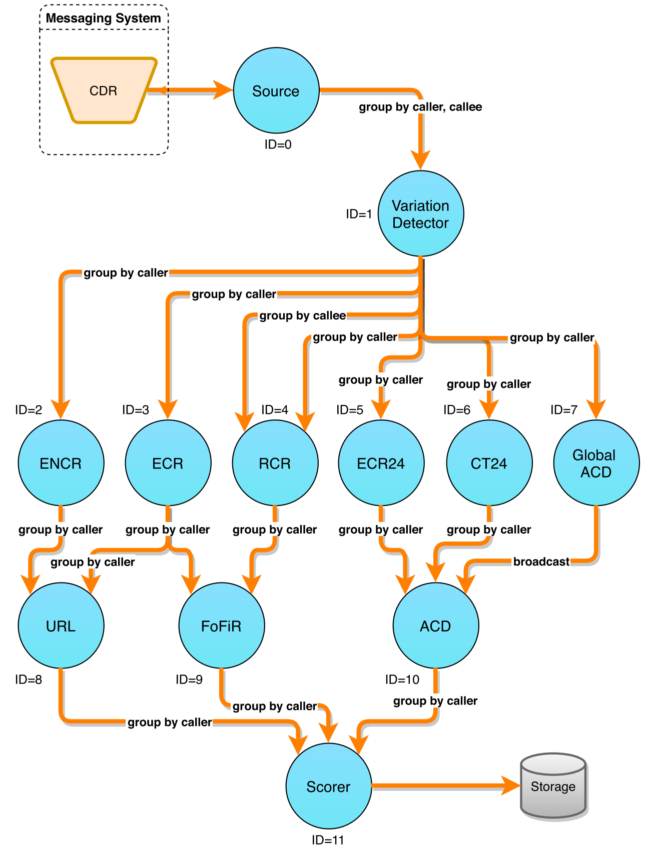


FIGURE 16. Voip Stream application (VS).

using a set of filters based on time-decaying bloom filters [81]. A similar application is used in the evaluation of the BlockMon system [67]. The application consists of a complex graph of 12 operators depicted in Figure 16.

Apart from the source, the other operators in VS belong to two categories: filters and scorers. Filters (CT24, ECR, ECR24, ENCR, and RCR) implement on-demand time-decaying bloom filters to keep track of the number of unique incoming participants in a way that the actual rate decays according to the insertion time. Scorers receive rates from filters and emit a properly weighted value once they have received at least one rate from each input. The application makes intensive use of group-by distributions (using two fields, the caller, and the callee identifiers).

15) ADS ANALYTICS (AD)

AD is an application based on previous research papers [82] to calculate the Click Through Rate (CTR) of advertisements (ads). It receives two streams of events: impressions of ads and clicks of ads. With those two streams, the CTR is calculated for the combination of the user query and advertisement identifier. The total sum of impressions and clicks per query/ads pair is stored in memory and updated based on a time window. The CTR is emitted at fixed intervals of time for each query/ads pair. The graph (see Figure 17) is a graph

TABLE 4. Areas and datasets of each application in DSPBench.

Acronym	Application Name	Area	Dataset
WC	Word Count	Text Processing	Project Gutenberg (∼8GB)
MO	Machine Outlier	Network Monitoring	Google Cluster Traces (36GB)
LP	Log Processing	Web Analytics	1998 WorldCup (104GB)
SA	Sentiment Analysis	Social Network	Twitter (<collected>)
TM	Traffic Monitoring	Sensor Network	Beijing Taxi Traces (∼300MB)
SF	Spam Filter	Telecommunications	Enron Email Dataset (2.6GB)
TT	Trending Topics	Social Network	Twitter (<collected>)
CA	Click Analytics	Web Analytics	1998 WorldCup (104GB)
FD	Fraud Detection	Finance	<generated>
SD	Spike Detection	Sensor Network	Intel Berkeley Research Lab (150MB)
BI	Bargain Index	Finance	Kaggle Stock Market Dataset of NASDAQ (3GB)
RL	Reinforcement Learning	Advertising	<generated>
SM	Smart Grid Monitoring	Sensor Network	DEBS 2014 Grand Challenge (3.2GB)
VS	VoipStream	Telecommunications	<generated>
AD	Ads Analytics	Advertising	KDD Cup 2012 (12GB)

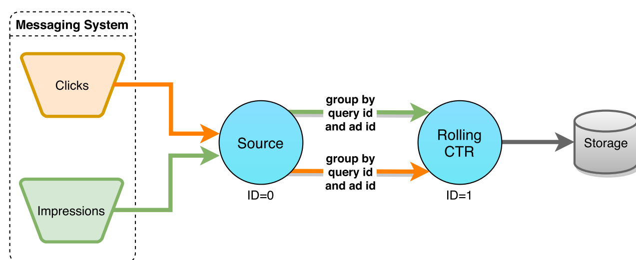


FIGURE 17. Ads Analytics application (AA).

of four operators: two source operators sending data with a group-by distribution.

V. WORKLOAD CHARACTERIZATION

A summary of the chosen applications, the selected areas, and the origin of the datasets used to reproduce a realistic input stream for each of them is provided in Table 4. Most of the datasets consist of data from real-world scenarios. In the case of FD, RL, and VS, a dataset has not been found, instead, a random generator has been used and included in the source code of DSPBench.

