Quantitative Impact Evaluation of an Abstraction Layer for Data Stream Processing Systems

Guenter Hesse*, Christoph Matthies*, Kelvin Glass†, Johannes Huegle* and Matthias Uflacker*
*Hasso Plattner Institute
University of Potsdam
Email: firstname.lastname@hpi.de
†Department of Mathematics and Computer Science
Freie Universität Berlin
Email: kelvin.glass@fu-berlin.de

***Abstract*—With the demand to process ever-growing data volumes, a variety of new data stream processing frameworks have been developed. Moving an implementation from one such system to another, e.g., for performance reasons, requires adapting existing applications to new interfaces. Apache Beam addresses these high substitution costs by providing an abstraction layer that enables executing programs on any of the supported streaming frameworks. In this paper, we present a novel benchmark architecture for comparing the performance impact of using Apache Beam on three streaming frameworks: Apache Spark Streaming, Apache Flink, and Apache Apex. We find significant performance penalties when using Apache Beam for application development in the surveyed systems. Overall, usage of Apache Beam for the examined streaming applications caused a high variance of query execution times with a slowdown of up to a factor of 58 compared to queries developed without the abstraction layer. All developed benchmark artifacts are publicly available to ensure reproducible results.**

***Index Terms*—Data Stream Processing, Abstraction Layer, Performance Benchmarking**

I. Introduction

While the world of Big Data analysis presents a multitude of opportunities, it also comes with unique obstacles. For software developers, who are tasked with building Big Data applications, the need to work with many different frameworks, Application Programming Interfaces (APIs), programming languages, and software development kits (SDKs) can be challenging. Especially as the long-lasting paradigm "one size fits all" seems to be obsolete, which Stonebraker and Çetintemel [1] already predicted back in 2005, handling and keeping an overview of the multitude of technologies becomes more difficult. Furthermore, flexibility is crucial in our rapidly changing world in order to maintain or establish a leader position. An example for that is a desired change of an IT system for reasons such as variations in pricing policy, changed volume or velocity of data that needs to be processed, or altered performance characteristics of execution engines. Ideally, this adaption should happen at minimal costs, which is a challenging task if each system uses its own APIs.

The wide variety of available tools in the area of stream processing has spurred the development of an abstraction layer, which allows defining programs independent of the technologies used. This layer is the open source project Apache Beam [2], which provides a unified programming model for describing both batch and streaming data-parallel processing pipelines. Pipelines are described using a single Software Development Kit (SDK) and can then be executed by a variety of different frameworks, without developers needing detailed knowledge of the employed implementations. Thus, execution frameworks can be exchanged without the need to adapt code. As an additional benefit, Apache Beam enables to benchmark multiple systems with a single implementation. Conceptually, this idea can be compared to object-relational mapping (ORM), where data stored in database tables is encapsulated in objects. Data can be queried and manipulated just by using these objects instead of writing SQL [3].

A question concerning abstraction layers is if their usage has consequences on the performance characteristics of an application. If introduced performance penalties are too high, they might outweigh the gained benefits. Great performance impact variations between systems can, with regard to performance benchmarking, lead to a distorted result. Thus, it is crucial to understand and quantify the impact of used abstraction layers. The following contributions are presented in this paper:

- We give a description of Apache Beam as well as the three data stream processing systems (DSPSs) used for the conducted measurements, which are Apache Flink [4], Apache Spark Streaming [5], and Apache Apex [6].
- We propose a lightweight benchmark architecture for comparing DSPSs. This benchmark covers information on the process as well as data, query, and metric aspects. All developed artifacts are available online which ensures transparency and reproducibility.
- We present the measurement results with focus on the performance impact of Apache Beam. That is done by comparing the query implementations using native system APIs with the implementations using Apache Beam.

The remainder of this paper is structured as follows: Section II describes the employed technologies. Section III illustrates the benchmark environment and the performance results. Section IV, gives an overview of related work and Section V concludes, giving a summary on results and highlighting areas for future work.

II. Data Stream Processing System Technologies

This section describes the technologies used for the presented measurements.

A. Apache Beam

Apache Beam [2] describes itself as a unified programming model, which allows defining batch and stream processing applications. For that, Apache Beam SDKs are provided. Currently, three SDKs are part of the Apache Beam repository: a Java SDK, a Python SDK, and a Go SDK [7].

Instead of developing an application for a single DSPS, Apache Beam allows writing programs that are compatible with any supported execution engine. Engine-specific runners translate the Apache Beam code to the target runtime. Using such an abstraction layer theoretically allows, e.g., for an arbitrary exchange of engines without the need of code adaption. Central elements of the Apache Beam SDK are:

- **Pipeline** represents the entire application definition, including data input, transformation, and output.
- **PCollection** embodies a distributed data set that can be either bounded or unbounded. The latter is used for data stream processing applications.
- **PTranform** is an abstraction for data transformation. It receives one or more PCollection objects and applies the a transformation on this data. That leads to an output of zero or more PCollection objects. Moreover, read or write operations on external storage systems are realized with PTransform objects. Apache Beam provides some core transforms. Selected ones are outlined in the following:
  - **ParDo** is an element-by-element processing of data, whereby the processing of a single element can lead to zero or more output elements. In addition to standard operations like map or flat map, a ParDo also supports aspects such as side inputs and stateful processing.
  - **GroupByKey**, as the name already states, processes key-value pairs and collects all values belonging to the same key. It is an aggregation operation that outputs pairs consisting of a key and a collection of values that belong to this key. For use with data streams, one must use an aggregation trigger or non-global windowing in order to enable the grouping to be applied to a finite data set.
  - **Flatten** merges the data of multiple PCollection objects that contain data of the same type into a single PCollection [8]–[10]

Next to Apache Flink, Apache Spark, and Apache Apex, other frameworks supporting Apache Beam exist [11]. These are Apache Gearpump [12], Apache Hadoop MapReduce [13], Apache Samza [14], Alibaba JStorm [15], IBM Streams [16], and Google Cloud Dataflow [17]. This group covers both, closed source systems, e.g., Google Cloud Dataflow and IBM Streams, as well as open source systems, e.g., Apache Flink and Apache Spark. Hence, Apache Beam can be seen as a widely spread project with a high relevance.

Apache Beam itself resulted out of the donation of the Cloud Dataflow SDKs and programming model [10] to the Apache Software Foundation [18]. Therefore, as mentioned before, Google Cloud Dataflow is one supported system.

B. Apache Flink

Apache Flink is an open source system with batch and stream processing capabilities. It offers a Java and a Scala API for developing applications. Additionally, there are multiple libraries on top of Apache Flink that provide, e.g., machine learning or graph processing functionalities [4], [19]. The architecture of an Apache Flink cluster is shown in Figure 1.

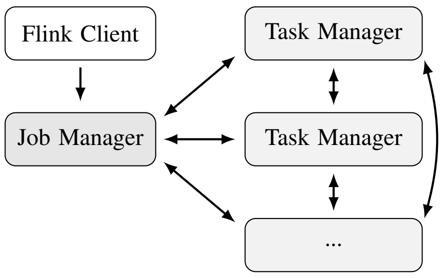

Fig. 1. Architecture of an Apache Flink Runtime (based on [4], [19])

Figure 1 shows an Apache Flink Client, a Job Manager, and Task Managers. When a program is deployed to the system, the client transforms it into a dataflow graph, i.e., a directed acyclic graph (DAG), and sends it to the Job Manager. The client itself is not part of the program execution and can, after transmitting the dataflow graph, either disconnect from the Job Manager or stay connected in order to receive information about the execution progress.

The Job Manager or master is responsible for scheduling work amongst the Task Manager instances and for keeping track of the execution. There can be multiple Job Manager instances whereas only one Job Manager can be the leader. Others would be standby and could take over in case of failure.

