Exploratory Analysis of Spark Structured Streaming

Todor Ivanov
Frankfurt Big Data Lab, Goethe University
Frankfurt am Main, Germany
todor@dbis.cs.uni-frankfurt.de

Jason Taaffe
Frankfurt Big Data Lab, Goethe University
Frankfurt am Main, Germany
jasontaaffe@yahoo.de

ABSTRACT

In the Big Data era, stream processing has become a common requirement for many data-intensive applications. This has lead to many advances in the development and adaption of large scale streaming systems. Spark and Flink have become a popular choice for many developers as they combine both batch and streaming capabilities in a single system. However, introducing the Spark Structured Streaming in version 2.0 opened up completely new features for SparkSQL, which are alternatively only available in Apache Calcite.

This work focuses on the new Spark Structured Streaming and analyses it by diving into its internal functionalities. With the help of a micro-benchmark consisting of streaming queries, we perform initial experiments evaluating the technology. Our results show that Spark Structured Streaming is able to run multiple queries successfully in parallel on data with changing velocity and volume sizes.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; • **Information systems** → *Data management systems*; • **Computing methodologies** → *Symbolic and algebraic manipulation*;

KEYWORDS

Spark Structured Streaming, Spark, Big Data Benchmarking

ACM Reference Format:
Todor Ivanov and Jason Taaffe. 2018. Exploratory Analysis of Spark Structured Streaming. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion , April 9–13, 2018, Berlin, Germany,* Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/3185768.3186360

1 INTRODUCTION

Big Data is growing in both volume and velocity. The combination of both creates data streams. As is often the case, different disciplines use different definitions but a shared characteristic among these definitions includes: the real-time or near real-time nature of data arrival [13]. The data arrives in a continuous stream opposed to certain intervals and is unbounded in nature, hence the stream potentially never ends [1, 5, 20]. With the rise of the Internet of Things more data streams will be created, which will push the boundaries even further. Due to potentially millions of sensors constantly sending data in mere fractions of a second the torrent of data could reach between 10^2 and 10^5 messages per second. As noted by Shukla et al. [28], Twitter by comparison receives around 6000 tweets per second. Since analysis takes place in-memory, it is not feasible to store the data on disks or other external data storage devices [35].

Micro-batching is a hybrid concept , with the main idea being *"[...] to treat the stream as a sequence of small batch chunks of data. On small intervals, the incoming stream is packed to a chunk of data and is delivered to the batch system to be processed."*[26]. Spark Streaming [31] utilises this approach by limiting the batch size to keep latency at an acceptable level [35]. Other hybrid concepts follow similar approaches of combining batch processing with streaming, e.g. the Lambda Architecture [22] and the Kappa Architecture [16]. On the contrary, Flink [10] offers native streaming support, which is also used to implement batch operations. An important drawback of both Spark Streaming and Flink approaches is that you have to implement and compile your streaming application code, before being able to execute it. This restricts the technology usability and development efficiency. However, recently in Spark version 2.0, Spark Structured Streaming [14, 36], which enhances the SparkSQL [2] engine with streaming capabilities was introduced. The goal of Structured Streaming is to make it easier to create streaming applications, by extending SparkSQL, so that the user no longer has to worry about the details of implementing streaming and can focus on the results, while offering strong fault-tolerance and consistency. Similar approach follow the Flink Streaming [30], which uses Apache Calcite [6] as an underlying engine that provides the SQL streaming capabilities.

This work focuses on evaluating the new features of Spark Structured Streaming by using queries inspired by the BigBench V2 [11] benchmark and implemented in a streaming context. Our experiments showed multiple pros and cons of the current Structured Streaming technology.:

- The more queries that are run in parallel, the longer these need to be completed.
- An increase in file sizes also results in longer completion times, although this relationship was not always consistent in every file bracket.
- Tentative evidence points towards heap memory being the bottleneck, while CPU utilization decreased at larger file sizes.