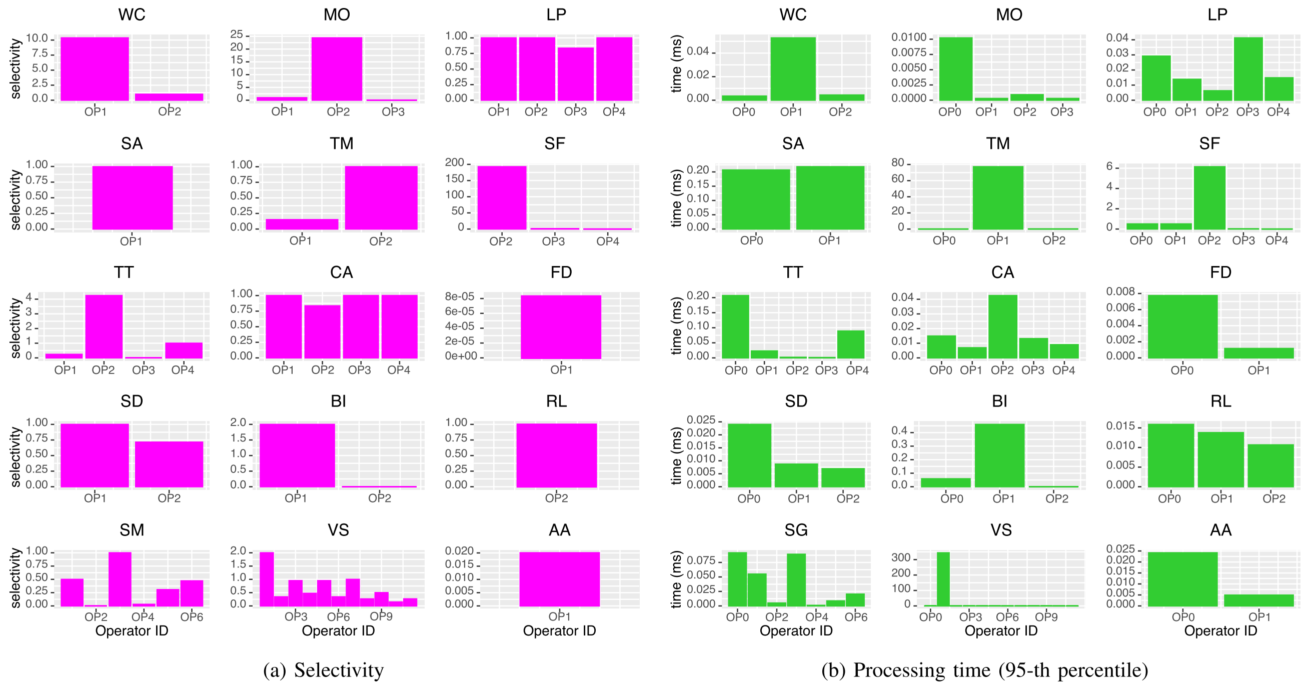
There are also cases where the size of the dataset is not big enough, i.e. the dataset can be consumed in a few minutes, requiring it to be replicated until its size becomes acceptable. It is important to note that replicating a dataset is not always as simple as making copies of it. There are cases where some data fields have to be changed in order to not break the application semantics. As an example, datasets that have timestamp fields must be altered in order to follow a continuous timeline. This is done automatically by the Input component of the DSBBench architecture (see Figure 1).

To characterize the selected applications, we conducted specific experiments on a single machine in order to measure the selectivity of operators, the time required to process one tuple per operator, the size of the tuples at each operator, and the memory usage of the applications. They are all characteristics requiring instrumentation as shown in Table 3. The machine is an Intel Core i7 5500U with a clock rate of 2.4 GHz, 8 GB of RAM, 1 TB Hard Drive + 8 GB SSD, and Ubuntu 16.04 LTS 64bit. For what regards the remaining two static characteristics (operator type and communication pattern) requiring source code analysis, they have already been provided in the previous section with the description of the applications and of their data-flow graphs.

The results of the selectivity analysis are shown in Figure 18a for the operators of the 15 applications (we excluded sources because they do not receive inputs and sinks because they do not emit outputs). As already stated, the selectivity is the ratio between the total number of tuples received and emitted by the same operator. The greater the selectivity, the greater is the number of tuples emitted per input. However, this does not necessarily mean that the overhead increases, as some DSPSs group tuples and send them in batches to optimize communications. From the analysis, the applications in the suite with higher selectivity are WC (with the Splitter operator), TT (RollingCounter), and SF (Tokenizer). This last is by far the one with the highest selectivity. For the other applications, selectivity is in most cases smaller than 1.0 (filtering) or exactly 1.0 (for map and projection operators).

The second observed characteristic is the time spent by an operator to process a single input tuple. The results are shown in Figure 18b correspond to the 95-th percentiles obtained from samples retrieved every 5 seconds (several hundreds of samples are retrieved for each operator during the long-running of the applications). Here we also include the source operators in the analysis.

There are applications with homogeneous processing times across operators, while others have bigger differences. The highest processing time is for the Tokenizer operator (OP2) in SF, which can be explained by its high selectivity, see Figure 18a. The second computationally-demanding application is TM, where the slowest operator is MapMatcher (OP1). This happens although this operator actually filters some tuples (it checks the bounding box of the city by discarding inputs not falling in the monitored area). However, although with a selectivity less than 1.0, its computational cost is high due to the calls of the GeoTools functions, which revealed



(a) Selectivity

(b) Processing time (95-th percentile)

FIGURE 18. **Selectivity and processing time per input (95-th percentile) of the operators in DSPBench.**

expensive. For WC the slowest operator is the one splitting sentences into words (OP1). MO is a fine-grained computation where the slowest operator is the source producing the stream inputs. This also happens for TT. For VS, although it has a complex graph, one operator (ValidationDetector, OP1) is the slowest one, because it broadcasts each input tuple to six distinct operators as shown in Figure 16.

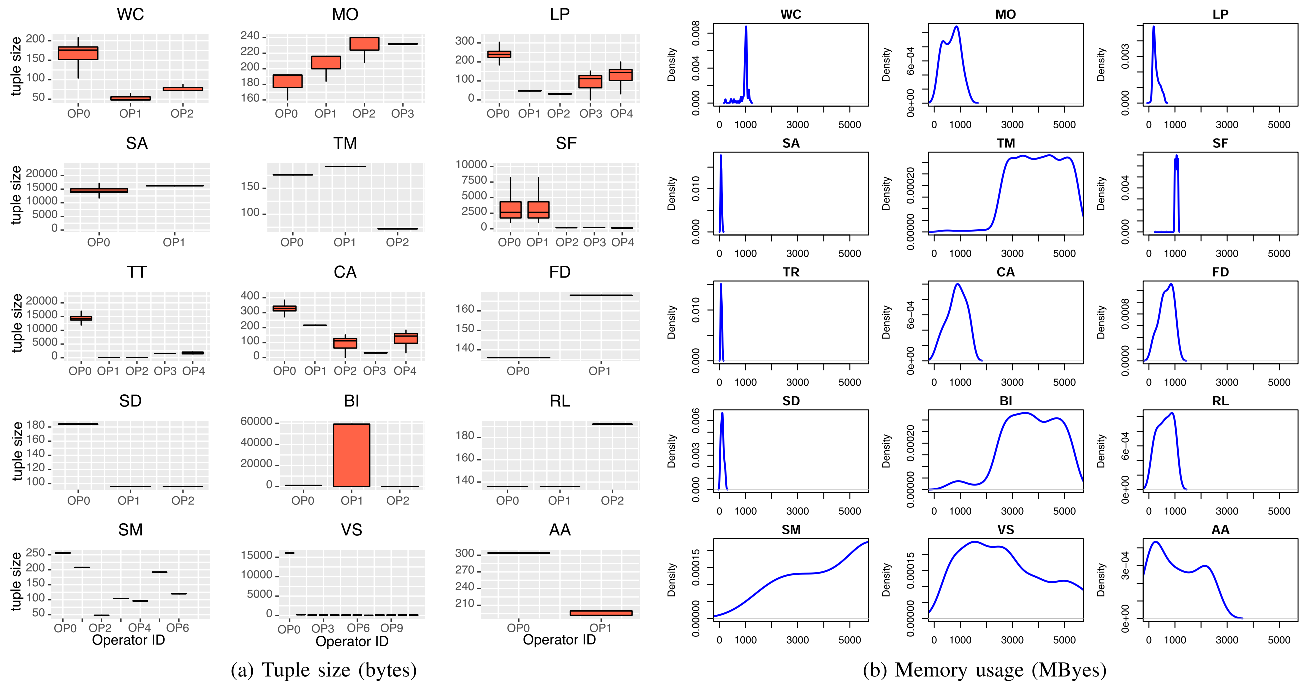
Figure 19a shows the size of the tuples handled by the operators of the applications in our suite. In some cases, the size is variable during the execution, while in others it is fixed to a specific value. In both cases, we report the sizes using boxplots. The usual behavior of the tuple size is to decrease along the path from the sources to downstream the data-flow graph, which is why the source has usually the greater tuple size. Some exceptions are MO, FD, SA, SM, and RL, where the tuple size increases along the path. This happens because the original tuple is carried along with the steps in the graph, while more information fields are aggregated with the tuple at each step. It is worth noting that the tuple size and the selectivity are often not correlated. There are cases, e.g., SF with the Tokenizer operator (OP2), where although the selectivity is high, the tuple size is small as well as its variation.

