Performance Evaluation of Stream Data Processing Systems

ABSTRACT

Over the past years, Stream data processing is gaining compelling attention both in industry and in academia due to its wide range of applications in various use-cases. To fulfil the need for efficient and high performing Big data analytics, numerous open source stream data processing systems were developed. Processing data with high throughput while retaining low latency is key performance indicator for such systems. In this paper, we propose a benchmarking system to evaluate the performance of stream data processing systems, Storm, Spark and Flink in terms of indicators shown above. More specifically, the latency of windowed aggregation and windowed join operators is evaluated jointly with the throughput of a system.

1. INTRODUCTION

Streaming Data Processing (SDP) has been gaining significant attention due to its wide range of uses in Big data analytics. The main reason is that processing big volumes of data periodically is not enough anymore and data has to be processed fast to enable fast adaptation and reaction to changed conditions. Several engines are widely adopted and supported by open source community, such as Apache Storm [18], Apache Spark [22] , Apache Flink [3]. There are definitely, different ways to process stream data before it is persisted in database. For example, while Storm and Flink provide record-by-record stream processing, Spark has a different approach to collect events together and process all in a series of mini-batches.

Determining the right Stream Data Processing System (SDPS) for use case is key to get best performance. Latency and throughput are two main KPIs to measure the performance of SDPS. Latency is the time required to calculate the final result. Because the stream is theoretically infinite and therefore there is no final result, defining latency in SDPS can be tedious. Throughput on the other hand determines the number of successful calculations per unit time. In stream data processing, it is number of tuples source operator can ingest for further process per unit time.

In this work we propose the benchmark system to assess the performance of three major, open source and community driven streaming engines, being Apache Storm, Apache Spark and Apache Flink. Throughout the tests, we use latency and throughput as the major performance indicators. The benchmarks are designed on top of real world use cases, specifically the ones from *Rovio Entertainment*. The first use case is finding an average price over numerous streaming data sources by dividing them into sliding windows. Here, we aim to evaluate the performance of partitioned windowed aggregations. The second use case is comparison of prices which originated in same geo locations by dividing them in sliding windows. Here, we focus on performance of partitioned windowed joins.

There are numerous works that do benchmarks among different SDPSs with specific Key Performance Indicators (KPI) [15, 5]. However, researchers in previous works, either don't clearly specify the semantics of KPIs or the determined KPIs are affecting system's actual performance. That is, the definition and calculation of KPIs should be clear and those should be kept separate from system under test. Moreover, keeping benchmark design simple and with possible less systems, makes the results less biased and reproducible. For example, if one of underlying benchmarking systems is a bottleneck for measurements, then the actual results of system under test, can be interpreted wrongly.

TODO: STATEFUL AND STATELESS OPERATORS LATENCY

In this paper, we overcome all of possible bottlenecks listed above. Our system is generic, has simple design with clear semantics and can be applied to any streaming system. It consists of predefined number of data generators and actual System Under Test (SUT). The link between SDPS and data generator is a socket. To overcome with extra latency of persistent queues, we removed persistent queue as a connector and used sockets. The main intuition is to simulate an environment to get actual latency of a tuple with minimum affections of other factors. Each tuple has its timestamp field which indicates the time it was generated by data generator. Rather than connecting the SDPS to directly data generator, we put a queue between them as the SDPS may not ingest all generated data. Waiting in queue increases the latency of a tuple, so the sooner the system pulls data from queue, the lower the latency will be. The throughput on the other hand, is calculated by number of pulls of SDPS to queue in a unit time.

The main contributions of this paper are listed above:

- In this paper, we introduce the first mechanism to measure the latency in stateful operators is SDPS. We applied the proposed method with partitioned windowed aggregation and partitioned windowed join use cases.
- The proposed benchmarking system measures the throughput of a system and the latency of an operator out of the box. That is, the throughput is associated with the system and therefore we measure throughput outside the system, in data generator module. Moreover, the latency is linked with operator, therefore, we assess it outside of operator. The main goal is, to eliminate side factors affecting the measurements.
- Because we are testing SDPSs, the backpressure is an inportant feature for such systems and should be considered. We introduce the first solution to check the SUT's upper limit of throughput taking into consideration backpressure and system initialization delays. If the system cannot sustain the data rate, depending on user's configuration, data-queue module will tolerate it for some time.
- We test the SDPSs with different configurations and cluster size and provide the analysis of the experimental results.