The Task Manager instances execute the assigned parts of the program. Technically, a Task Manager is a JVM process. There must be at least one Task Manager in an Apache Flink installation. Thereby, they exchange data amongst each other where needed. Each Task Manager provides at least one task slot in which subtasks are executed in multiple threads. A task slot can be shared by multiple subtasks as long as they belong to the same application, even if they are part of different tasks. While one task is executed by one thread, Apache Flink chains multiple operator subtasks into a single task, such as two subsequent map operations. A benefit of this optimization is, e.g., a reduced overhead for inter-thread communication.

Every task slot has a subset of the resources that belong to its corresponding Task Manager. Particularly, the available memory is split amongst task slots. CPU separation does not happen in the current Apache Flink version [4], [19], [20].

C. Apache Spark Streaming

Apache Spark is another open source system for distributed data processing. It offers, next to batch processing functionalities, stream processing features as part of its library Apache Spark Streaming. However, stream processing is implemented using micro-batches, i.e., it is not a tuple-by-tuple processing as in Apache Flink. Apache Spark Streaming applications can be written in Java, Scala, or Python. Besides Apache Spark Streaming, there are other libraries built on top of Apache Spark, e.g., similar to Apache Flink's ecosystem, a library for machine learning as well as for graph processing [5], [21].

The architecture of an Apache Spark installation is shown in Figure 2. An application is executed in the form of multiple independent processes distributed across a cluster. The SparkContext coordinates these processes. This coordinator is an object in the *main()* function of the application, which is called Driver Program. Moreover, the SparkContext connects to a Cluster Manager that takes care of resource allocation.

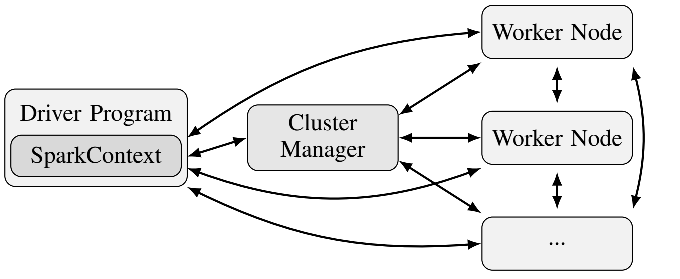

Fig. 2. Architecture of Apache Spark in Cluster Mode (based on [19], [22])

Currently, there are four Cluster Managers supported by Apache Spark - Spark Standalone, Apache Mesos [23], Apache Hadoop YARN (Yet Another Resource Negotiator) [24], and Kubernetes [25]. As soon as a connection is established, the SparkContext acquires so-called executors on the Worker Node instances. Each executor is a process belonging to exactly one application, which stores data and performs computations. So different applications running on the same Apache Spark Cluster are executed in different JVMs, which is different to, e.g., the execution concept in the previously illustrated Apache Flink. Thus, data cannot be exchanged between different Apache Spark application without making use of an external storage system.

Once executors are acquired, the SparkContext transmits the program in the form of a JAR or Python files to them. Afterwards, it sends tasks to the executor processes. One process can run multiple tasks in several threads [22], [26].

A central data structure that is used in Apache Spark is the Resilient Distributed Dataset (RDD). An RDD can be viewed as a distributed memory abstraction. To be more concrete, it is a partitioned and read-only collection of records. Apache Spark Streaming leverages a processing model called discretized streams (D-Streams). Such a D-Stream is a sequence of RDDs. An incoming data stream is divided into batches stored in RDDs. Data transformations are then performed on these RDDs, which again output a D-Stream [27], [28].

D. Apache Apex

Apache Apex is based on Apache Hadoop [29] with its components Apache Hadoop YARN and Hadoop Distributed File System (HDFS) [30]. Similar to Apache Flink, it offers batch as well as stream processing functionalities. Moreover, stream processing is also implemented in a way of processing data in a tuple-by-tuple fashion [31], [32].

Apex Malhar, built on top of Apex Core, is a library containing different input/output operators and compute operators. The former group includes, e.g., connectors to Apache Kafka [33] and other messaging systems [6]. The high-level architecture of Apache Hadoop 2, which is the version that is used for the presented measurements, is depicted in Figure 3. HDFS at the bottom is a distributed file system as its name already states and serves as the storage layer. Apache Hadoop YARN acts as a resource manager on top of HDFS. On top of YARN, there are multiple data processing frameworks available from various areas such as batch or stream processing. Two of those are Hadoop MapReduce and Apache Apex. The latter is the stream processing system used for measurements described in this paper [34].

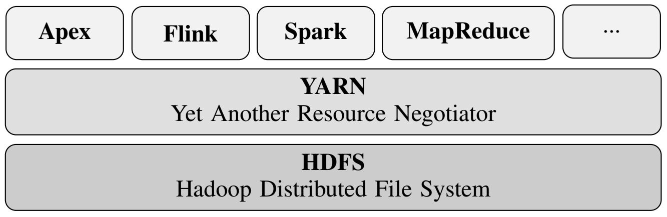
Apex
Flink
Spark
MapReduce
...
YARN
Yet Another Resource Negotiator
HDFS
Hadoop Distributed File System

Fig. 3. Architecture of Apache Hadoop 2 (based on [34])

The two other DSPSs used for those measurements, i.e., Apache Spark (Streaming) and Apache Flink, can also run on Apache Hadoop YARN as one of multiple deployment options [35], [36]. However, for the presented measurements we use the standalone cluster deployment option that is available in both systems, Apache Flink and Apache Spark.

Figure 4 illustrates the architecture of an Apache Hadoop YARN deployment. The depicted installation runs one application. Its components are marked with dashed lines.

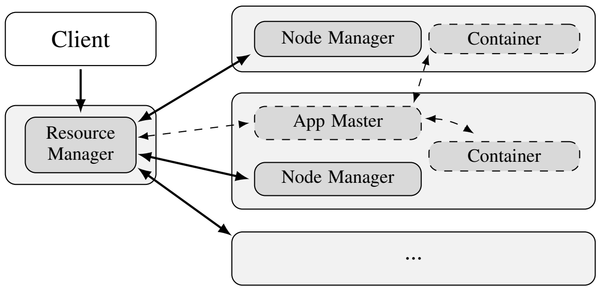

Fig. 4. Architecture of an Apache Hadoop YARN (based on [24], [37])

Two major components are part of Apache Hadoop YARN, a Resource Manager and Node Manager instances. Both are daemons running on defined nodes. A client submits an application to the Resource Manager. It is responsible for distributing cluster resources amongst applications. Particularly, the Resource Manager allocates so-called containers on dedicated cluster nodes for applications. A container is defined as a logical bundle of resources, e.g., a bundle of 4GB RAM and 1 CPU, that is tied to a certain node. Communication between the Resource Manager and the Node Manager daemons happens via a heartbeat mechanism [24].

There is one special container spawned for each program, the Application Master. It manages application execution with respect to, e.g., resource needs, execution flow, or fault handling. The Application Master can be written in any programming language, though many applications make use of higher-level frameworks such as MapReduce or Apache Apex. Other containers may communicate directly with the Application Master if necessary. This communication would need to be managed by the application as YARN does not arrange that. The Application Master implemented in Apache Apex is called Streaming Application Manager (STRAM) [24], [38].

E. Similarities and Differences Between the Presented Systems

Table I gives an overview of the presented DSPSs. Apache Beam is not listed as it is not a DSPS but an SDK for developing stream processing programs.

Criteria	Apache Flink	Apache Spark Streaming	Apache Apex
Mainly Written in	Java, Scala	Scala, Java, Python	Java
Languages for App Development	Java, Scala, Python	Scala, Java, Python	Java
Data Processing	Tuple-by-tuple	Batch	Tuple-by-tuple
Processing Guarantees	Exactly-once	Exactly-once	Exactly-once

TABLE I
COMPARISON OF APACHE FLINK, APACHE SPARK STREAMING, AND APACHE APEX (BASED ON [19], [39]–[41])

All systems are mainly developed in a JVM language, specifically Java or Scala [39]–[41]. Programs can be written in either Java, Scala, or Python on Apache Flink and Apache Spark Streaming. Apache Apex only offers the possibility to write applications in Java. However, there are plans to also support the development of programs in Scala [1]. With respect to data processing characteristics, Apache Flink and Apache Apex process data in a tuple-by-tuple fashion. Contrary, Apache Spark Streaming makes use of a batch processing approach. Furthermore, all three systems guarantee exactly-once processing, i.e., each input tuple is processed exactly once. This ensures correct results also in recovery scenarios.