It is important to have applications with different patterns of memory usage in the benchmark suite. Figure 19b shows the distribution of the memory usage, measured at fixed time intervals during the execution (every second). The chart is a kernel density plot showing the density corresponding to continuous values of memory occupation. The selected applications exhibit five memory behaviors: fixed and variable memory usage; and low, medium, and high memory usages. In Figure 19b, applications with a wide base in the chart have a high variation of memory usage (like AA, BI, VS, and TM), while those with a narrow base have a less variable memory consumption (like WC, SD, SA, LP, and SF). As a further observation, we point out that bigger tuple sizes do not necessarily mean high and variable memory usage (like in VS).

VI. EXPERIMENTS

The aim of this final section is to exemplify the use of DSPBench to evaluate different DSPSs. We do this by selecting a small representative subset of the applications in our suite, and we consider two different DSPSs: Apache Storm and Spark Streaming. These two systems are worth being compared because they adopt different processing models (one tuple-at-a-time versus micro-batching). Although a full comparison between DSPSs is not our goal, we propose this analysis to show the potential of our suite to stress different aspects of the DSPS architecture.

The chosen subset of applications is composed by Log-Processing (LP), TrafficMonitoring (TM), and WordCount (WC). LP is an application example with a complex acyclic data-flow graph (see Figure 5), involving several kinds of distributions (shuffle and group-by) as well as a broadcast out of the Source operator. TM has been chosen for its large memory footprint and because it is very computationally demanding (the MapMatcher operator is one with the highest processing time per input). Finally, WC has been selected because it exhibits a medium level of selectivity (see Figure 18a), which is instead very low in the other chosen applications.



(a) Tuple size (bytes)

(b) Memory usage (MByes)

FIGURE 19. Tuple size and memory usage of the 15 applications in DSPBench.

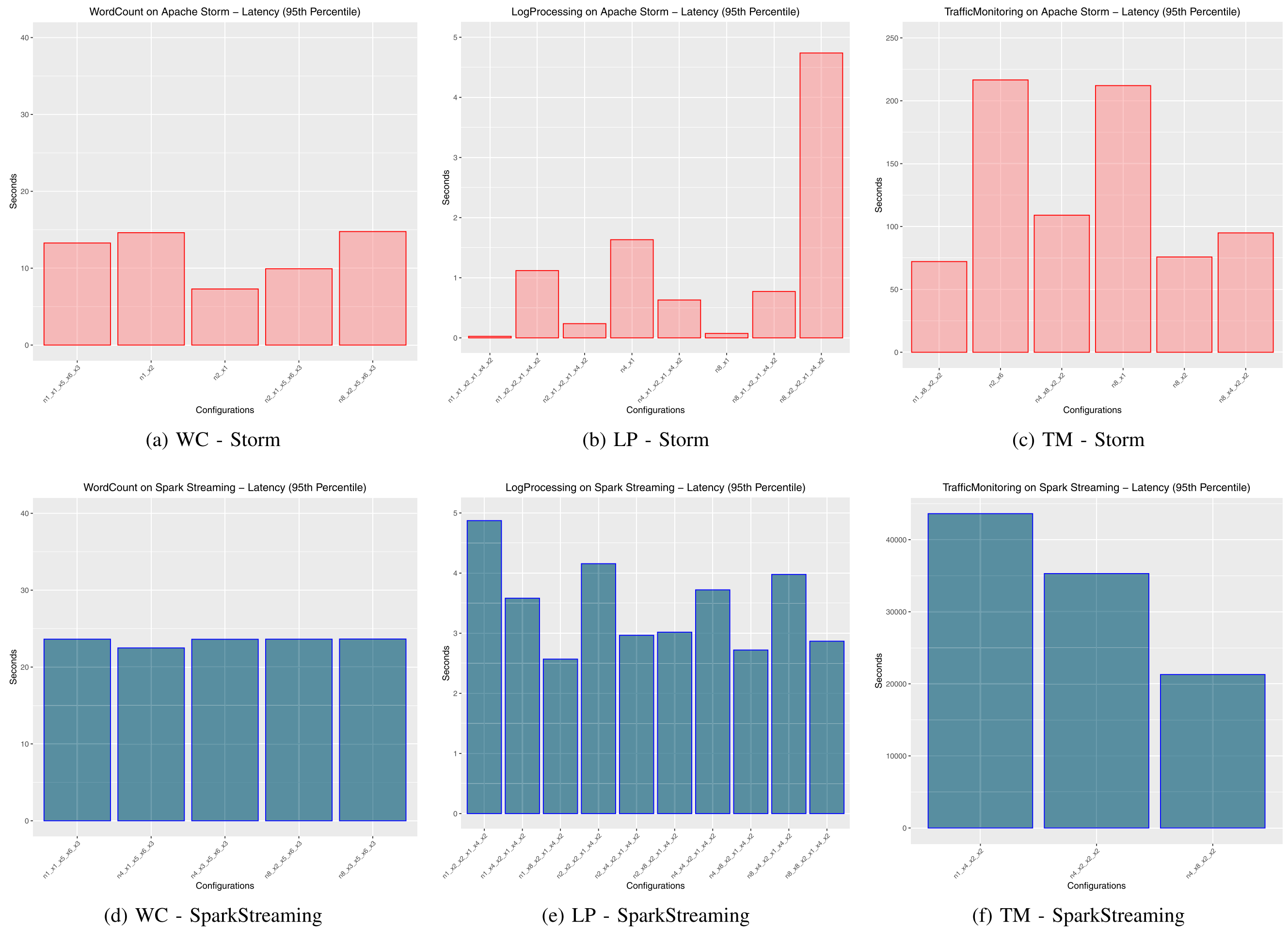
A. PLATFORM SETUP AND METRICS

The experiments were executed in the Azure Cloud computing service with a cluster of eight computing instances, one master instance, and three data instances, all of the type Medium (Standard_A2) running Ubuntu. Each instance has two CPU cores, 3.5 GB of memory, a local HDD of 135 GB with a maximum I/O disk throughput estimated in 500 IOPS. The message system adopted was Apache Kafka installed on the 3 data instances. The data producers that fed Kafka were also installed on the data instances, thus avoiding network traffic. Each data instance had one data producer reading data from a separate HDD and forwarding it to Kafka. On the application side, the number of Kafka partitions always matched the number of instances of the source operator in each application.