The remainder of paper is organised as follows. In Section ... TODO

2. RELATED WORK

The main concepts and methodologies used in benchmarking SDPS were inherited from big data benchmarks. Now with emerging next generation stream data processing engines, batch processing is seen as a special case of stream processing where the data size is bounded. Huang et.al. propose HiBench, the first benchmark suite for evaluation and characterisation of Hadoop [20]. Authors conduct wide range of experiments from micro-benchmarks to machine learning algorithms [8]. Covering end-to-end big data benchmark with all major characteristics such as three Vs in the lifecycle of big data systems is the main intuition behind BigBench [7]. Wang et.al. introduce BigDataBench, a big data benchmark suite for Internet Services, characterising the 19 big data benchmarks covering broad application scenarios and diverse and representative data sets [19].

Benchmarks on SDPS are Researchers from Yahoo Inc. have done benchmarks on streaming systems to measure latency and throughput [5]. They used Apache Kafka [9] and Redis [4] for data fetching and storage. Later on, on the other hand, Data Artisans, showed those systems actually being a bottleneck for SUT's performance [6]. The extensive analysis of the differences between Apache Spark and Apache Flink in terms of batch processing is done by correlating the operators execution plan with the resource utilization and the parameter configuration [13]. In another benchmark, authors compare the performances of Apache Spark and Apache Flink to provide clear, easy and reproducible configurations that can be validated by community in clouds [15]. Authors conducted benchmarks to assess the fault tolerance and throughput efficiency for open source stream data processing engines [11]. In another benchmark, authors motivate IoT as being main application area for SDPS and perform common tasks in particular are with different stream data processing engines and evaluate performance [17]. One of the pioneers in SDPS benchmarks, developed framework StreamBench analysing the current standards in streaming benchmarks and proposing a solution to measure throughput, latency considering the fault tolerance of SUT [12]. Authors put a mediator system between data generator module and SUT and define the latency as the average time span from the arrival of a record till the end of processing of the record. LinearRoad benchmarking framework was presented by Arasu et al. to measure performance of standalone stream data management systems such as Aurora [1] by simulating a scenario of toll system for motor vehicle expressways. Several stream processing systems implement their own benchmarks to assess the performance [14, 16, 22]. SparkBench is a benchmarking framework to evaluate machine learning, graph computation, SQL query and streaming application on top of Apache Spark [10].

3. PRELIMINARY AND BACKGROUND

In this section we provide preliminary and background information about the stream data processing engines used throughout this paper. Initially, the general information about the working principles of particular system is given. Afterwards, we provide more use case specific info for each system. Because we test engines' partitioned windowed join and aggregation operators, basic semantics of particular operators , computational model and back-pressure mechanism are analysed.

3.1 Apache Storm

Apache Storm is a distributed stream processing computation framework which was open sourced after being acquired by Twitter.

Computational model Storm operates on tuple streams and provides record-by-record stream processing. It supports at-least-once processing (when there are failures events are replayed) mechanism and guarantees all tuples to be processed. Storm also supports exactly-once semantics with its Trident abstraction. The core of Storm data processing is a computational topology which consists of spouts and bolts. Spouts are source operators whereas bolts are processing and sink operators. Because Storm topology is DAG structured, where the edges are stream tuples and vertices are operators (bolts and spouts), when a spout or bold emits a tuple, the ones that are subscribed to particular spout or bolt receive input. Storm's parallelism model is based on *tasks*. Each task runs in parallel and by default single thread is allocated per task.

Storm's lower level API's provide little support for managing the memory and state. Therefore, choosing the right data structure for state management, utilizing memory usage efficiently my making computations incrementally is up to the user. memory management. Storm supports cashing and batching the state transition. However, the efficiency of particular operation degrades as the size of state grows. Storm supports back-pressure.

Windowing. Storm has built-in support for windowed calculations. That is, partitioned windowed joins and aggregations are supported internally. Although the information of expired, new arrived and total tuples within window is provided through APIs, the state management and making

computations incremental should be done manually. Storm supports processing and event-time windows with sliding and tumbling window features. Processing time windows include time and count based semantics. For event-time windows, tuples should have separate timestamp field so that the engine can create periodic watermarks. One of the downsides of Storm's relying heavily on ackers, is that tuples can be acked once they completely flush out of window. This can be an issue specially, on windows with big length and small slide.

3.2 Apache Spark

Apache Spark is an open source data processing engine, originally developed at the University of California, Berkeley.