ICPE '18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5629-9/18/04…$15.00
https://doi.org/10.1145/3185768.3186360

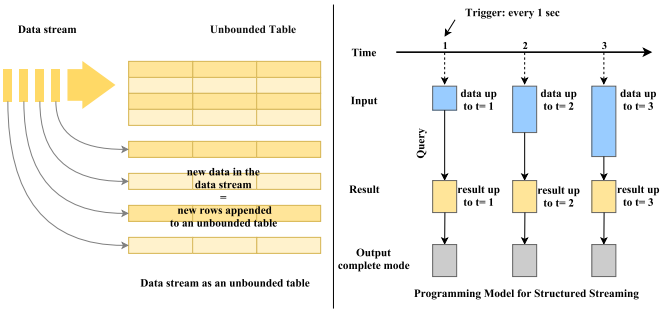


Figure 1: Structured Streaming Models [14]

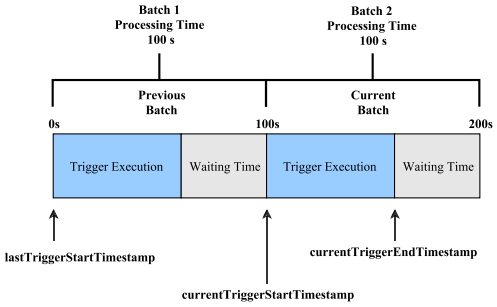


Figure 2: Trigger Process

The remaining paper is structured as follows: Section 2 describes the main features of Spark Structured Streaming. Section 3 looks at related benchmarks and studies followed by Section 4, which presents the micro-benchmarks used in the evaluation. Section 5 analyses the experimental results and the final Section 6 concludes the paper.

2 SPARK STRUCTURED STREAMING

Structured Streaming is a stream processing engine, which was built on top of the Spark SQL engine with fault-tolerance and scalability in mind [9]. It was first added in Apache Spark 2.0 to build and simplify the development of continuous and real-time Big Data applications, by combining batch and streaming computation [14]. It uses the already existing Dataset and DataFrame APIs and is intended to complement Spark Streaming in the long-term [36]. One of the main benefits is that users no longer have to concern themselves with the details of streaming as this is handled by the new streaming architecture.*"The key idea in Structured Streaming is to treat a live data stream as a table that is being continuously appended. [...] Every data item that is arriving on the stream is like a row being appended to the Input Table."* [14]. The left part of Figure 1 depicts this process in a graphic manner. Depending on a trigger interval set by the user, the input table is updated with new rows on which queries are run. The output of these queries is saved in a results table. This is illustrated in the right part of Figure 1. It shows that every second the input table is updated with new data and afterwards a query is run, the results of which are then saved in the result table. Following this, the user has several options for saving the content of the result table to an external storage.

Structured Streaming combats certain inherent weaknesses in streaming systems such as inconsistency when processing records that can lead to nonsensical results, fault tolerance both inside and outside the engine, and out-of-order data. Structured Streaming offers a guarantee: *"at any time, the output of the application is equivalent to executing a batch job on a prefix of the data."* [36]. It also supports event time aggregation to enable the processing of out of order data, which is very similar to grouped aggregations.

Spark collects a large assortment of metrics once the application is running, of which only a small subset is relevant to the benchmark that we ran. Table 1 below lists the names and a definition of each relevant metric with an explanation taken directly from the official Apache Spark Scala source files in Github [8].

During the benchmark implementation process, we identified a problem with the processingRate-total file. Due to an error in the MetricsReporter scala file, which is used to create the CSV files, the files inputRate-total and processingRate-total were reporting the same values. After opening a ticket and submitting a fix to the issue, this bug was fixed in the latest release (Bug [SPARK-22052]).

Figure 2 illustrates how the metrics relate to each other and to which point in the streaming process they correspond. The figure assumes processing time is set to 100 seconds to showcase the difference between processing time and trigger execution.

Changes in trigger processing time have different effects on the calculation of the metrics. *ProcessingTimeSec* is defined as *currentTriggerEndTimestamp - currentTriggerStart - Timestamp*, effectively the duration of the Trigger Execution. Setting trigger processing time to a specific value thereby does not affect this metric as the *Trigger Execution* time is not affected by the user, merely the amount of waiting time is altered by setting trigger processing time. However, *InputTimeSec* is affected by a change in trigger processing time as it is defined as *currentTriggerStartTimestamp - lastTriggerStartTimestamp*, thereby a decrease or increase in waiting time will alter this metric. Assuming no specific trigger processing time is set, meaning as soon as one trigger execution is completed the next one will start, the difference between *ProcessingRate* and *InputRate* will be marginal, but not zero.