Each experiment is executed three times until the whole dataset for the given application has been emitted to the DSPS. All metrics are calculated after the end of the execution, by querying the storage system that recorded the stream outputs. Some measurements, such as the throughput of a single operator instance, are logged locally and then flushed to the storage system. We configure DSBBench with some probers (see Sect. III) to collect the following measurements and derive corresponding metrics:

- **throughput**: it is collected at the level of single operators by using a counter and a variable to record the current timestamp. The *instant throughput* is measured as the ratio between the number of outputs produced and the current running time. Instant throughput samples are then aggregated to produce the *average throughput* per operator;
- **latency**: every tuple has a creation timestamp recorded when it is generated by the source outside the DSPS. All the tuples flowing in the graph inherit the creation timestamp of the upstream tuples according to [83]. The *instant latency* is measured as the difference between the current time at the sink and the creation timestamp contained in the received tuple. We usually use the 95-th percentiles of the instant latency, and the Network Time Protocol (NTP) to ensure a coherent global time in the cluster;
- **resource consumption**: this prober uses Ganglia [84] to monitor the nodes in the clusters. The goal is to collect consumption metrics related to CPU, memory and network utilization, in order to study the behavior of the DSPS by changing the application configuration (e.g., number of used nodes and the parallelism of each operator).

A common issue in configuring Storm and Spark Streaming applications is to choose the right parallelism level of each operator/transformation. Such parameters must be carefully chosen by the user to get satisfactory performance with reasonable resource consumption. The approach chosen in this article is to calculate the weighted average of the processing time required by one tuple for each operator in relation to the overall processing time of one tuple in the whole application. To experiment with more configurations, the initial assignment of parallelism to each operator is used as a starting point, and, based on a set of multipliers, we experimented with other configurations with higher parallelism within the cluster limit.



(a) WC - Storm (b) LP - Storm (c) TM - Storm

(d) WC - SparkStreaming (e) LP - SparkStreaming (f) TM - SparkStreaming

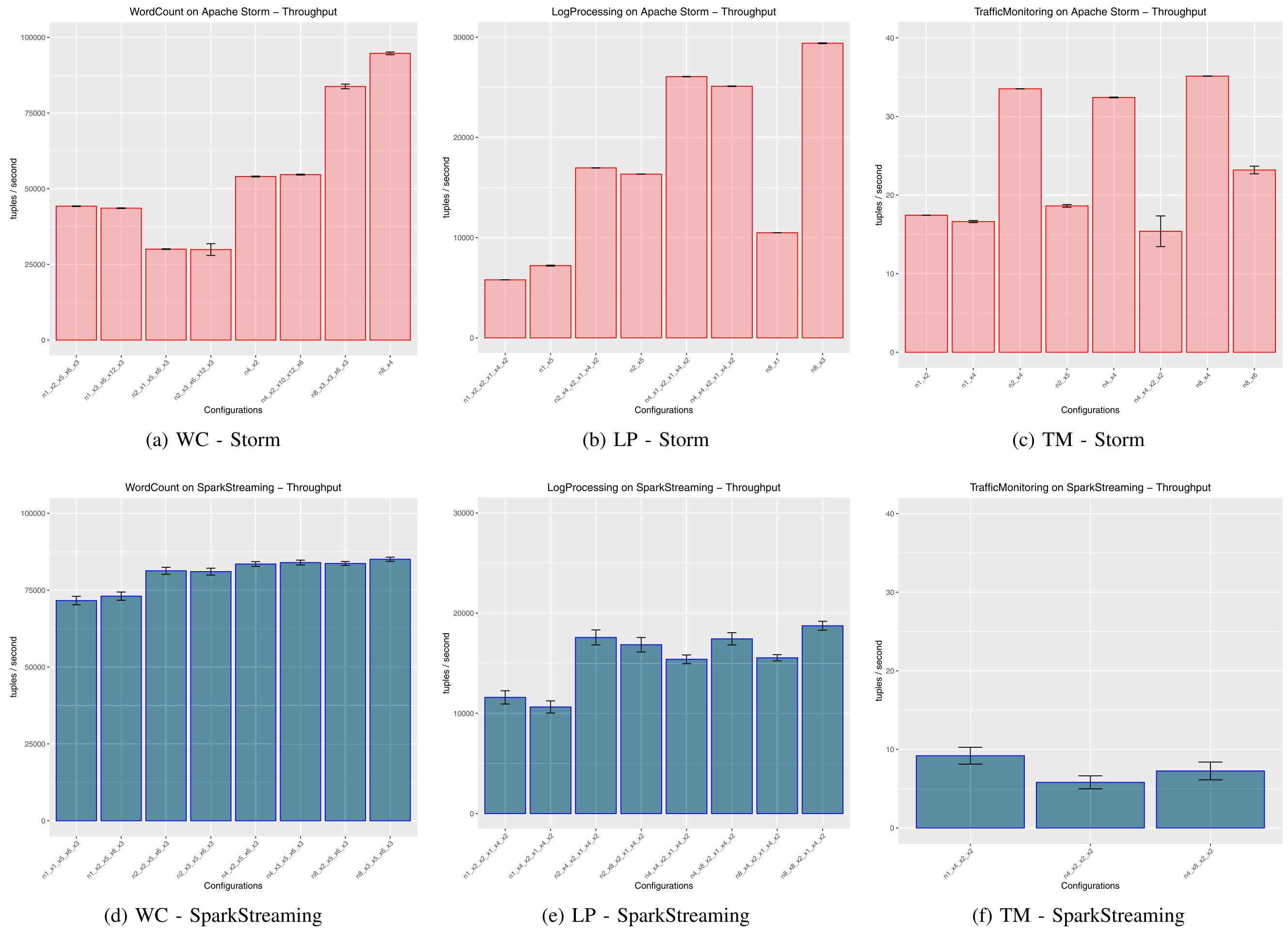
FIGURE 20. **Latency (95-th percentile) of WordCount (WC), LogProcessing (LP) and TrafficMonitoring (TM) with Apache Storm and Spark Streaming on the Azure cluster.**

B. LATENCY AND THROUGHPUT ANALYSIS

We report the 95-th percentile of the latency measured on the Azure cluster. The results are shown in Figure 20. We report the 95-th percentile because the mean value hides the presence of outliers while the maximum value is ill-affected by single outliers. In each plot, we report the best configurations found. Each configuration is identified with a label in the form (for WC for example): *nNodes_xSources_xSplitter_xCounters_xSinks*, i.e. we report the number of used nodes in the cluster, and the parallelism of each operator (for WC four operators, the source, splitter, counter and sink). If the operators have the same parallelism M, we use the notation *nNodes_xM*.