III. PERFORMANCE ANALYSIS

This section describes the conducted performance analysis. First, the general benchmark setup as well as the data used for the benchmark are presented. Afterwards, the executed queries are highlighted. Lastly, the performance results are discussed. That particularly includes an analysis of the execution times, standard deviation, and a detailed view of the performance impact of Apache Beam. Query implementations and other used programs and scripts can be found online [2].

[1]https://malhar.atlassian.net/browse/APEX-175

[2]http://hpi.de/fileadmin/user_upload/fachgebiete/plattner/publications/papers/gh/StreamBenchOnApacheBeamBenchmark.zip

A. Benchmark Architecture and Process

The proposed benchmark setting is depicted in Figure 5. The benchmark process is divided into three separate and consecutive phases. The components that are involved in the corresponding part of the process are marked in the figure by dashed curly brackets.

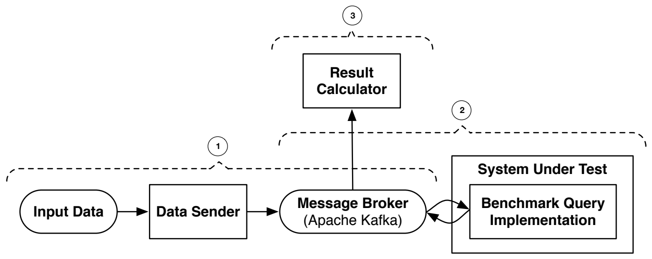

Fig. 5. Overview About the General Benchmark Architecture and Process Based on Fundamental Modeling Concepts (FMC)

On the left-hand side there is the input data which is read by the data sender and forwarded to the message broker, which in particular is Apache Kafka. The data sender is a program written in Scala with multiple configuration parameters such as the data ingestion rate or the level of Kafka Producer acknowledgments. The system under test (SUT) on the right-hand side, i.e., the DSPS to be benchmarked, executes the implemented query. Thereby, it reads from and writes to the message broker. Moreover, there is a result calculator tool developed in Scala that reads the query output from the message broker and leverages Apache Kafka functionality for the execution time computation. The three different benchmark process steps marked in Figure 5 are described in the following:

1) Data Ingestion: Firstly, data is inserted into an Apache Kafka topic using the data sender. Particularly, 1,000,001 records of the AOL Search Query Log [42] dataset that is also used in [43] are sent. The input topic is created with a replication factor of one and one partition in order to ensure the correct order of messages. Apache Kafka only guarantees the correct order for entries within one partition [44]. The data file consists of records with five tab-separated columns. These columns contain a user ID, the query issued by the user, the time at which the query was issued, the search result rank the user clicked on if applicable, and the search result Uniform Resource Locator (URL) the user clicked on if applicable [42].

2) Program Execution: During the execution phase, each query is run ten times for each execution setup. Meanwhile, there are no other programs executed on the system and each system is restarted at the beginning of this benchmarking step. The stream processing program computes the output and sends it to an Apache Kafka topic that is also created with a replication factor of one and one partition for the same reasons as mentioned previously. Each query is executed with a parallelism of one and two, ten times each. Moreover, every query is implemented using the APIs provided by the DSPS as well as using Apache Beam. So for each query, as we analyzed three different systems, there are twelve different query execution setups as it can be seen in Section III-C.

The mentioned parallelism is set differently depending on the system and the used APIs. Regarding Apache Flink, it is configured using the command line option *-p* or *--parallelism* that is offered when submitting an application for specifying parallelism [45]. For programs executed on Apache Spark Streaming, the configuration parameter *spark.default.parallelism* is used [46].

Apache Apex does not provide an option for configuring parallelism, so instead the number of VCOREs is set accordingly in the Apache Hadoop YARN configuration [3] as well as within the Apache Apex application as a DAG attribute [47]. The approach of using this configuration is also applied for the programs running on Apache Apex that are developed using Apache Beam APIs. Details can be found in the mentioned archive that is provided online.

3) Result Calculation: Lastly, the result records are read from Apache Kafka for each query and the time difference between the firstly inserted and the lastly inserted record is computed. Apache Kafka is configured to use *LogAppendTime*, i.e., the timestamp when a record is appended to the Apache Kafka log is stored together with the record itself [44]. For execution time calculation, we use these timestamps which allows keeping the measurements application- and system-independent. That is a crucial benefit with respect to result correctness as definitions of performance criteria vary among systems. Thus, one cannot rely on performance data provided by DSPSs [48]. The overhead between having the correct result computed within the SUT and having it appended to Apache Kafka log is identical for every system and hence, results are comparable.

With regard to the hard- and software setup, virtual machines are used for all nodes. Apache Kafka version 2.11-0.10.1.0 is installed on a three node cluster with 64GB main memory and an Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz CPU with eight cores each. The DSPSs are installed on a two node cluster where both nodes act as worker nodes or the equivalent. These two nodes are identical to the Apache Kafka nodes with regard to both, main memory and CPU. Ubuntu 14.04 is installed as the operating system on all virtual machines. Regarding system and framework versions, Apache Apex 3.7.0, Apache Hadoop 2.7.3, Apache Spark 2.3.0, Apache Flink 1.4.0, and Apache Beam 2.3.0 are used. The configuration files for the different systems can be found in the previously linked archive.

B. Benchmarked Queries

The executed queries are taken from the StreamBench [43] benchmark. StreamBench defines seven different queries. Four of these are stateless, i.e., it is not required to keep a state for producing the correct answer. The remaining three queries are stateful. The stateless queries used for the benchmark presented in this paper are listed in Table II. Stateful queries are excluded as Apache Beam does not support stateful processing when executed on Apache Spark, see [11].

[3]http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-common/yarn-default.xml

Query	Description
Identity	Read input and output it without performing any data transformation. Can be seen as a baseline query with respect to computational complexity.
Sample	Read input and output only a certain percentage of data that is randomly chosen. The number of output tuples is as big as about 40% of the number of input tuples.
Projection	Read input and output only a certain column of the input record. In the presented measurements, the values of the first column are chosen for being included in the output.
Grep	Read input and output only records that match a certain regex. The search string used for the measurements is ”test”, which leads to an output of 3,003 records or about 0.3% of the number of input records.

TABLE II
OVERVIEW OF THE BENCHMARK QUERIES (BASED ON [43])

C. Performance Results

This section illustrates the performance results regarding execution times and the performance impact of Apache Beam.

1) Execution Times: The following charts visualize the measured average execution times. On the y-axis, the combinations of system, parallelism, and kind of implementation, i.e., using Apache Beam or system APIs, are listed. The x-axis shows the times in seconds. The letter *P* stands for parallelism.

Figure 6 shows the results for the identity query. It can be seen that the query implementations using Apache Beam are slower compared to the implementations using the APIs provided by the corresponding system in all cases. That is true for almost all measurements presented in the following.

The overall shortest execution time belongs to queries run on Apache Spark Streaming, closely followed by Apache Apex and Apache Flink, both of which have a noticeable slower average runtime for one kind of parallelism. That could be due to outliers in the corresponding series of runs. Details on that can be found in Section III-C2.

When looking at the runtimes of queries implemented using on Apache Beam, differences are much larger. Apache Beam queries running on Apache Apex have by far the highest execution times with around 240s. So the differences between the execution times of the analyzed systems are significantly higher for the queries implemented using Apache Beam compared to those developed using native system APIs. In comparison to these variances, distinctions between parallelism factors are very small.