3 RELATED WORK

Spark [29] and Spark Streaming [31] have been adapted by the industry as key technologies in developing big data applications. Therefore, multiple studies investigated how Spark performs under different workloads and benchmarks [25], including micro-benchmarks such as HiBench [15] , SparkBench [19] and Yahoo Streaming Benchmark [7]. Other important aspects like efficient memory management [3, 4, 17, 18] and competitiveness with other frameworks like MapReduce and Flink [21, 27, 33] were also investigated.

Recently, a new approach for performance clarity, called Monotasks, was presented by Ousterhout et al. [23, 24]. It points out the importance of understanding and visualizing the bottlenecks in todays' complex systems and was demonstrated through a Spark prototype. In the same spirit, an improvement of the Spark Streaming concept, called Drizzle [34], was also demonstrated to compensate

Table 1: Metrics

Name	Definition	Description
Latency	triggerExecution - The amount of time taken to perform various operations in milliseconds.	Describes the amount of time needed to perform the SQL query operations during one trigger interval.
InputRate-total	numRecords/inputTimeSec	Describes how many rows were loaded per second between the start of the last trigger and the start of the current trigger.
ProcessingRate-total	numRecords/processingTimeSec	Describes how many rows were processed per second during the start of the current trigger and the end of the current trigger.
InputTimeSec	currentTriggerStartTimestamp - lastTrigger-StartTimestamp	The period of time between the start of the current trigger and the start of the last trigger.
ProcessingTimeSec	currentTriggerEndTimestamp - currentTrigger-StartTimestamp	The period of time between the beginning and the end of a trigger period.
NumInputRows	NumRecords	The number of rows in a batch.

for the overhead of micro-batching and bring it closer to native streaming.

Another work by Zhang et al. [37] presented a distributed streaming query engine running on a cluster and compared its performance to other streaming engines, with Structured Streaming being one of them. The work found out that many streaming operations are unsupported in the current implementation of Structured Streaming and therefore it is not possible to fully compare it with other stream engines. However, when looking at those operations that did work, the Structured Streaming performance was worse than when using Spark Streaming. As there are no extensive evaluations of the Spark Structured Streaming performance and benchmarks targeting exactly these types of engines, we address this in the remaining part of our study.

4 STRUCTURED STREAMING MICRO-BENCHMARK

Before performing the evaluation it is important to identify the benchmark and methodology that will be used in the analysis. Since there are no benchmarks targeting streaming SQL engines as mentioned in the related work, we reuse and extend the BigBench V2 [11] benchmark as a basis for our micro-benchmark. BigBench (TPCx-BB) [12] is an end-to-end benchmark used to test Big Data systems, partly based on the TPC-DS benchmark. BigBench V2 [11] is the updated version of BigBench and addresses some of the drawbacks in BigBench. It no longer uses complex queries from TPC-DS and simplifies the schema. In addition, semi-structured data is no longer handled as a structured table with a fixed schema, instead semi-structured data is treated as a pair of key-values. The new BigBench V2 data model retains the variety of structured, semi-structured and unstructured data, however, the structured data component now only includes six tables and product reviews are generated in a synthetic manner. The benchmark consists of 30 queries covering different business workload requirements.

Our experiments consist of two phases. In the first phase, data was generated using the BigBench V2 data generator and then manually split into files stored on disk to simulate a stream of data. In the second phase the queries were triggered to execute on the simulated stream data and produced a set of statistical results reporting performance and resource characteristics. The following subsections describe the query workloads, how they are implemented and finally the data preparation and execution phases.

4.1 Workloads

Out of the 30 queries that were part of the original BigBench, four were selected for testing, because they were suitable for real-time analytics and relevant from a business and technical point of view. The fifth query (Q_{milk}) is very simple and checks how many products of a certain type are sold in a particular time frame. The streaming workload consists of these five queries executed periodically on a stream of data. All queries are defined in plain text as followed:

- Q_5 : Find the 10 most browsed products in the last 100 seconds.
- Q_6 : Find the 5 most browsed products that were not purchased across all users (or only specific user) in the last 100 seconds.
- Q_{16} : Find the top ten pages visited by all users (or specific user) in the last 10 minutes.
- Q_{22} : Show the number of unique visitors in the last hour.
- Q_{milk} : Show the sold products (of a certain product or category).

4.2 Setup

The experiments were performed on a workstation machine with 8GB main memory, Intel Core i5 CPU 760 @3.47GHz x4 and 1TB hard disk. On top, Ubuntu LTS 16.04 was installed running Java version 1.8.0.131, Scala version 2.11.2 and Apache Spark 2.3. Spark was used in standalone mode with the default configuration parameters for all experiments.