The latency of WC in Storm and Spark Streaming is reported in Figure 20a and 20d. For WC, we tried 52 different configurations using 1, 2, 4, and 8 nodes. For each number of nodes, we report only the best configurations found for the sake of brevity. As a general consideration, Storm provides better latency, under 15 seconds in most cases. Spark Streaming provides instead more stable latency values, stable in the best configurations but higher than Storm. The batch size of Spark Streaming has been set to one second, which was an acceptable compromise to optimize throughput without hampering latency too much. Qualitatively, the same outcome has been achieved with the other two applications: LP and TM. TM is the most computationally demanding application in our suite, and this reflects in high latency values measured in both the DSPSs (with Storm still better), while LP exhibits a smaller latency, due to its more fine-grained nature.

The average throughput is shown in Figure 21. Also in this case we report the best configurations, while error bars (when visible) report the standard error. For WC and TM the average throughput is measured in the Sink (number of outputs received), while for LP we aggregate the average throughput of the last three operators present in the dataflow graph (see Figure 5): VolumeCounter, StatusCounter, and GeoStats. The result is that throughput slowly increases with more nodes (and a proper choice of the parallelism per operator) except in TM with Spark Streaming, where using more nodes does not bring any improvement for this specific application. In general, stream processing applications do not exhibit perfect scalability, due to their very erratic nature



(a) WC - Storm (b) LP - Storm (c) TM - Storm

(d) WC - SparkStreaming (e) LP - SparkStreaming (f) TM - SparkStreaming

FIGURE 21. Throughput (error bars report the standard error) of WordCount (WC), LogProcessing (LP) and TrafficMonitoring (TM) with Apache Storm and Spark Streaming on the Azure cluster.

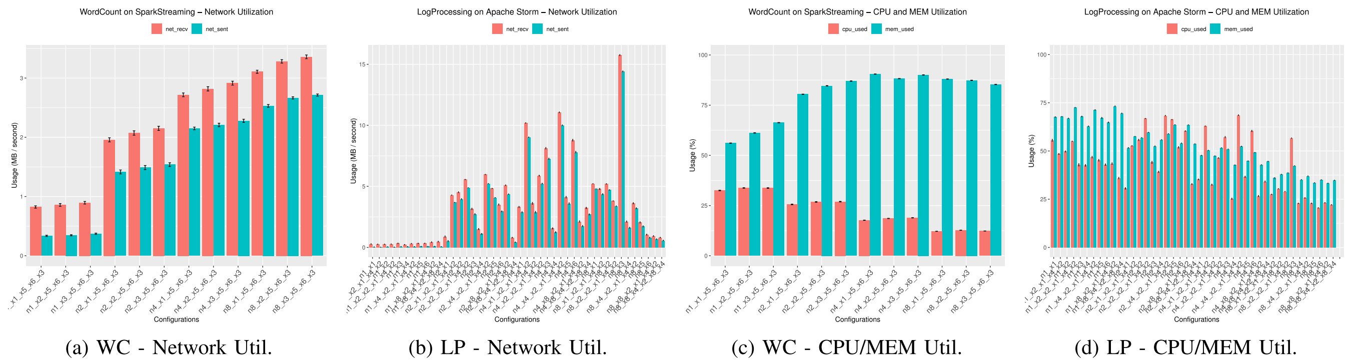
(e.g., in most cases the throughput improvement is limited by a skewed distribution of the key attributes used to group input tuples). However, the general outcome is that Storm outperforms Spark Streaming both in latency and throughput for the three considered applications (although Spark Streaming, with its micro-batched architecture, provides stabler results).

C. RESOURCE CONSUMPTION ANALYSIS

One of the features of DSPBench is to provide easy-to-plug probers for different performance/consumption metrics. In this part, we report the results in terms of resource utilization, where we are interested in CPU, memory, and network utilization. The results are in Figure 22.

We report the most interesting results. Network utilization is shown for two applications: WC and LP (TM, which is computationally demanding and with low throughput, shows very low network utilization). For both Spark Streaming (with WC, see Figure 22a) and Storm (with LP, see Figure 22b), the network utilization in MB/second slightly increases with configurations using more nodes in our cluster, while it is minimal with few used nodes. This trend is less evident in LP with Storm because the application has a low selectivity (see Figure 18a) compared with WC, and fewer tuples are emitted per time unit by its operators. As expected, configurations providing better throughput are also the ones exhibiting the greater network activity, since more tuples are pushed downstream.

In terms of CPU and memory consumption, Figures 22c and 22d show the results for the same scenarios. As a general trend, CPU utilization decreases with configurations using more nodes in the cluster. By looking at the throughput results, there is a clear correlation between throughput and CPU utilization: configurations using more nodes which, however, do not provide higher throughput are the ones with lower CPU utilization on the cluster nodes because more resources are not used effectively. In terms of memory consumption, the two DSPSs look different in dealing with the two applications. While WC exhibits a constant memory consumption with more nodes, LP exhibits instead a decreasing trend, although this happens slowly. This is in line with the workload characterization done in the previous section, where LP exhibited a lower overall memory occupation, more sensible to variations based on how the internal state of the



(a) WC - Network Util. (b) LP - Network Util. (c) WC - CPU/MEM Util. (d) LP - CPU/MEM Util.

FIGURE 22. **Resource (network, cpu and memory) utilization of WordCount (WC) with Spark Streaming and LogProcessing (LP) with Apache Storm on the Azure cluster. Error bars report the standard error.**

operators is maintained and split in case of higher operator parallelism.

D. FINAL DISCUSSION

The results show that Storm performs better than Spark Streaming for the three chosen applications. It generally provides higher throughput and lower latency. In terms of resource utilization, experiments that performed better on throughput also had the highest CPU, memory, and network utilization among the configurations with the same number of nodes.

One of the takeaways of this analysis is the importance of a careful configuration of the application in terms of the number of nodes and the choice of the parallelism per operator. DSPSs traditionally adopt basic strategies for choosing how to map operators onto nodes, while the research is dense of advanced strategies for operator placement that are not however included in the default run-time system of those DSPSs. The choice of the degree of parallelism for each operator is also a decisive point in achieving good performance. However, its selection is essentially left to the programmer in all the existing traditional DSPSs.

The presented experiments have the purpose of showing that DSPBench is a benchmark suite able to assess the differences among the existing DSPSs on a rich set of applications. This section focuses on three of them for the sake of space, while for all the applications in our suite a complete workload characterization has been given.