Nevertheless, the result differences between parallelism factors of the Apache Beam query running on Apache Spark Streaming is noticeable. The average execution time for the parallelism of two is close to 70% higher compared to these for the parallelism factor of one. As the relative standard deviation for these benchmark runs is low as illustrated later on in Figure 10, that is not due to outliers. A reason for this observation could be the introduced overhead with respect to, e.g., data transfer, that comes with splitting up tasks and that may not pay off for simple queries like the identity query.

Figure 7 displays the results for the sample query. Again, it can be seen that implementations using native system APIs outperform these using Apache Beam. Moreover, results of queries using the system APIs do not differ significantly be-

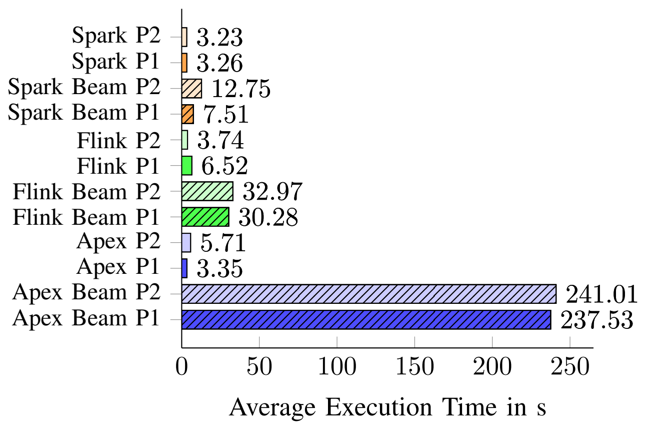

Fig. 6. Average Execution Times - Identity Query

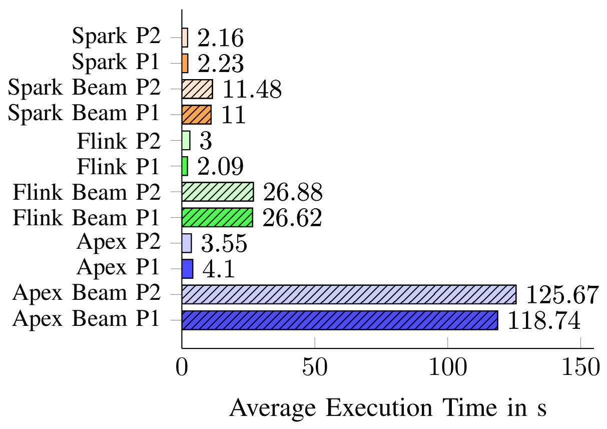

Fig. 7. Average Execution Times - Sample Query

tween the analyzed systems and parallelism factors. Compared to identity query results, times are slightly lower overall, which could be a result of the lower number of output records as described in Section III-B. The Apex Beam implementation is an exception as there is a major difference. To be more concrete, the average execution times for the sample query amount to only about 50% of the identity query times.

With average execution times of about 2.09s and 3s for the sample query developed using Apache Flink APIs for parallelisms of one and two respectively, these numbers are below the corresponding times for Apache Apex. Thus, the performance ranking between systems is identical for the sample query. That means, the times for Apache Spark are lowest, followed by these of Apache Flink and Apache Apex for both kinds of implementation.

The projection query results are shown in Figure 8. They are similar to the numbers for the identity query in all aspects. This closeness leads to the conclusion that splitting a string and accessing one column of the resulting list does not introduce a noticeable overhead. Regarding the number of output tuples, both queries are identical, though the tuple size for the projection query is smaller as only a subset of columns is sent to the output topic. However, this reduction in output size does not have a noticeable impact on the times either.

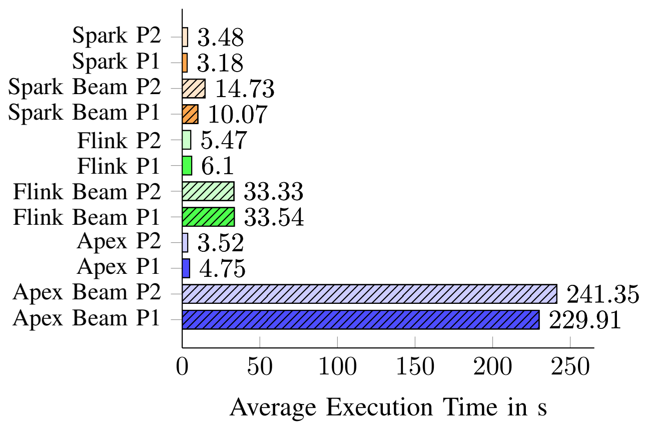

Fig. 8. Average Execution Times - Projection Query

Figure 9 visualizes the results for the grep query measurements. These times are overall the lowest ones whereas there are differences between systems and used APIs. Especially the implementations using the native APIs offered by Apache Spark Streaming and Apache Flink have relatively low execution times in comparison to the corresponding numbers for the other three queries. With about 20s, the Apache Beam version for Apache Flink is close to 7s faster than the corresponding sample query result with about 27s. There is no noticeable difference between parallelism factors.

Contrary to Apache Flink, the average execution times for queries developed using Apache Beam and running on Apache Spark Streaming differ amongst parallelism factors. Similar to the corresponding times for the identity query depicted in Figure 6, the average execution time for a parallelism factor of two is noticeably higher. In particular, with absolute times of about 11.8s and 6.34s, a parallelism factor of two slows down the average execution time by more than 85% in comparison to the time measured for a parallelism of one. Reasons for that can be as described for the identity query results.

A surprising result is the Apex Beam performance for Apache Apex. While the times for the native Apache Apex implementation is about on the same level as the corresponding results for all other queries, the ones for the query developed using Apache Beam are drastically lower. For the projection and the identity query, Apex Beam results are approximately between 230s and 240s. With about 120s, the sample query performance is already significantly better. However, with 2.58s and 3.76s, the execution times for the Apex Beam grep query implementation are orders of magnitude lower.

A reason for the relatively low execution times could lie in the number of output records and the resulting smaller effort that is needed for emitting query outcomes. To be more concrete, the output for the grep query is significantly lower than for the other three queries, though, as described in Section III-B, the sample query already outputs fewer tuples than the projection and the identity query.

2) Standard Deviation in Measured Execution Times: Figure 10 visualizes the relative standard deviations for the measurements. These values are calculated for every system-

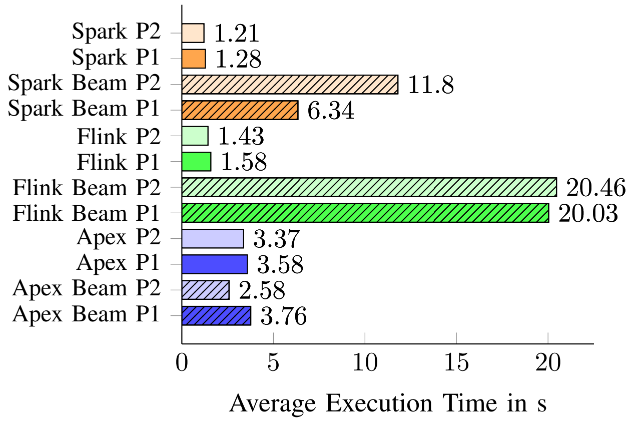

Fig. 9. Average Execution Times - Grep Query

query-SDK combination. By SDK it is distinguished between using Apache Beam or native system APIs for application development. Deviations for the two parallelism factors are averaged and condensed in this way. This is done since separate visualizations for different parallelisms would not reveal any further insights. Additionally, the reduced number of values simplifies analysis of standard deviations.