4.3 Data Preparation

For the data generation, we used the data generator in BigBench V2 [11] with scale factor 1 and in particular the web logs consisting of web clicks in JSON format (around 20GB) and the web sales structured data(around 10MB). Web log file sizes ranging from 50MB to 2000MB were created to test the system performance at different file sizes. For every file size, 10 files were created to simulate a stream of 10 data files. As the web sales file was only 10MB large, that size was retained, but the file was multiplied 10 times to ensure 10 files could be streamed successfully. In total, 31 different combinations of parallel query executions were tested, the web sales file size for the Qmilk query was kept equal the entire time at 10MB. The web log sizes for all other queries (Q5 and Q16) were increased stepwise, starting from 50MB up to 2000MB.

4.4 Implementation

To this end four existing queries were initially chosen (Q05, Q06, Q16 and Q22) and a new one was created (Qmilk) to test the Structured Streaming environment. Due to technical limitations present in Structured Streaming at the time of writing, it was not possible to run queries Q06 and Q22 because of their specific nature. Query Q06 is the most complex query and performs join operations on two streaming datasets, which is not (yet) supported by Structured Streaming, as pointed out by Zhang et al. [37]. Performing a join between one streaming dataset and one static dataset is possible, but the SQL statement includes multiple aggregations (initiating the count function), which is an operation not yet supported on streaming datasets. Query Q22 uses a distinct operation, which is also not supported on streaming datasets. Furthermore, it utilizes sorting operations (Order By) that are only supported after an aggregation. Lastly, the *Limit* keyword is not supported at all in a streaming context. Hence, it was only possible to implement and test queries Q05, Q16 and Qmilk. The Scala code is available on Github [32] together with all the test data and query starting code. Listing 1 shows the Scala Structured Streaming implementation of the three queries. The respective code examples are for Spark running in local mode, in order to run on a cluster several changes are necessary.

Listing 1: Query 5, Query 16 and Qmilk

```
var web_logs_05 = web_logsDF
      .groupBy("wl_item_id").count()
      .orderBy("count")
      .select("wl_item_id","count")
      .where("wl_item_id IS NOT NULL")

var web_logs_16 = web_logsDF
      .groupBy("wl_webpage_name").count()
      .orderBy("count")
      .select("wl_webpage_name","count")
      .where("wl_webpage_name IS NOT NULL")

var web_sales_milk = web_salesDF
      .groupBy("ws_product_id").count()
      .orderBy("ws_product_id")
      .select("ws_product_id","count")
      .where("ws_product_ID IS NOT NULL")
```

Listing 2 shows how the query execution is triggered for query Q05, which is performed in phase two of the benchmark. Furthermore, the format option determines which output sink is to be used. As of writing there are four available options: *file sink*, *foreach sink*, *console sink* and *memory sink*. The *file sink* option stores the output to a directory, by setting the text within format to either parquet, json, csv or similar file types. The *foreach sink* runs arbitrary computation on the records. The *console and memory sink* options are primarily used for debugging purposes or when file sink is not an option. For our benchmark the *console* option, shown in Listing 2, was chosen to facilitate error detection and because the queries were preventing the use of *the file sink* option. As of writing it is not possible to write output to file if the queries use aggregation, which is the case here.

By setting the optional *queryName* in the configuration, the query gets an internal name that is used for reporting purposes and in naming the statistical files. The *trigger* configuration option is not required to run a streaming query. It determines at which interval Spark adds new rows to the input table. If the line is omitted Spark will update the input table as soon as possible, depending on the system performance, file complexity and the file size used for the streaming dataset. An additional option is to run the trigger only one time *trigger(Trigger.Once())* after which the query will stop. If the user sets the processing time to be faster than the system can manage a warning message, with the actual processing time will be displayed, after the query is executed. The *outputMode* configuration option on Listing 2 (depicted also in the right part of Figure 1) determines what mode will be selected when writing the output to external storage. As of writing there are three mode options: *complete mode*, *append mode* and the newest, *update mode*. When *complete mode* is selected the entire updated result table will be written to the external storage. When choosing *append mode*, Spark only writes the new rows that were added to the result table since the last trigger to the external storage. The final option, *update mode* tells Spark to only write those rows to external storage that were updated in the result table since the last trigger, the emphasis here being old rows that were updated, and not new ones being added. The different modes are also subject to certain limitations.