VII. CONCLUSION

Data Stream Processing is a computing paradigm useful in an emerging set of applications that process live streams of data. Several existing open-source systems (DSPSs) have been released to develop such kind of applications in distributed systems like clusters. Although DSPSs have been available for years, few benchmark suites of real-world applications have been developed and made publicly available to the research community. This article presented DSPBench which, as far as we know, is the most complete suite of real-world applications present in the literature. It is composed of 15 applications. Some of them have been taken from the literature and adapted for the specific purpose of comparing systems, while others have been designed from scratch. This article provides a full workload characterization of all the applications in terms of memory utilization, tuple size, cost per tuple, and selectivity of their operators.

In the final part of this article, we have shown results that can be obtained by using our suite for comparing DSPSs. We chose three applications as a representative subset and two DSPSs of wide popularity. Results are provided in terms of latency, throughput and resource utilization (CPU, memory, and network), showing the effectiveness of our benchmark suite (based on probers) to get metrics. Our work has a high degree of extendibility, with other applications that can be included in the future by respecting our API. Furthermore, our work can be the basis of several new kinds of research that can be developed in the future to evaluate new optimizations for DSPSs evaluated on our rich and publicly available set of applications that we made available to the community.

REFERENCES

[1] H. C. M. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*, 1st ed. Cambridge, U.K.: Cambridge Univ. Press, 2014.
[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A new model and architecture for data stream management," *VLDB J. Int. J. Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, Aug. 2003.
[3] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Ç. Etintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the borealis stream processing engine," in *Proc. CIDR*, vol. 5, 2005, pp. 277–289.
[4] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom, "Stream: The stanford stream data manager (demonstration description)," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, New York, NY, USA, 2003, p. 665, doi: 10.1145/872757.872854.
[5] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *Proc. 11th Int. Conf. Compiler Construct. (CC)*. Berlin, Germany: Springer-Verlag, 2002, pp. 179–196.
[6] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: Semantic foundations and query execution," *VLDB J.*, vol. 15, no. 2, pp. 121–142, Jun. 2006, doi: 10.1007/s00778-004-0147-z.
[7] (2020). *Apache Storm*. [Online]. Available: https://storm.apache.org/index.html
[8] (2020). *Apache Flink*. [Online]. Available: https://flink.apache.org/

[9] (2020). *Spark Streaming*. [Online]. Available: https://spark.apache.org/streaming/

[10] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: A stream data management benchmark," in *Proc. 13th Int. Conf. Very Large Data Bases (VLDB)*, vol. 30, 2004, pp. 480–491.

[11] A. Shukla, S. Chaturvedi, and Y. Simmhan, "RIoTBench: An IoT benchmark for distributed stream processing systems," *Concurrency Comput. Pract. Exper.*, vol. 29, no. 21, p. e4257, Nov. 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4257

[12] S. Zhang, B. He, D. Dahlmeier, A. C. Zhou, and T. Heinze, "Revisiting the design of data stream processing systems on multi-core processors," in *Proc. IEEE 33rd Int. Conf. Data Eng. (ICDE)*, Apr. 2017, pp. 659–670.

[13] S. Zhang, J. He, C. Zhou, and B. He, "BriskStream: Scaling stream processing on multicore architectures," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2019, doi: 10.1145/3299869.3300067.

[14] M. Bilal, H. Alsibyani, and M. Canini, "Mitigating network side channel leakage for stream processing systems in trusted execution environments," in *Proc. 12th ACM Int. Conf. Distrib. Event-Based Syst.*, Jun. 2018, pp. 16–27.

[15] Yahoo Storm Team. (2015). *Benchmarking Streaming Computation Engines at Yahoo!* Accessed: Mar. 2017. [Online]. Available: https://yahooeng.tumblr.com/post/135321837876

[16] G. Theodorakis, A. Koliousis, P. Pietzuch, and H. Pirk, "LightSaber: Efficient window aggregation on multi-core processors," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, Jun. 2020, pp. 2505–2521, doi: 10.1145/3318464.3389753.

[17] P. M. Grulich, B. Sebastian, S. Zeuch, J. Traub, J. V. Bleichert, Z. Chen, T. Rabl, and V. Markl, "Grizzly: Efficient stream processing through adaptive query compilation," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, Jun. 2020, pp. 2487–2503, doi: 10.1145/3318464.3389739.

[18] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "BigDataBench: A big data benchmark suite from Internet services," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2014, pp. 488–499.

[19] Bigdatabench. (2020). *Bigdatabench: A Big Data Benchmark Suite, Benchcouncil*. Accessed: Jun. 2020. [Online]. Available: http://www.benchcouncil.org/BigDataBench/index.html

[20] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream bench: Towards benchmarking modern distributed stream computing frameworks," in *Proc. IEEE/ACM 7th Int. Conf. Utility Cloud Comput.*, Dec. 2014, pp. 69–78.

[21] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Proc. IEEE 26th Int. Conf. Data Eng. Workshops (ICDEW)*, 2010, pp. 41–51.

[22] (2020). *Apache Kafka*. [Online]. Available: https://kafka.apache.org/

[23] (2020). *Apache Cassandra*. [Online]. Available: https://cassandra.apache.org/

[24] P. Best, B. Taylor, R. Manktelow, and J. McQuilkin, "Systematically retrieving research in the digital age: Case study on the topic of social networking sites and young people's mental health," *J. Inf. Sci.*, vol. 40, no. 3, pp. 346–356, 2014, doi: 10.1177/0165551514521936.

[25] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "Spade: The system s declarative stream processing engine," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 1123–1134.

[26] H. Andrade, B. Gedik, K.-L. Wu, and P. S. Yu, "Scale-up strategies for processing high-rate data streams in system s," in *Proc. IEEE 25th Int. Conf. Data Eng. (ICDE)*, Mar. 2009, pp. 1375–1378.

[27] M. Dayarathna, S. Takeno, and T. Suzumura, "A performance study on operator-based stream processing systems," in *Proc. IISWC*, 2011, p. 79.

[28] M. Dayarathna and T. Suzumura, "A performance analysis of system S, S4, and Esper via two level benchmarking," in *Quantitative Evaluation of Systems*. Springer, 2013, pp. 225–240.

[29] S. Chakravarthy, *Stream Data Processing: A Quality of Service Perspective: Modeling, Scheduling, Load Shedding, and Complex Event Processing*, vol. 36. Springer, 2009.

[30] M. Smit, B. Simmons, and M. Litoiu, "Distributed, application-level monitoring for heterogeneous clouds using stream processing," *Future Gener. Comput. Syst.*, vol. 29, no. 8, pp. 2103–2114, Oct. 2013.

[31] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "TimeStream: Reliable stream computation in the cloud," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 1–14.

[32] M. A. Lopez, A. G. P. Lobato, and O. C. M. B. Duarte, "A performance comparison of open-source stream processing platforms," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Washington, DC, USA, Dec. 2016, pp. 1–6.