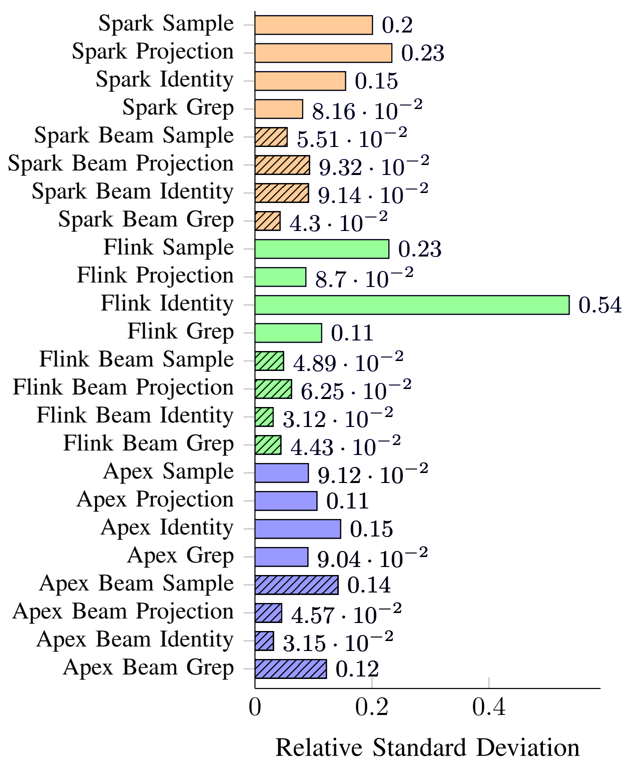

Fig. 10. Relative Standard Deviation for System-Query-SDK Combinations

There is one value that is notably higher than others, which belongs to the identity query executed on Apache Flink. Figure 6, which is visualizing the identity query execution times, makes visible that there is a noticeable difference between the numbers for the two parallelism factors for Apache Flink. Particularly, the execution time for Apache Flink with a parallelism of one is almost 75% higher than the one for Apache Flink with a parallelism of two. Although it seems to be plausible at a first glance that higher parallelism leads to better performance, this correlation is absent for other results.

Number of Run	Parallelism = 1	Parallelism = 2
1	6.25s	4.15s
2	21.56s	3.77s
3	3.42s	2.71s
4	3.31s	5.29s
5	3.73s	3.00s
6	12.69s	3.93s
7	3.90s	2.90s
8	3.96s	3.66s
9	3.42s	3.57s
10	3.01s	4.45s

TABLE III
EXECUTION TIMES FOR THE IDENTITY QUERY ON APACHE FLINK

Table III shows the execution times for the benchmark runs of the identity query on Apache Flink, i.e., numbers for the corresponding ten runs with a parallelism of one as well as for the ten runs with a parallelism of two. When looking at these measurements it becomes clear that there are two to three outliers that cause this relatively high coefficient of variation. While results for the higher parallelism are relatively homogeneous, there are outliers in the list of execution times for runs with a parallelism of one. Particularly, seven out of ten execution times range from three to four seconds. The results for the remaining benchmark runs differ significantly. To be more concrete, these runs lasted about 6s, 12.5s, and 21.5s. The highest execution time, e.g., is more than seven times higher than the lowest one. These outliers cause the comparatively high relative standard deviation. Apart from the identity query executed on Apache Flink, there are no further values that stand out in Figure 10.

3) Performance Impact of Apache Beam: The performance impact factors are calculated based on the arithmetic means of the execution times. These times are measured as defined in Section III-A. The averages are determined as follows:

$$\bar{t}(dsps, query, k, p) = \frac{1}{N_{run}} \sum_{r=1}^{N_{run}} t(dsps, query, k, p, r),$$

where $\bar{t}(dsps, query, k, p)$ denotes the average over the execution times for a certain data stream processing system, query, kind of implementation, i.e., using Apache Beam or native system APIs, and a certain parallelism. Variable k represents the mentioned kind of implementation and p stands for the used degree of parallelism. The number of benchmark runs is expressed as N_{run}, which is equal to ten for the context of this paper. The execution time for a single query run of a certain benchmark scenario is shown as $t(dsps, query, k, p, r)$.

The slowdown factor calculation makes use of these arithmetic means. Specifically, it is computed as follows:

$$sf(dsps, query) = \frac{1}{N_p} \sum_{p=1}^{N_p} \frac{\bar{t}(dsps, query, Beam, p)}{\bar{t}(dsps, query, native, p)},$$

where $sf(dsps, query)$ denotes the slowdown factor for a given data stream processing system and a given query. N_p

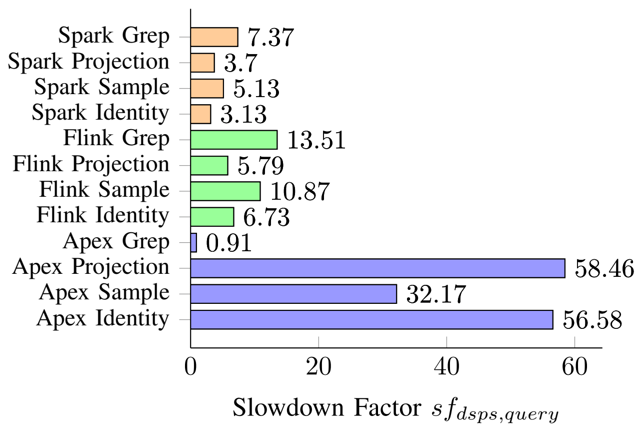

Fig. 11. Slowdown Factor for the Analyzed Systems and Queries

depicts the number of parallelisms tested, which equals two in the previously discussed benchmark scenario, particularly a parallelism factor of one as well as a parallelism of two. So in simplified terms, the ratio of average execution times for Apache Beam implementations and these using native system APIs is calculated and again averaged over parallelisms, all for a given query and data stream processing system combination.

Concretely, the average execution times for a certain system, query, and parallelism are determined, separately for the Apache Beam version as well as the implementation using native system APIs. The average execution time belonging to the Apache Beam variant is then divided by the corresponding average for the native query. That is done for every parallelism. The resulting factors for each parallelism are finally averaged by dividing their sum by the number of parallelisms.

All in all, the result tells how much slower or faster the Apache Beam version for a certain query and data stream processing system performed in the conducted measurements. That is with regard to execution times as defined in Section III-A and independent of parallelism. A result greater than one marks a slowdown, whereas a result smaller than one means that the Apache Beam implementation was faster than the one using native system APIs.

The results for the computed slowdown factors are visualized in Figure 11. It can be seen that Apache Beam implementations are slower for almost all DSPSs and queries in comparison to these developed using native system APIs.

Generally, one can recognize differences between the studied systems and queries. When looking at Apache Flink and Apache Spark, factors are similar, especially with respect to relative distinctions amongst queries. Particularly, the performance penalty for the fastest query, namely the grep query, is highest. Accordingly, it is lowest for the longest-running queries projection and identity for both systems. Overall, the performance impact on Apache Flink is slightly higher.

In contrast, the Apache Apex results show a different pattern. The highest performance impact can be seen, contrary to Apache Flink and Apache Spark, for the longest-running queries projection and identity. The query with the shortest execution time, the grep query, is overall the only query

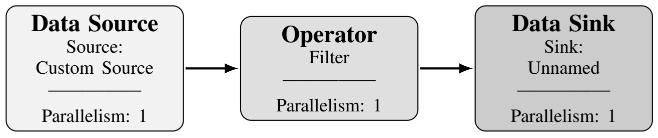

Fig. 12. Apache Flink Execution Plan for the Grep Query

where the Apache Beam implementation is even faster than the one using native system APIs according to the calculated slowdown factor. However, this speedup is very low, i.e., it is about as fast as the implementation without Apache Beam.

When looking at the absolute slowdown factors, there are also noticeable differences between Apache Apex and the other two analyzed systems. Except for the grep query slowdown, all slowdown factors are significantly higher compared to Apache Flink or Apache Spark Streaming. The slowdown factor for the projection query, e.g., is about 58 and so more than four times higher than the highest slowdown factor for either Apache Flink or Apache Spark Streaming.