Listing 2: Spark WriteStream

```
var query05 = web_logs_05.writeStream
      .format("console")
      .queryName("05")
      .trigger(Trigger
        .ProcessingTime("150 seconds"))
      .outputMode(OutputMode.Complete())
      .start()
```

Append mode does not work in our situation as queries are being run using aggregations based not on event-time, but on other attributes such as the number of web pages or product and item IDs. Furthermore, *update mode* is also not possible when using these queries as they include sorting operations which are not permitted. Hence, *complete mode* is the only available option, but at the same

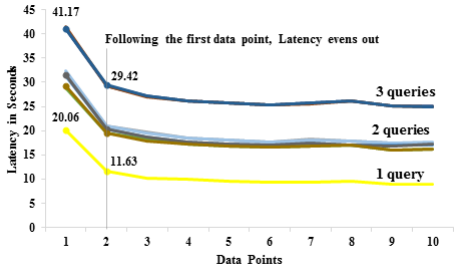


Figure 3: Latency Distribution

time this prevents file sink from being used as the output sink due to technical restrictions. Finally *.start()* tells Spark to initiate the streaming query.

4.5 Limitations

This subsection outlines some Structured Streaming limitations and problems that we encountered during our experiments.

File Creation Date: During testing, we discovered strange behavior in Structured Streaming, when manually copying and moving the data files to the assigned directories. It appeared that when all the streaming files were copied to the streaming directory and had the same file creation or modification date, Spark considered these to be the same file even if the files had different names (web-logs1, web-logs2, etc.). After starting the program, the results of the SQL queries did not update following the first file. Instead the results array was either blank or statically displayed the results obtained after the first file was processed. This issue could not be replicated in all cases, but by making sure the file creation or modification dates differed slightly or by cutting, not copying, the files to the streaming directory, this issue was solved.

Bigger File Sizes: The largest file size used was 2000MB, this is relatively small in the Big Data context and did not test Sparks true capabilities. Future research should use larger files to investigate potential bottlenecks and peak performance. Additionally, as only ten files were streamed in each micro-benchmark the influence of fluctuations in system performance and other external factors cannot be discounted. To solve this issue longer streams with more files should be tested to facilitate more robust testing conditions.

Spark Cluster Mode: Finally, as Spark was run in standalone mode and not on a cluster the advantages of parallel computation were not utilized. The use of BigBench as an end-to-end benchmark on a cluster to reflect actual usage scenarios should be conducted in the future research.

5 EXPLORATORY ANALYSIS

This section looks at the results obtained after performing a series (around 31 different combinations) of experiments using the 3 queries implemented with Spark Structured Streaming. Initially, we observed that all runs took much longer than the following run as depicted in Figure 3. The reason for this is that the system needed to warm up before reaching a consistent level of processing speed. Therefore, the first measured data point out of the total 10 data points (streamed files) was omitted from our analysis.

Figure 4 depicts the average execution times excluding the first run for all combinations of queries. We can identify three main groups of queries: group 1 consists of single executions of Q5 and Q16; group 2 consists of parallel pair executions of Q5, Q16 and Qmilk and finally group 3 consists of triple parallel executions of Q5, Q16 and Qmilk. In general, it can be said that the more queries are run in parallel and the larger the file sizes used, the longer these take, to be completed and the more resource intensive they are.

For example, Q5 median latency increased by more than 200% between the 50MB and 2000MB file sizes and when testing resource utilization across file sizes, the time to achieve completion more than doubled between the smallest and the largest file sizes. When comparing single query latency to triple query latency, it increased two to three-fold. Additionally, the larger the file sizes the more fragmented the query groupings became. Initially all queries could be assigned to three distinct groups based on their latency distribution and how many queries were running at the same time, yet later on five or six groups were observed.

The resource utilization also showed interesting results as CPU utilization was the highest at the 50MB level and decreased every time the file size was increased. In contrast, the JAVA heap size increased for larger file sizes. It looks like the memory size will be the limiting factor when using larger files, but this needs to be further proved with extensive tests in future research.