[33] D. S. Turaga, H. Park, R. Yan, and O. Verscheure, "Adaptive multimedia mining on distributed stream processing systems," in *Proc. IEEE Int. Conf. Data Mining Workshops*, Dec. 2010, pp. 1419–1422.

[34] P. Bellavista, A. Corradi, and A. Reale, "Design and implementation of a scalable and QoS-aware stream processing framework: The quasit prototype," in *Proc. IEEE Int. Conf. Green Comput. Commun. (GreenCom)*, Nov. 2012, pp. 458–467.

[35] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters," in *Proc. 4th USENIX Conf. Hot Topics Cloud Comput.*, 2012, p. 10.

[36] J. Chauhan, S. A. Chowdhury, and D. Makaroff, "Performance evaluation of Yahoo! S4: A first look," in *Proc. 7th Int. Conf. P2P, Parallel, Grid, Cloud Internet Comput. (3PGCIC)*, Nov. 2012, pp. 58–65.

[37] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: Fault-tolerant stream processing at Internet scale," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[38] Q. Zou, H. Wang, R. Soulé, M. Hirzel, H. Andrade, B. Gedik, and K.-L. Wu, "From a stream of relational queries to distributed stream processing," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 1394–1405, Sep. 2010.

[39] T. Hunter, T. Moldovan, M. Zaharia, S. Merzgui, J. Ma, M. J. Franklin, P. Abbeel, and A. M. Bayen, "Scaling the mobile millennium system in the cloud," in *Proc. 2nd ACM Symp. Cloud Comput.*, 2011, p. 28.

[40] S. Geisler and C. Quix, "Evaluation of real-time traffic applications based on data stream mining," in *Data Mining for Geoinformatics*. Springer, 2014, pp. 83–103.

[41] A. Artikis, M. Weidlich, F. Schnitzler, I. Boutsis, T. Liebig, N. Piatkowski, C. Bockermann, K. Morik, V. Kalogeraki, J. Marecek, A. Gal, S. Mannor, D. Kinane, and D. Gunopulos, "Heterogeneous stream processing and crowdsourcing for urban traffic management," in *Proc. EDBT*, 2014, pp. 712–723.

[42] M. Hanif, H. Yoon, and C. Lee, "Benchmarking tool for modern distributed stream processing engines," in *Proc. Int. Conf. Inf. Netw. (ICOIN)*, 2019, pp. 393–395.

[43] Y. Wang, "Stream processing systems benchmark: Streambench," Ph.D. dissertation, School Sci., Aalto Univ., Espoo, Finland, 2016.

[44] E. Shahverdi and S. Sakr, "Comparative evaluation for the performance of big stream processing systems," Tech. Rep., 2018.

[45] S. Zeuch, B. D. Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl, "Analyzing efficient stream processing on modern hardware," *Proc. VLDB Endowment*, vol. 12, no. 5, pp. 516–530, Jan. 2019.

[46] H. Nasiri, S. Nasehi, and M. Goudarzi, "Evaluation of distributed stream processing frameworks for IoT applications in smart cities," *J. Big Data*, vol. 6, no. 1, p. 52, Dec. 2019.

[47] Y. Simmhan, B. Cao, M. Giakkoupis, and V. K. Prasanna, "Adaptive rate stream processing for smart grid applications on clouds," in *Proc. 2nd Int. Workshop Sci. Cloud Comput.*, 2011, pp. 33–38.

[48] B. Lohrmann and O. Kao, "Processing smart meter data streams in the cloud," in *Proc. 2nd IEEE PES Int. Conf. Exhib. Innov. Smart Grid Technol.*, Dec. 2011, pp. 1–8.

[49] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proc. 7th ACM Int. Conf. Distrib. Event-Based Syst.*, 2013, pp. 207–218.

[50] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, New York, NY, USA, 2013, pp. 725–736.

[51] S. Girtelschmid, M. Steinbauer, V. Kumar, A. Fensel, and G. Kotsis, "On the application of big data in future large scale intelligent smart city installations," *Int. J. Pervasive Comput. Commun.*, vol. 10, no. 2, p. 4, 2014.

[52] R. C. Fernandez, M. Weidlich, P. Pietzuch, and A. Gal, "Scalable stateful stream processing for smart grids," in *Proc. 8th ACM Int. Conf. Distrib. Event-Based Syst.*, 2014, pp. 276–281.

[53] G. Hesse, B. Reissaus, C. Matthies, M. Lorenz, M. Kraus, and M. Uflacker, "Senska—Towards an enterprise streaming benchmark," in *Proc. Technol. Conf. Perform. Eval. Benchmarking*. Springer, 2017, pp. 25–40.