Computational model Spark internally is batch processing engine. It handles the stream processing by micro-batches. As can be seen from Figure **??**, Spark Streaming resides at the intersection of batch and stream processing. Resilient Distributed Dataset (RDD) is a fault tolerant abstraction which enables in memory parallel computation in distributed cluster environment [21]. Unlike Storm and Flink, which support one record at a time, Spark Streaming inherits its architecture from batch processing which support processing records in micro-batches.

One of Spark's features is that it supports lazy evaluation and tries to limit the amount of work it has to do. This enables the engine to run more efficiently. Spark also supports DAG based execution graph which works implementing stage-oriented scheduling. Unlike from Flink and Storm, which also work based on DAG exetuion graph, Spark computing unit in graph is data set rather than streaming tuple and each vertex in graph is a stage rather than operators. RDDs are guaranteed to be processed in order in a single DStream. However, the order guarantee within RDD is not provided since each RDD is processed in parallel.

Spark Streaming has improved significantly its memory management in recent releases. The memory is shared between execution and storage. This unified memory management supports dynamic memory management between the two modules. Moreover, Spark supports dynamic memory management throughout the tasks and within operators of each task.

Windowing Spark Streaming has a built-in support for windowed calculations. Processing time windows with sliding and tumbling versions are supported in Spark. The operations done with sliding windows, are internally incrementalized transparent to the user. However, choosing the length batch interval can affect the window based analytics. Firstly, the latency and response time of windowed analytics is strongly replying on batch interval. Secondly, supporting only processing time windowed analytics, can still be a bottleneck in some use cases. Spark supports back pressure which is very useful in windowed calculations. The window size must be a multiple of the batch interval, because window keeps the particular number of batches until it is purged.

3.3 Apache Flink

Apache Flink which was started off as an academic open source project (Stratosphere [2]) in Technical University of Berlin.

Computational model Similar to Storm, The computational model of Flink could be described as a graph consists of transformations as vertices and stream tuples as edges.

Windowing

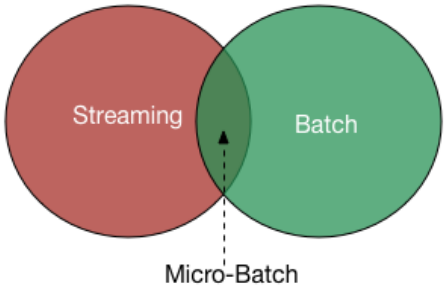


Figure 1: Conceptual view of micro-batching

4. CHALLENGES

There are several challenges to be taken into consideration while designing a benchmark for SDPSs. In this section we analyse the challenges and provide solutions.

Simple is beautiful. The first challenge is to design a simple system, because simple is beautiful. As the number of systems included inside benchmark increases, the complexity escalades at the same time. One of downsides of the complex system is, difficult to determine the bottleneck. For example, to test the stream data processing engine, the data generator component is essential, to simulate the real life scenarios. Figure 2 shows possible three cases to link data generator and SUT. The simplest design would be connecting the SDPS directly to data generators as shown in Figure 2a. Although this is perfectly acceptable case, it confronts with real life use cases. That is, in real life, the stream data processing engines, do not connect to pull based data sources unless there is a specific system design. Usually, SDPSs pull data from the distributed message queues which reside between data generators and SUT. One bottleneck on this option is throughput which is bounded by the maximum throughput of message queueing system. We selected the third option which stands between the first two. As can be seen from Figure 2c, we embedded the queues as a separate module in data generators. In this way, the throughput is bounded only by network bandwidth, the system works more efficient as there are no se/deserialisation overheads and finally it has a simple design. Moreover, while measuring the performance of SUT, using extra systems for checkpointing the current state of measurements or for saving the intermediate evaluation results can affect the SUT's performance.

Keep driver and test unit as separate as possible. The second challenge is to isolate the benchmark driver and SUT as much as possible. For example, in benchmarking SDPSs, it is common to measure the throughput inside the SUT. Because both computations can affect each other the results can be biased which we discuss below. We solved this problem by categorising the test unit and pointing measurements accordingly. The first evaluation is throughput. Throughput is associated with the system. So, we kept the

throughput assessment outside the SUT, inside data generator module. The second evaluation is latency. The latency is linked with an operator inside system. So, we kept all latency measurements outside the particular operator.