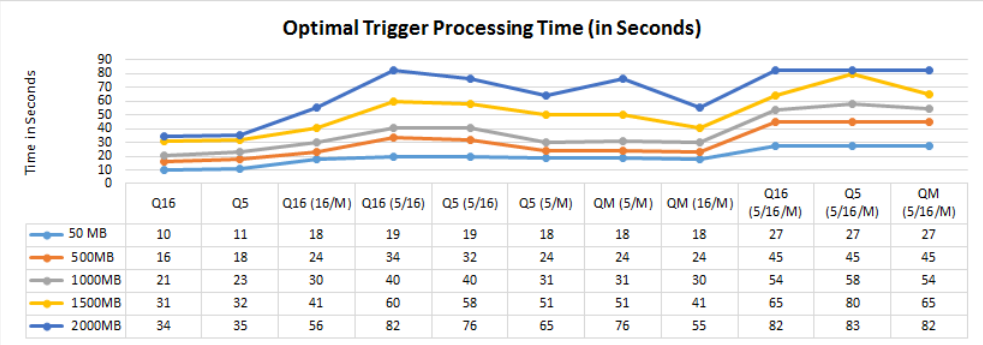
Another point is the gathering of Spark metrics. It needs to be improved and made more user friendly as, it was often more efficient to extract them via the log4j configuration compared to the various sink options in Spark. Additionally, the sink option only exports three streaming metrics, whereas the log4j method offers more metrics for evaluation. The Structured Streaming trigger process itself is not documented and thorough experimentation was needed to understand the inner workings of the process as mentioned in Section 2 (Figure 2).

6 LESSONS LEARNED

Table 2: Structured Streaming Pros and Cons

	Pros	Cons
1	Simple programming model and streaming API	Undocumented trigger process
2	Several built-in metrics	Query limitations due to streaming API
3	Several extraction and sink options for metrics	Complicated metric extraction process
4	Possible to run queries in parallel	

Based on our exploratory analyses and experiments we summarize the finding of our study. Structured Streaming exhibits several advantages over the Spark legacy streaming module, but is also subject to certain weaknesses listed in Table 2. The simple programming model and streaming API are countered by query limitations that restrict queries to operations supported in the current version



	Q16	Q5	Q16 (16/M)	Q16 (5/16)	Q5 (5/16)	Q5 (5/M)	QM (5/M)	QM (16/M)	Q16 (5/16/M)	Q5 (5/16/M)	QM (5/16/M)
50 MB	10	11	18	19	19	18	18	18	27	27	27
500MB	16	18	24	34	32	24	24	24	45	45	45
1000MB	21	23	30	40	40	31	31	30	54	58	54
1500MB	31	32	41	60	58	51	51	41	65	80	65
2000MB	34	35	56	82	76	65	76	55	82	83	82

Figure 4: Optimal Trigger Processing Time

of the API. And even though several metrics are already built-in to Structured Streaming, including various ways of extracting these for analysis, this process is still fairly complicated. By streamlining the metric extraction process it would facilitate faster analysis of the data and make Structured Streaming more attractive to companies searching for viable streaming solutions. Finally, the trigger process is not documented in a detailed and intuitive manner. As Structured Streaming is the newest addition to Apache Spark it still has room for improvement. Future updates could remedy these issues by improving documentation and extending the metric environment in Apache Spark.

REFERENCES

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *PVLDB* 8, 12 (2015).

[2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *the 2015 ACM SIGMOD, Melbourne, Victoria, Australia, May 31 - June 4, 2015.*

[3] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguadé. 2015. How Data Volume Affects Spark Based Data Analytics on a Scale-up Server. In *6th Workshop, BPOE 2015, Kohala, HI, USA, Aug. 31 - Sept. 4, 2015.*

[4] Ahsan Javed Awan, Vladimir Vlassov, Mats Brorsson, and Eduard Ayguadé. 2016. Node architecture implications for in-memory data analytics on scale-in clusters. In *the 3rd IEEE/ACM BDCAT 2016, Shanghai, China, Dec. 6-9, 2016.*

[5] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002. Models and Issues in Data Stream Systems. In *the 21st ACM PODS, June 3-5, Madison, Wisconsin, USA.*

[6] Calcite. 2018. calcite.apache.org/. (2018).

[7] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Peng, and Paul Poulosky. 2016. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *2016 IEEE IPDPS Workshops, Chicago, IL, USA, 2016.*

[8] Structured Streaming Code. 2018. https://github.com/apache/spark/tree/fa0092bddf695a757f5ddaed539e55e2dc9fccb7/sql/core/src/main/scala/org/apache/spark/sql/streaming. (2018).