Summarizing, the conducted benchmark shows that Apache Beam has a negative performance impact for almost all scenarios. Averaged over systems, the performance penalty is lowest on Apache Spark, closely followed by Apache Flink. Patterns between these two systems and executed queries are similar. The performance impact on Apache Apex is much different, meaning the impact is significantly higher in most of the cases and the previously mentioned pattern is vice-versa. So the more output or the higher the execution time on Apache Apex, the higher impact of Apache Beam on performance. The grep query running on Apache Apex is an exception to that as explained before. Except for this exceptional case, slowdown factors range from about three to almost 60. Thus, in most of the studied cases, Apache Beam has a significant influence on performance when looking at the calculated slowdown factors.

Figure 12 and Figure 13 visualize the execution plans for the grep query executed with a parallelism of one on Apache Flink, implemented without and with Apache Beam respectively. Information on execution plans are retrieved from the Apache Flink system and visualized using the Apache Flink Plan Visualizer [4]. These two plans serve as an example highlighting differences between the execution of applications developed with native APIs and those using Apache Beam.

The first execution plan depicted in Figure 12 contains three elements, a data source, an operator, and a data sink. Particularly, the source is shown as a custom source, the sink as an unnamed sink, and the operator is a filter, which fits the definition of the grep query as it basically filters data. Data is forwarded along these three elements.

The second execution plan is presented in Figure 13. It comprises seven elements in total. In particular, these elements are a data source followed by six operators. The data source at the beginning is named PTransformTranslation.UnknownRawPTransform. PTransformTranslation is a registry of familiar transforms and uniform resource names

[4]https://flink.apache.org/visualizer/

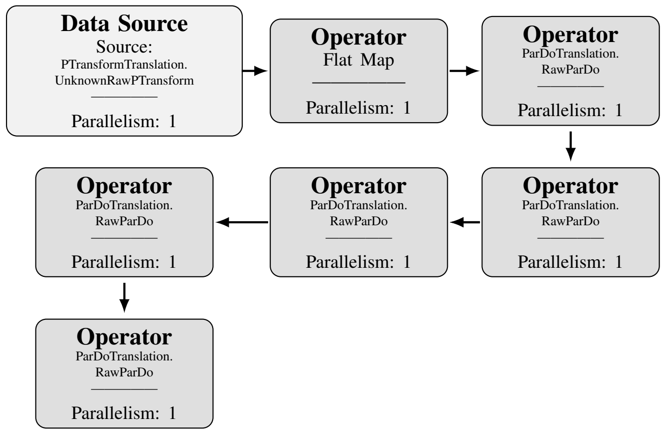

Fig. 13. Apache Flink Execution Plan for the Grep Query Implemented Using Apache Beam

(URNs) [9]. As outlined in Section II-A, a PTransform is used, e.g., for reading or writing to an external storage system [8].

The data source forwards data to the first operator, a flat map, which performs an action on each input value and produces zero or more output values. Its Apache Beam counterpart is the *read()* method of the *KafkaIO* class, which creates a *Read* PTransform. The remaining five ParDoTranslation.RawParDo operators follow the flat map. A ParDoTranslation comprises tools for working with instances of ParDo. A ParDo is one of the core transforms provided by Apache Beam and described in Section II-A. The first ParDo represents calling *withoutMetadata()* on the *Read* PTransform, which drops the Kafka metadata as it is not needed. Moreover, the method again returns a PTransform containing a PCollection of key-value pairs. The downstream operator represents the call of the *create()* method belonging to the class *Values*. This operator takes the previously created PCollection of key-value pairs and returns a PCollection containing only the values. Further downstream, the grep query logic is applied and resulting values are sent to Apache Kafka [7]–[10].

When comparing both execution plans it becomes visible that the plan for the query implemented using Apache Beam is significantly larger, i.e., it contains more elements in comparison to its counterpart. That is due to the more complex management of communication with Apache Kafka and could cause a lower performance. Both plans have in common that they start with a data source and that all elements are executed with a parallelism of one, due to the defined degree of parallelism. Moreover, a dedicated data sink is not identified for the program developed using Apache Beam. Thus, the sink must be represented as an operator.

Overall, the performance of Apache Beam applications highly depends on the runner implementations. Effort put into this development is likely to vary between systems. The closeness of the DSPSs' programming model to the underlying concepts of Apache Beam also impacts the application execution. Further details, e.g., with respect to the concrete impact of the additional operator, could be uncovered through profiling applications. However, all measurements are a snapshot in time and results may differ with different versions of Apache Beam, other DSPSs versions, or alternative system configurations. Moreover, changed workloads characteristics might also influence performance results.

IV. Related Work

For supporting Apache Beam, a system has to implement a so-called runner. The development of the runner for IBM Streams is described in [49]. Next to highlighting three implemented optimizations with regard to the IBM Streams runner, performance evaluations between IBM Streams, Apache Flink, and Apache Spark are presented.

Besides abstraction layers such as Apache Beam that allow for developing data stream processing applications using a programming language such as Java, there is the idea of leveraging or extending SQL to be able to do that.

Continuous Query Language (CQL) [50] is a comprehensive approach for such an SQL extension. To be more concrete, CQL is a SQL-based language for defining continuous queries over data streams as well as updatable relations. It is not only a concept but also integrated into the STREAM [51] system, a data stream management system developed at Stanford University. Next to describing the CQL implementation in STREAM, the semantics of CQL are outlined and a comparison to other languages is included in the linked paper.

Apache Calcite [52] is another approach that was developed more recently. It is a framework that comprises various functionalities, e.g., with respect to query processing, query optimization, and query language support, which is the relevant aspect in this context. Amongst others, the architecture of Apache Calcite is presented in the mentioned work as well as developed SQL extensions for different areas such as semi-structured data or geospatial queries. Another depicted example is the extensions for data stream processing queries that are called STREAM extensions. They are inspired by the mentioned CQL and also explained on their website [53]. However, not all the presented concepts have been implemented yet [53]. A few DSPSs already integrate Apache Calcite, Apache Apex and Apache Flink being two of them [52].

Jain et al. [54] discuss the differences of two SQL-based languages for defining streaming queries, particularly Oracle Continuous Query Language [55] and StreamBase StreamSQL [56]. Moreover, an approach for unifying both languages is proposed. However, they highlight that for achieving a complete standard, further challenges needs to be tackled.

Besides, there are more SQL extensions for stream processing scenarios developed for certain systems, e.g., streaming SQL for Apache Kafka called KSQL [57], Continuous Computation Language (CCL) that is the extended SQL used in SAP HANA Smart Data Streaming [58], or SamzaSQL [59] as extended SQL for the DSPS Samza [60].

With respect to benchmarking DSPSs in general, the Linear Road benchmark by Arasu et al. [61] is a very well-known work. It is an application benchmark that provides a benchmarking toolkit. This toolkit consists of a data generator, a data sender, and a result validator. The underlying idea of the benchmark is a variable tolling system for a metropolitan area. This area covers multiple expressways with moving vehicles. The amount of accumulated tolls depends on various aspects concerning the traffic situation.

The mentioned data sender emits data to the DSPS, which is mostly car position reports. Depending on the overall situation on the expressways, car position reports may require the DSPS to create an output or not. Next to car position reports, the remaining input data represent an explicit query which always requires an answer. Linear Road defines four distinct queries, whereas the query lastly presented in [61] was skipped in the two presented implementations due to complexity reasons.

The benchmark result for a system is summarized as a so called L-rating. This metric defined by Linear Road expresses how many expressways the system could handle while meeting the defined response time requirements for each query. A higher number of expressways corresponds to a higher data input rate for the SUT. When generating data, the amount of expressways can be configured.s The Linear Road benchmark was applied to the DSPS Aurora [62] and a commercial relational database. Results are presented in the paper.

Another related benchmark is the already mentioned StreamBench [43]. It aims at benchmarking distributed DSPSs. Regarding its category in the area of performance benchmarks it can be viewed as a microbenchmark, i.e., it measures atomic operations such as a projection rather than more complex applications as in, e.g., Linear Road.