Avoid misleading tests. The third challenge is abstain from biased test results and keep the evaluation semantics clear. One example for this is, throughput measurement. In the previous SDPS benchmarks, the throughput of a SUT is measured by either taking quantiles over test time, or showing max, min and average assessments. From user's perspective on the other hand, the system's throughput is the one with upper bound that it can sustain the data processing. When conducting the experiments with stream data processing engines, back pressure is another factor affecting the throughput, that should be taken into consideration. Another example for this challenge is latency measurement. In the previous SDPS benchmarks, the latency of an operator is measured by taking difference of tuple's ingestion timestamp to system and output timestamp from system. However, from user's perspective, the start timestamp of a tuple is once it is created and latency of particular tuple should be calculated by taking into consideration its event timestamp. We solve the first issue by measuring the SUT's max throughput that it can sustain under back pressure with a given amount of input. We developed the mechanism for handing the back pressure. For the second issue, we measure the latency of a tuple by differentiating the time when output is done and its event timestamp.

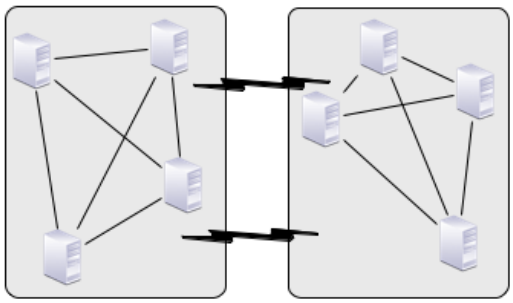
Latency of stateful operator. The fourth challenge is measuring the latency of stateful operator. Up to this point, the related works in literature either concentrated on stateless operators or evaluated the latency of stateful operators by checkpointing to external systems like lightweight distributed databases. As we discussed above, this approach can be a bottleneck in some cases. In this paper, we provide a solution for this problem without any external system.

5. BENCHMARK SYSTEM DESIGN

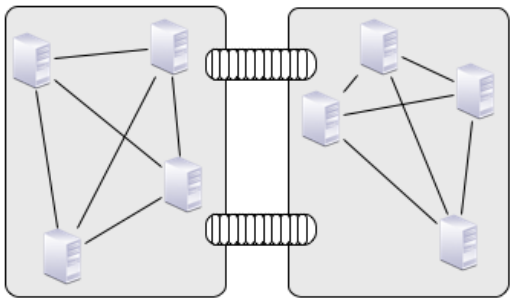
We keep overall design of benchmark simple and akin to the systems under to be tested. The stream data processing system connects to predefined number of socket input data sources. Each input data source has its own data generator and queue where tuples are put after generation. When the stream data processing system pulls data from socket server, serving thread sends the data from queue rather than directly from data generator.

Figure 3 shows the overall intuition of the benchmark system.Firstly, the data generator, at the left side of the figure, spawns the tuples and appends the current timestamp as a separate field. Secondly, the generated tuples wait in queue until the stream data processing system pulls them. Queue is based on FIFO semantics. Thirdly, the stream data processing system combines the tuples from different sockets by uniting streams. Depending on the nature of operator to be tested, there can be several unions of streams. For example, windowed aggregation is single input stream operator but join operator accepts two streams as an input. Fourthly, the operator to be tested computes the data in windows. In our case, we test windowed aggregation and windowed join operators. Finally, the next operator calculates the latency of a tuple by subtracting event timestamp from current timestamp.

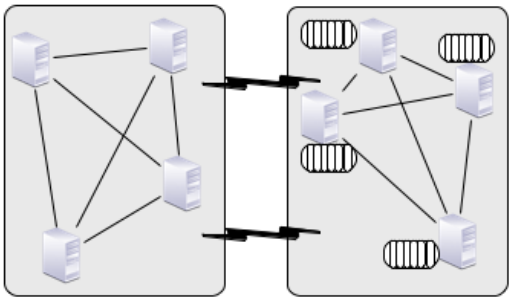
Figure 2: Different system designs to link data generator and SUT.



(a) Without message queue



(b) With message queue



(c) Partial message queue

Proper handling tuples' timestamp fields while joining or aggregating is crucial. In this work, *merging* of tuples' timestamp fields is done by selecting *maximum* over them. That is, latest arrived tuple's timestamp is transferred to the new tuple as a result of aggregation or join. Equation 3 defines this logic formally.

Here $t \in T$ is a stream tuple, $t[k]$ is k^{th} field of particular data point and $\equiv$ means equivalent in terms of type. For aggregation function $|T|$, the size of a set is not bounded, whereas for join function it is bounded by two, being $|T| = 2$.