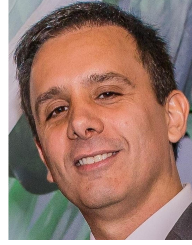
[54] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song, "Design and evaluation of a real-time URL spam filtering service," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2011, pp. 447–462.
[55] B. Chandramouli, J. J. Levandoski, A. Eldawy, and M. F. Mokbel, "StreamRec: A real-time recommender system," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 1243–1246.
[56] T. L. A. de Souza Ramos, R. S. Oliveira, A. P. de Carvalho, R. A. C. Ferreira, and W. Meira, Jr., "Watershed: A high performance distributed stream processing system," in *Proc. 23rd Int. Symp. Comput. Archit. High Perform. Comput. (SBAC-PAD)*, Oct. 2011, pp. 191–198.
[57] T. Chardonnens, P. Cudre-Mauroux, M. Grund, and B. Perroud, "Big data analytics on high velocity streams: A case study," in *Proc. IEEE Int. Conf. Big Data*, Oct. 2013, pp. 784–787.
[58] L. Lin, X. Yu, and N. Koudas, "Pollux: Towards scalable distributed real-time search on microblogs," in *Proc. 16th Int. Conf. Extending Database Technol.*, 2013, pp. 335–346.
[59] F. Alvanaki and S. Michel, "Scalable, continuous tracking of tag co-occurrences between short sets using (almost) disjoint tag partitions," in *Proc. ACM SIGMOD Workshop Databases Social Netw.*, 2013, pp. 49–54.
[60] T. Lunze, P. Katz, D. Röhrborn, and A. Schill, "Stream-based recommendation for enterprise social media streams," in *Business Information Systems*. Springer, 2013, pp. 175–186.
[61] C. Chen, H. Yin, J. Yao, and B. Cui, "Terec: A temporal recommender system over tweet stream," *Proc. VLDB Endowment*, vol. 6, no. 12, pp. 1254–1257, 2013.
[62] Z. Nabi, E. Bouillet, A. Bainbridge, and C. Thomas, "Of streams and storms," IBM, Endicott, NY, USA, White Paper, 2014.
[63] E. Bouillet, R. Kothari, V. Kumar, L. Mignet, S. Nathan, A. Ranganathan, D. S. Turaga, O. Udrea, and O. Verscheure, "Processing 6 billion CDRs/day: From research to production (experience report)," in *Proc. 6th ACM Int. Conf. Distrib. Event-Based Syst.*, 2012, pp. 264–267.
[64] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, pp. 2351–2365, Jan. 2012.
[65] M. A. Abbasoğlu, B. Gedik, and H. Ferhatosmanoğlu, "Aggregate profile clustering for telco analytics," *Proc. VLDB Endowment*, vol. 6, no. 12, pp. 1234–1237, Aug. 2013.
[66] L. Pan, J. Qian, C. He, W. Fan, C. He, and F. Yang, "NIM: Scalable distributed stream process system on mobile network data," in *Proc. IEEE 13th Int. Conf. Data Mining Workshops*, Dec. 2013, pp. 1101–1104.
[67] D. Simoncelli, M. Dusi, F. Gringoli, and S. Niccolini, "Scaling out the performance of service monitoring applications with blockmon," in *Passive and Active Measurement*. Springer, 2013, pp. 253–255.
[68] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in *Proc. IEEE 34th Int. Conf. Data Eng. (ICDE)*, Apr. 2018, pp. 1507–1518.
[69] P. Balaprakash, D. Buntinas, A. Chan, A. Guha, R. Gupta, S. H. K. Narayanan, A. A. Chien, P. Hovland, and B. Norris, "Exascale workload characterization and architecture implications," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Apr. 2013, p. 5.
[70] Y. Bai and C. Zaniolo, "Minimizing latency and memory in DSMS: A unified approach to quasi-optimal scheduling," in *Proc. 2nd Int. Workshop Scalable Stream Process. Syst.*, 2008, pp. 58–67.
[71] B. Babcock, S. Babu, R. Motwani, and M. Datar, "Chain: Operator scheduling for memory minimization in data stream systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2003, pp. 253–264.
[72] Y. Wei, V. Prasad, S. Son, and J. Stankovic, "Prediction-based QoS management for real-time data streams," in *Proc. 27th IEEE Int. Real-Time Syst. Symp. (RTSS)*, Dec. 2006, pp. 344–358.
[73] K.-A. Yoon, O.-S. Kwon, and D.-H. Bae, "An approach to outlier detection of software measurement data using the K-means clustering method," in *Proc. 1st Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Sep. 2007, pp. 443–445.
[74] (2020). *Geotools Library*. [Online]. Available: https://www.osgeo.org/projects/geotools/
[75] I. Androutsopoulos, J. Koutsias, K. V. Chandrinos, G. Paliouras, and C. D. Spyropoulos, "An evaluation of naive Bayesian anti-spam filtering," 2000, *arXiv:cs/0006013*. [Online]. Available: https://arxiv.org/abs/cs/0006013
[76] M. Mathioudakis and N. Koudas, "TwitterMonitor: Trend detection over the Twitter stream," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 1155–1158.
[77] A. Srivastava, A. Kundu, S. Sural, and A. K. Majumdar, "Credit card fraud detection using hidden Markov model," *IEEE Trans. Dependable Secure Comput.*, vol. 5, no. 1, pp. 37–48, Jan. 2008.
[78] M. Dayarathna and T. Suzumura, "Automatic optimization of stream programs via source program operator graph transformations," *Distrib. Parallel Databases*, vol. 31, no. 4, pp. 543–599, Dec. 2013.
[79] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, "IBM infosphere streams for scalable, real-time, intelligent transportation services," in *Proc. Int. Conf. Manage. Data SIGMOD*, New York, NY, USA, 2010, pp. 1093–1104, doi: 10.1145/1807167.1807291.
[80] A. L. Strehl and M. L. Littman, "An analysis of model-based interval estimation for Markov decision processes," *J. Comput. Syst. Sci.*, vol. 74, no. 8, pp. 1309–1331, Dec. 2008.
[81] G. Bianchi, N. d'Heureuse, and S. Niccolini, "On-demand time-decaying Bloom filters for telemarketer detection," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 5, pp. 5–12, Oct. 2011.
[82] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proc. IEEE Int. Conf. Data Mining Workshops*, Dec. 2010, pp. 170–177.
[83] B. Chandramouli, J. Goldstein, R. Barga, M. Riedewald, and I. Santos, "Accurate latency estimation in a distributed event processing system," in *Proc. IEEE 27th Int. Conf. Data Eng. (ICDE)*, Apr. 2011, pp. 255–266.
[84] (2020). *Ganglia Monitoring System*. [Online]. Available: http://ganglia.info/

MAYCON VIANA BORDIN received the master's degree in computer science from the Federal University of Rio Grande do Sul (UFRGS), in 2017, where he is still a Research Collaborator. He is currently a Data Engineer with Sicredi, working with big data, cloud computing, and software engineering tools. His research interests include distributed computing, cloud computing, data intensive computing, big data, and data stream processing.

DALVAN GRIEBLER received the Ph.D. degree in computer science from PUCRS and the University of Pisa, Italy, in 2016. He is currently a part-time Professor with the Faculdade Três de Maio, SETREM, the Head of the Laboratory of Advanced Research on Cloud Computing (LARCC), SETREM, a Postdoctoral Researcher with the Pontifical Catholic University of Rio Grande do Sul (PUCRS), and a Research Coordinator with the Parallel Applications Modeling Group (GMAP), PUCRS. His research interests include parallel and distributed computing, cloud computing, stream processing, high-level parallel programming, and autonomic and self-adaptive systems.

GABRIELE MENCAGLI received the Ph.D. degree in computer science from the University of Pisa, Italy, in 2012. He is currently an Assistant Professor with the Department of Computer Science, University of Pisa. He is also a member of the Parallel Programming Models (PPMs) Group. His research interests include parallel and distributed systems, autonomic computing, data stream processing, and distributed systems. He is a member of the Editorial Board of *Future Generation Computer Systems* (Elsevier) and *Cluster Computing* (Springer), and a member of the Program Committee of several conferences in his research fields.

CLÁUDIO F. R. GEYER received the degree in mechanical engineering and the master's degree in computer science from the Federal University of Rio Grande do Sul (UFRGS), in 1978 and 1986, respectively, and the Ph.D. degree in informatics from the Universite de Grenoble I (Scientifique Et Medicale–Joseph Fourier), in 1991. He is currently a Full Professor with UFRGS. He has experience in computer science, with an emphasis on computer systems, acting mainly in following themes: pervasive (ubiquitous) computing, grid, cloud and voluntary computing, tools for big data applications, and multiplayer game support, with an emphasis on building middleware for these areas of computing.



LUIZ GUSTAVO L. FERNANDES received the Ph.D. degree in computer science from the Institut National Polytechnique de Grenoble, France, in 2002. He is currently an Associate Professor of the Graduate Program in Computer Science (PPGCC) with the Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil. He also leads the Parallel Applications Modeling Group (GMAP), PUCRS. His primary research interests include parallel and distributed computing, high-performance applications modeling, green computing, and parallel programming interfaces.