The seven queries defined by StreamBench partly contain a single computational step and partly comprise multiple computational steps. Three queries require to keep a state in order to calculate correct results while the remaining four queries are stateless. One query uses numerical data as input while the others process textual data. With respect to the benchmark architecture, StreamBench makes use of a message broker, specifically Apache Kafka, for decoupling data generation and consumption. That is different to Linear Road but similar to the benchmark architecture proposed in Section III-A. As part of the evaluation section, the DSPSs Apache Storm [63] and Apache Spark Streaming are compared by applying StreamBench.

NEXMark [64] is a benchmark aiming to benchmark streaming queries in the context of an online auction system. It seems the benchmark was never finished as the website still states that it is "work in progress" [64] and there is only a draft version of a paper that was published a couple of years ago [65]. However, in the context of Apache Beam this benchmark was used as inspiration and foundation for a NEXMark-based benchmark suite [66]. This suite extends the eight NEXMark queries by five additional ones. A complete implementation of all queries for all runners is work in progress according to [66].

The work presented in [67] compares Apache Flink and Apache Spark. The conducted measurements include different queries, a grep query being one of them. One focus area that is analyzed is the scaling behavior with regard to different numbers of nodes in the cluster. However, studying both systems from a data stream processing point of view is out of scope in the performed measurements.

Lopez et al. [68] compare Apache Storm, Apache Flink, and Apache Spark Streaming in their paper. Besides describing the architecture of these three systems, the performance is studied in a network traffic analysis scenario. Additionally, the behavior in case of a node failure is investigated.

V. Conclusion and Future Work

This paper describes the characteristics of three state-of-the-art DSPSs, particularly Apache Apex, Apache Flink, and Apache Spark Streaming, as well as the abstraction layer Apache Beam for implementing stream processing programs.

To study the performance impact of Apache Beam, we propose a lightweight benchmark architecture that uses a recognized workload and a novel approach for measuring execution times using Apache Kafka. All benchmark implementations are provided in order to ensure reproducibility.

Our benchmark results show that Apache Beam has a noticeable impact on the performance of DSPSs in almost all cases. Programs developed using Apache Beam suffered from a slowdown of up to a factor of 58 in the worst case. At the same time, there is one scenario where the query developed using Apache Beam is about as fast as its counterparts using the APIs of the corresponding DSPS. However, for most scenarios we observed a slowdown of at least a factor three.

The results lead to two major conclusions. Firstly, using Apache Beam as an abstraction layer for application development comes at a cost in terms of runtime performance. Secondly, the results of benchmarking different DSPSs using a program developed with Apache Beam are not likely to represent the performance differences that are to be expected from a benchmark with programs developed using native system APIs. While using Apache Beam certainly provides a greater flexibility to switch underlying DSPSs with relatively low effort, one needs to be aware of the fact that this advantage comes with a negative impact on performance. This performance penalty varies among systems and applications and is currently unpredictable.

Future work in this area involves studying the reasons for performance differences in greater detail. Particularly, the applications could be profiled in order to see how much time is spent in which part of the execution plans and thus, to identify possible performance bottlenecks. Finding potential reasons for the comparatively large performance penalties when employing Apache Apex represents another interesting area for future work. In the best case, it is possible to identify factors that influence the performance penalty applications suffer from and make them predictable. Additionally, measurements can be extended with respect to various aspects such as the number of studied systems or query complexity as well as scaling, parallelism, or fault-tolerance behaviors. Furthermore, upcoming Apache Beam versions and other abstraction layers can be compared against the presented results to supplement the initial overview presented here.

References

[1] M. Stonebraker and U. Çetintemel, “”one size fits all”: An idea whose time has come and gone (abstract),” in *Proc. International Conference on Data Engineering, ICDE*, 2005, pp. 2–11. [Online]. Available: https://doi.org/10.1109/ICDE.2005.1

[2] “Apache Beam Overview,” https://beam.apache.org/get-started/beam-overview/, accessed: 2018-10-30.

[3] M. Lorenz, J. Rudolph, G. Hesse, M. Uflacker, and H. Plattner, “Object-Relational Mapping Revisited - A Quantitative Study on the Impact of Database Technology on O/R Mapping Strategies,” in *Hawaii International Conference on System Sciences, HICSS*, 2017.

[4] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, “Apache Flink™: Stream and Batch Processing in a Single Engine,” *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015. [Online]. Available: http://sites.computer.org/debull/A15dec/p28.pdf

[5] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, “Apache Spark: A Unified Engine for Big Data Processing,” *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016. [Online]. Available: http://doi.acm.org/10.1145/2934664

[6] “Apache Apex,” https://apex.apache.org/docs/apex/, accessed: 2018-09-11.

[7] “Apache Beam,” https://github.com/apache/beam, accessed: 2018-08-17.

[8] “Apache Beam Programming Guide,” https://beam.apache.org/documentation/programming-guide/, accessed: 2018-10-18.

[9] “Runner Authoring Guide,” https://github.com/apache/beam/blob/master/website/src/contribute/runner-guide.md, accessed: 2018-10-18.

[10] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, “The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing,” *PVLDB*, vol. 8, no. 12, pp. 1792–1803, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1792-Akidau.pdf

[11] “Beam Capability Matrix,” https://beam.apache.org/documentation/runners/capability-matrix/#cap-summary-what, accessed: 2018-09-19.

[12] “Apache Gearpump,” https://gearpump.apache.org/overview.html, accessed: 2018-10-15.

[13] “MapReduce Tutorial,” https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html, accessed: 2018-10-15.

[14] “What is Samza?” https://samza.apache.org, accessed: 2018-10-15.

[15] “Alibaba JStorm,” http://jstorm.io, accessed: 2018-10-15.

[16] “IBM Streams,” https://www.ibm.com/de-en/marketplace/stream-computing, accessed: 2018-10-15.

[17] “CLOUD DATAFLOW - Simplified stream and batch data processing, with equal reliability and expressiveness,” https://cloud.google.com/dataflow/, accessed: 2018-08-17.

[18] “Cloud Dataflow, Apache Beam and you,” https://cloud.google.com/blog/products/gcp/cloud-dataflow-apache-beam-and-you, accessed: 2018-10-15.

[19] G. Hesse and M. Lorenz, “Conceptual Survey on Data Stream Processing Systems,” in *IEEE International Conference on Parallel and Distributed Systems, ICPADS*, 2015, pp. 797–802. [Online]. Available: https://doi.org/10.1109/ICPADS.2015.106

[20] “Flink - Distributed Runtime Environment,” https://ci.apache.org/projects/flink/flink-docs-master/concepts/runtime.html, accessed: 2018-09-27.

[21] “Spark Streaming Programming Guide,” https://spark.apache.org/docs/latest/streaming-programming-guide.html, accessed: 2018-09-26.

[22] “Apache Spark - Cluster Mode Overview,” https://spark.apache.org/docs/2.3.1/cluster-overview.html, accessed: 2018-09-10.

[23] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center,” in *Proc. USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2011. [Online]. Available: https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center

[24] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O’Malley, S. Radia, B. Reed, and E. Baldeschwieler, “Apache Hadoop YARN: Yet Another Resource Negotiator,” in *ACM Symposium on Cloud Computing, SOCC*, 2013, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633

[25] E. A. Brewer, “Kubernetes and the Path to Cloud Native,” in *Proc. ACM Symposium on Cloud Computing, SoCC*, 2015, p. 167. [Online]. Available: http://doi.acm.org/10.1145/2806777.2809955

[26] X. Lu, M. Wasi-ur-Rahman, N. S. Islam, D. Shankar, and D. K. Panda, “Accelerating Spark with RDMA for Big Data Processing: Early Experiences,” in *IEEE Annual Symposium on High-Performance Interconnects, HOTI*, 2014, pp. 9–16. [Online]. Available: https://doi.org/10.1109/HOTI.2014.15

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing,” in *Proc. USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2012, pp. 15–28. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[28] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized Streams: Fault-Tolerant Streaming Computation at Scale,” in *ACM SIGOPS Symposium on Operating Systems Principles, SOSP*, 2013, pp. 423–438. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522737

[29] “Apache Hadoop,” https://hadoop.apache.org, accessed: 2018-09-11.