[9] Srinivas Duvvuri and Bikramaditya Singhal. 2016. *Spark for Data Science.* Packt Publishing Ltd.

[10] Flink. 2018. flink.apache.org/. (2018).

[11] Ahmad Ghazal, Todor Ivanov, Pekka Kostamaa, Alain Crolotte, Ryan Voong, Mohammed Al-Kateb, Waleed Ghazal, and Roberto V. Zicari. 2017. BigBench V2: The New and Improved BigBench. In *33rd IEEE ICDE 2017, San Diego, CA, USA, April 19-22, 2017.*

[12] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. 2013. BigBench: towards an industry standard benchmark for big data analytics. In *the ACM SIGMOD 2013, New York, NY, USA, June 22-27, 2013.*

[13] Lukasz Golab and M. Tamer Özsu. 2003. Issues in data stream management. *SIGMOD Record* 32, 2 (2003).

[14] Structured Streaming Programming Guide. 2018. spark.apache.org/docs/latest/structured-streaming-programming-guide.html. (2018).

[15] HiBench. 2017. github.com/intel-hadoop/HiBench. (2017).

[16] Jay Kreps. 2014. Questioning the lambda architecture. *Online article, July* (2014).

[17] Mayuresh Kunjir and Shivnath Babu. 2017. Thoth in Action: Memory Management in Modern Data Analytics. *PVLDB* 10, 12 (2017).

[18] Mayuresh Kunjir, Yuzhang Han, and Shivnath Babu. 2016. Where does Memory Go?: Study of Memory Management in JVM-based Data Analytics. (2016). https://pdfs.semanticscholar.org/8590/b5d66e0dc429578cf6ac64b8abda6a125701.pdf

[19] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. 2017. SparkBench: a spark benchmarking suite characterizing large-scale in-memory data analytics. *Cluster Computing* (2017).

[20] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. 2014. Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks. In *the 7th IEEE/ACM UCC 2014, London, United Kingdom, Dec. 8-11, 2014.*

[21] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, and María S. Pérez-Hernández. 2016. Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks. In *the IEEE CLUSTER 2016, Taipei, Taiwan, 2016.*

[22] Nathan Marz and James Warren. 2015. *Big Data: Principles and best practices of scalable realtime data systems.* Manning Publications Co.

[23] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *26th SOSP, Shanghai, China, 2017.*

[24] Kay Ousterhout, Christopher Canel, Max Wolffe, Sylvia Ratnasamy, and Scott Shenker. 2017. Performance clarity as a first-class design principle. In *the 16th Workshop HotOS 2017, Whistler, BC, Canada, May 8-10, 2017.*

[25] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *the 12th USENIX NSDI 15, Oakland, CA, USA, May 4-6, 2015.*

[26] Saeed Shahrivari. 2014. Beyond Batch Processing: Towards Real-Time and Streaming Big Data. *Computers* 3, 4 (2014).

[27] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. 2015. Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics. *PVLDB* 8, 13 (2015).

[28] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. 2017. RIoTBench: An IoT benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience* 29, 21 (2017).

[29] Spark. 2018. spark.apache.org/. (2018).

[30] Flink Streaming. 2018. ci.apache.org/projects/flink/flink-docs-release-1.4/dev/table/streaming.html. (2018).

[31] Spark Streaming. 2018. spark.apache.org/streaming/. (2018).

[32] Jason Taaffe. 2018. https://github.com/Taaffy/Structured-Streaming-Micro-Benchmark. (2018).

[33] Jorge Veiga, Roberto R. Expósito, Xoan C. Pardo, Guillermo L. Taboada, and Juan Touriño. 2016. Performance evaluation of big data frameworks for large-scale data analytics. In *2016 IEEE BigData 2016, Washington DC, USA, Dec. 5-8, 2016.*

[34] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *26th SOSP, Shanghai, China, 2017.*

[35] Wolfram Wingerath, Felix Gessert, Steffen Friedrich, and Norbert Ritter. 2016. Real-time stream processing for Big Data. *Information Technology* 58, 4 (2016).

[36] Zaharia. 2016. databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html. (2016).

[37] Yunhao Zhang, Rong Chen, and Haibo Chen. 2017. Sub-millisecond Stateful Stream Querying over Fast-evolving Linked Data. In *26th SOSP, Shanghai, China, Oct. 28-31, 2017.*