5.1 Key Performance Indicators

To measure the performance of a system, we connected max 16 data generators to system under test with order of 1,2,4,8 and 16 as increasing further does not increase the overall throughput significantly. We call the tests with related workloads as $1x$, $2x$, $4x$, $8x$ and $16x$. Moreover, the configuration of each data generator must be the same. Configuration includes parameters such as overall input size, generation speed, socket port and etc. Equation 1 defines this formally.

$$\begin{gathered}\textbf{Let } d_i^{c_i} \in D \\ \textbf{then}, |D| \leftarrow S \\ \textbf{and } c_1 = c_2 \ ... = c_n, \forall n \in S = 1, 2, 4, 8, 16\end{gathered} \tag{1}$$

5.1.1 Throughput

Throughput of a system is calculated as summing the consumer throughput of all queues. If the system pulls the data from all queues with same rate then, the throughput of all queues will be the same. It is crucial for experiments to adjust the throughput of producer to be approximately the same as the one of consumer. The semantics behind adjustability of consumer and producer throughputs must be clear and applicable to stream data processing system designs as well. Here producer is data generator and consumer is stream data processing system.

To overcome this problem, we propose simple mechanism to adjust upper limit of producer throughput to consumer throughput. Figure 4 shows the basic intuition behind this. There are three borderlines to be taken into consideration. The green portion of queue S_a, is acceptable sustainability. That is, if consumer can consume elements with no more than S_a elements left in queue, then this is acceptable and consumer and producer throughputs are said to be adjustable. However, the engines we test have back-pressure mechanism that can limit consumer pulls from queue. So, if consumer is slower than producer, we let S_b elements to be buffered additionally. This behaviour, on the other hand, is allowed not *limited time*(the definition of *limited time* is below). If after *limited time* the queue size is not within boundaries of S_a then the application quits, indicating the consumer cannot sustain current data generation rate. Finally, if queue size increases within boundaries of $S_a + S_b$ then there is no need to wait for back-pressure and application quits immediately.

To evaluate the *tolerance limit* in queue for possible back-pressure, we use rounds.

$$\begin{aligned} round_{length} &\leftarrow S_a \\ round_{max} &\leftarrow \frac{S_b}{S_a} \end{aligned} \tag{2}$$

$round_{length}$ is the check period in queue. That is, after every $round_{length}$ tuple put into the queue, the check has been done of queue size. If queue size is between S_a and S_b then, the application is not quit immediately, but instead checked $round_{max}$ times. The calculation of these variables are shown in Equation 2.

5.1.2 Latency

$$\begin{gathered}\textbf{Let} f : \{t | t \in T\} \rightarrow t', \\ \textbf{then}, \exists k \ s.t. \ t[k] \equiv t'[k] \ \forall t \in T \\ \textbf{and } t'[k] \leftarrow \text{argmax}\{t[k] \mid t \in T\}\end{gathered} \tag{3}$$

$$\begin{gathered}\textbf{Let } t_i \in I, t_o \in O \\ \textbf{then } Latency_{\ t_o} = time_{now} - t_o[k] \\ s.t. \ f : \{t_i \ | t_i \in I\} \rightarrow \{t_o \ | t_o \in O\}\end{gathered}$$

The latency of each tuple is calculated

6. EVALUATION

6.1 Configuration

spark batch size

6.2 Keyed Windowed Aggregations

scale up/down
window size increase/decrease
batch siez increase/decrease

6.3 Joins

same as above

7. CONCLUSIONS

8. ACKNOWLEDGMENTS

9. ADDITIONAL AUTHORS

Additional authors: John Smith (The Thørväld Group, jsmith@affiliation.org), Julius P. Kumquat (The Kumquat Consortium, jpkumquat@consortium.net), and Ahmet Sacan (Drexel University, ahmetdevel@gmail.com)

10. REFERENCES

[1] 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. *The VLDB JournalThe International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[2] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[3] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 2015.

[4] J. L. Carlson. *Redis in Action*. Manning Publications Co., 2013.

Figure 3: Design of benchmark system.