[30] “HDFS Architecture,” http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, accessed: 2018-09-11.

[31] M. Bhandarkar, “AdBench: A Complete Benchmark for Modern Data Pipelines,” in *TPC Technology Conference, TPCTC*, 2016, pp. 107–120. [Online]. Available: https://doi.org/10.1007/978-3-319-54334-5_8

[32] T. Dunning and E. Friedman, *Streaming Architecture: New Designs Using Apache Kafka and MapR Streams*. O’Reilly Media, 2016. [Online]. Available: https://books.google.de/books?id=EU8kDAAAQBAJ

[33] J. Kreps, N. Narkhede, and J. Rao, “Kafka: a Distributed Messaging System for Log Processing,” in *Proc. International Workshop on Networking Meets Databases, NetDB*, 2011, pp. 1–7.

[34] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. C. Murthy, and C. Curino, “Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications,” in *Proc. International Conference on Management of Data, ACM SIGMOD*, 2015, pp. 1357–1369. [Online]. Available: http://doi.acm.org/10.1145/2723372.2742790

[35] “Flink - YARN Setup,” https://ci.apache.org/projects/flink/flink-docs-master/ops/deployment/yarn_setup.html, accessed: 2018-09-23.

[36] “Running Spark on YARN,” https://spark.apache.org/docs/latest/running-on-yarn.html, accessed: 2018-09-23.

[37] “Apache Hadoop YARN,” https://hadoop.apache.org/docs/r3.0.3/hadoop-yarn/hadoop-yarn-site/YARN.html, accessed: 2018-09-25.

[38] “Apache Apex Documentation - Application Developer Guide,” http://apex.apache.org/docs/apex-3.7/application_development/, accessed: 2018-09-26.

[39] “Apache Flink,” https://github.com/apache/flink, accessed: 2018-10-21.

[40] “Mirror of Apache Apex core,” https://github.com/apache/apex-core, accessed: 2018-10-21.

[41] “Mirror of Apache Spark,” https://github.com/apache/spark, accessed: 2018-10-21.

[42] “AOL Search Query Logs,” http://www.researchpipeline.com/mediawiki/index.php?title=AOL_Search_Query_Logs, accessed: 2018-09-28.

[43] R. Lu, G. Wu, B. Xie, and J. Hu, “StreamBench: Towards Benchmarking Modern Distributed Stream Computing Frameworks,” in *Proc. IEEE/ACM International Conference on Utility and Cloud Computing, UCC*, 2014, pp. 69–78. [Online]. Available: https://doi.org/10.1109/UCC.2014.15

[44] “Documentation - Kafka 0.10.2 Documentation,” https://kafka.apache.org/documentation/, accessed: 2017-04-24.

[45] “Flink - Command-Line Interface,” https://ci.apache.org/projects/flink/flink-docs-release-1.6/ops/cli.html, accessed: 2018-09-27.

[46] “Spark Configuration,” https://spark.apache.org/docs/latest/configuration.html#spark-properties, accessed: 2018-09-27.

[47] “Interface Context.OperatorContext,” https://ci.apache.org/projects/apex-core/apex-core-javadoc-release-3.6/com/datatorrent/api/Context.OperatorContext.html, accessed: 2018-10-29.

[48] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, “Benchmarking Distributed Stream Data Processing Systems,” in *IEEE International Conference on Data Engineering, ICDE*, 2018, pp. 1507–1518. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ICDE.2018.00169

[49] S. Li, P. Gerver, J. Macmillan, D. Debrunner, W. Marshall, and K. Wu, "Challenges and Experiences in Building an Efficient Apache Beam Runner For IBM Streams," *PVLDB*, vol. 11, no. 12, pp. 1742–1754, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1742-li.pdf

[50] 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, 2006. [Online]. Available: https://doi.org/10.1007/s00778-004-0147-z

[51] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom, "STREAM: The Stanford Stream Data Manager," *IEEE Data Eng. Bull.*, vol. 26, no. 1, pp. 19–26, 2003. [Online]. Available: http://sites.computer.org/debull/A03mar/paper.ps

[52] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire, "Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources," in *Proc. International Conference on Management of Data, ACM SIGMOD*, 2018, pp. 221–230. [Online]. Available: http://doi.acm.org/10.1145/3183713.3190662

[53] "Streaming," https://calcite.apache.org/docs/stream.html, accessed: 2018-10-19.

[54] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. B. Zdonik, "Towards a Streaming SQL Standard," *PVLDB*, vol. 1, no. 2, pp. 1379–1390, 2008. [Online]. Available: http://www.vldb.org/pvldb/1/1454179.pdf

[55] "Oracle® CEP CQL Language Reference 11g Release 1 (11.1.1)," https://docs.oracle.com/cd/E16764_01/doc.1111/e12048/intro.htm, accessed: 2018-10-19.

[56] "StreamSQL Overview," https://docs.tibco.com/pub/sb-lv/2.1.8/doc/html/streamsql/ssql-intro.html, accessed: 2018-10-19.

[57] "KSQL and Kafka Streams," https://docs.confluent.io/current/streams-ksql.html, accessed: 2018-10-19.

[58] "SAP HANA Smart Data Streaming: Developer Guide," https://help.sap.com/doc/25fc8560420d4d5099d6df02f7cbff9e/1.0.12/en-US/streaming_developer_guide.pdf, accessed: 2018-10-19.

[59] M. Pathirage, J. Hyde, Y. Pan, and B. Plale, "SamzaSQL: Scalable Fast Data Management with Streaming SQL," in *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS*, 2016, pp. 1627–1636. [Online]. Available: https://doi.org/10.1109/IPDPSW.2016.141

[60] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: Stateful Scalable Stream Processing at LinkedIn," *PVLDB*, vol. 10, no. 12, pp. 1634–1645, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol10/p1634-noghabi.pdf

[61] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear Road: A Stream Data Management Benchmark," in *(e)Proc. International Conference on Very Large Data Bases*, 2004, pp. 480–491. [Online]. Available: http://www.vldb.org/conf/2004/RS12P1.PDF

[62] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik, "Aurora: a new model and architecture for data stream management," *VLDB J.*, vol. 12, no. 2, pp. 120–139, 2003. [Online]. Available: https://doi.org/10.1007/s00778-003-0095-z

[63] "Apache Storm," http://storm.apache.org, accessed: 2018-10-23.

[64] "NEXMark Benchmark," http://datalab.cs.pdx.edu/niagara/NEXMark/, accessed: 2018-10-20.

[65] P. Tucker, K. Tufte, V. Papadimos, and D. Maier, "NEXMark – A Benchmark for Queries over Data Streams DRAFT," http://datalab.cs.pdx.edu/niagara/pstream/nexmark.pdf, accessed: 2018-10-20.

[66] "Nexmark benchmark suite," https://beam.apache.org/documentation/sdks/java/nexmark/, accessed: 2018-10-20.

[67] O. Marcu, A. Costan, G. Antoniu, and M. S. Pérez-Hernández, "Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks," in *IEEE International Conference on Cluster Computing, CLUSTER*, 2016, pp. 433–442. [Online]. Available: https://doi.org/10.1109/CLUSTER.2016.22

[68] M. A. Lopez, A. G. P. Lobato, and O. C. M. B. Duarte, "A Performance Comparison of Open-Source Stream Processing Platforms," in *IEEE Global Communications Conference, GLOBECOM*, 2016, pp. 1–6. [Online]. Available: https://doi.org/10.1109/GLOCOM.2016.7841533