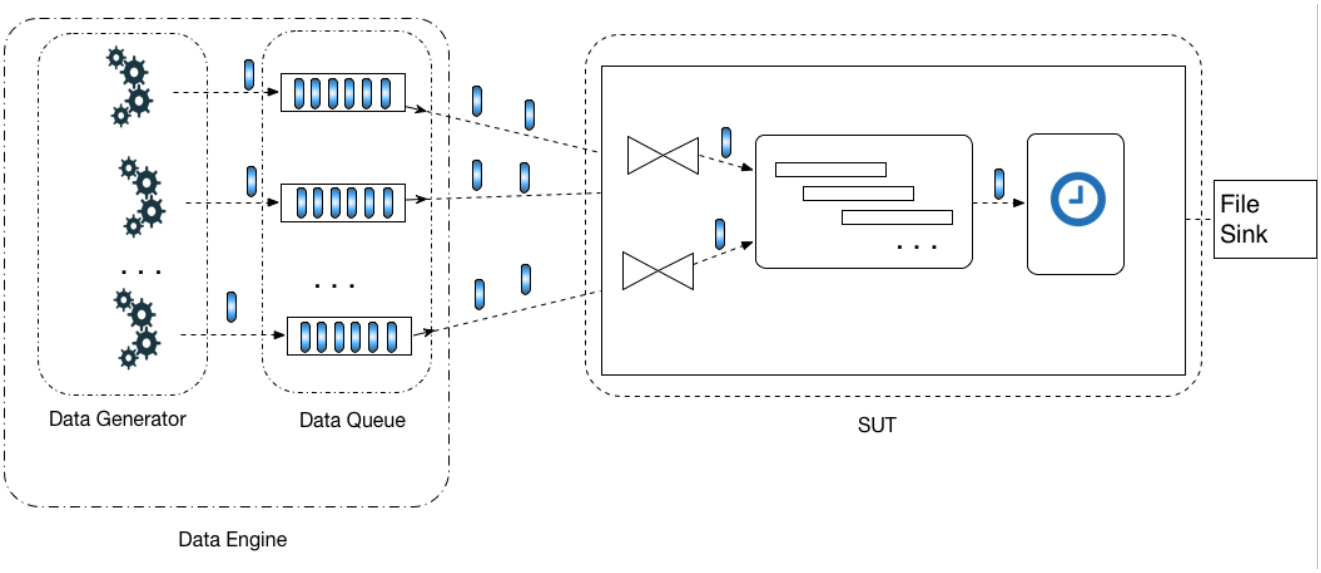
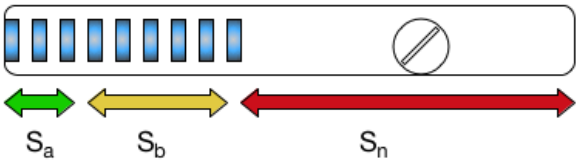


Figure 4: Basic intuition behind *back-pressure-compatible queue*



[5] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1789–1792. IEEE, 2016.

[6] DataArtisans. Extending the Yahoo! Streaming Benchmark. `http://data-artisans.com/extending-the-yahoo-streaming-benchmark/`, 2016. [Online; accessed 19-Nov-2016].

[7] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. Bigbench: towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 1197–1208. ACM, 2013.

[8] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *New Frontiers in Information and Software as Services*, pages 209–228. Springer, 2011.

[9] A. Kafka. A high-throughput, distributed messaging system. *URL: kafka. apache. org as of*, 5(1), 2014.

[10] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, page 53. ACM, 2015.

[11] M. A. Lopez, A. Lobato, and O. Duarte. A performance comparison of open-source stream processing platforms. In *IEEE Global Communications Conference (Globecom), Washington, USA*, 2016.

[12] R. Lu, G. Wu, B. Xie, and J. Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pages 69–78. IEEE, 2014.

[13] O.-C. Marcu, A. Costan, G. Antoniu, and M. S. Pérez. Spark versus flink: Understanding performance in big data analytics frameworks. In *Cluster 2016-The IEEE 2016 International Conference on Cluster Computing*, 2016.

[14] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, pages 170–177. IEEE, 2010.

[15] S. Perera, A. Perera, and K. Hakimzadeh. Reproducible experiments for comparing apache flink and apache spark on public clouds. *arXiv preprint arXiv:1610.04493*, 2016.

[16] 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 *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 1–14. ACM, 2013.

[17] A. Shukla and Y. Simmhan. Benchmarking distributed stream processing platforms for iot applications. *arXiv preprint arXiv:1606.07621*, 2016.

[18] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[19] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–499. IEEE, 2014.

[20] T. White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[21] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[22] 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 *Presented as part of the*, 2012.

10.1 References

Generated by bibtex from your .bib file. Run latex, then bibtex, then latex twice (to resolve references).

APPENDIX

You can use an appendix for optional proofs or details of your evaluation which are not absolutely necessary to the core understanding of your paper.