### PROTEUS
Scalable online machine learning for predictive analytics and real-time interactive visualization

687691

---

## D3.3 Hybrid computation tested system

**Lead Author:** Jeyhun Karimov

**With contributions from:** Bonaventura Del Monte, Alireza Rezaei Mahdiraji

**Reviewer:** Daniel Higuero

<table>
<thead>
<tr>
<th>Deliverable nature:</th>
<th>Demonstrator (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contractual delivery date:</td>
<td>31/05/2017</td>
</tr>
<tr>
<td>Actual delivery date:</td>
<td>30/04/2017</td>
</tr>
<tr>
<td>Version:</td>
<td>1.0</td>
</tr>
<tr>
<td>Total number of pages:</td>
<td>18</td>
</tr>
<tr>
<td>Keywords:</td>
<td>Benchmarking, hybrid computation</td>
</tr>
</tbody>
</table>
Abstract

Over the last years, Stream Data Processing (SDP) has been gaining attention both in industry and in academia due to its wide range of applications. Hybrid data processing (HDP) is a special case of stream processing where batch processing is one component of stream data processing system (SDPS). This demonstrator concentrates on the evaluation of hybrid data processing used in PROTEUS project. We develop generic solutions for benchmarking SDPSs which can be used for HDP as well. In particular, our evaluation focuses on measuring the throughput and latency of HDPs. For this benchmark, we design workloads based on PROTEUS use-case. The main contribution of this work is threefold. First, we give a definition of latency and throughput for stream data processing systems and in particular for HDPSs. Second, we completely separate the System Under Test (SUT) and driver, so that the measurement results are closer to the system performance under real conditions. Third, we build the driver to test the actual sustainable performance of a system under test.
Executive summary

Processing large volumes of data in batch is often not sufficient in cases where new data has to be processed fast to quickly adapt and react to changes. For that reason, SDP has gained significant attention. Apache Flink is one of the most popular engines used for SDP. HDP is a special case of SDP which incorporates batch data processing into SDPS. In this demonstrator, we develop a benchmark framework for HDP on top of Apache Flink. Our solution is not specific to HDP and can be used for benchmarking any SDPSs. Therefore, throughout this demonstrator, we refer to our solution as benchmarking framework for SDPSs.

The application area of HDP in PROTEUS is detecting errors in coil production based on both historical and real-time data. Sensors in a coil production pipeline produce massive amount of real-time data. PROTEUS HDP engine processes massive amounts of online sensor data in real time with defined business logic to detect whether a particular sensor data is defective by doing lookups from historical data. We build our benchmark framework based on this use-case.

There are several challenges to be addressed when designing a benchmarking framework for SDPSs. The first challenge is to design an efficient and simple benchmarking framework as complex benchmarking frameworks can cause additional, often hidden overheads. The second challenge is to isolate the driver from the SUT as much as possible. In previous works [1-4], researchers measured the throughput either inside the SUT, or used internal statistics of the SUT. The third challenge is to measure the metrics with close to real-life scenarios. One example is the throughput measurement. In this demonstrator, we analyze all challenges and provide our solutions to overcome all of them.

The main contributions of this demonstrator are:

- We introduce a mechanism to accurately measure the latency of stateful operators in SDPSs. We apply our measuring techniques to PROTEUS use-case.
- We accomplish the complete separation of the test driver from the system under test.
- We measure the maximum sustainable throughput of the SDPS. Our benchmarking framework handles system specific features like backpressure to measure the correct maximum sustainable throughput of a system.
- We use our benchmarking framework for an extensive evaluation of PROTEUS engine’s HDP feature with a practical use-case.
Abstract (for dissemination)

Over the last years, stream data processing (SDP) has been gaining attention both in industry and in academia due to its wide range of applications. Hybrid data processing (HDP) is a special case of stream processing where batch processing is one component of stream data processing system (SDPS). This demonstrator concentrates on the evaluation of hybrid data processing system (HDPS) used in PROTEUS. We develop generic solution for benchmarking SDPSs which can be used for HDPS as well. Our evaluation focuses in particular on measuring the throughput and latency of HDPS. For this benchmark, we design workloads based on PROTEUS use-case. The main contribution of this work is threefold. First, we give a definition of latency and throughput for stream data processing systems and in particular for HDPSs. Second, we completely separate the system under test (SUT) and driver, so that the measurement results are closer to the system performance under real conditions. Third, we build the driver to test the actual sustainable performance of a system under test.

Keywords

Benchmarking, hybrid computation
## Version Log

<table>
<thead>
<tr>
<th>Issue Date</th>
<th>Rev. No.</th>
<th>Author</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>04.05.2017</td>
<td>0.1</td>
<td>Daniel Higuero</td>
<td>Typos fixed, more experiments conducted</td>
</tr>
<tr>
<td>23.05.2017</td>
<td>1.0</td>
<td>Marcos Sacristán Cepeda</td>
<td>Filled missing points in document information section, added source code repository as a reference.</td>
</tr>
</tbody>
</table>
# Table of Contents

Executive summary ................................. 3  
Document Information .............................. 4  
Table of Contents .................................. 6  
List of figures and/or list of tables ............... 7  
Abbreviations ....................................... 8  
1  Introduction ..................................... 9  
2  Challenges ...................................... 11  
   2.1 Efficiency is key ............................. 11  
   2.2 Separate driver and system under test ...... 11  
   2.3 Separate driver and system under test ...... 11  
3  Benchmarking framework design ................ 13  
   3.1 Use-case: detecting defective coils ........ 14  
   3.2 Metrics .................................. 14  
4  Evaluation ..................................... 15  
5  Conclusion .................................... 18  
6  References ..................................... 19
List of figures and/or list of tables

Figure 1. Different system designs to connect data generator and SUT ................................................................. 10  
Figure 2. Design of the benchmark framework ....................................................................................................... 13  
Figure 3. Basic intuition behind the backpressure-compatible queue ........................................................................ 14  
Figure 4. Latency distributions in histogram ........................................................................................................... 15  
Figure 5. Latency distributions in time series ........................................................................................................... 16  

Table 1. Sustainable throughput of SUT. The unit time is 1 second ........................................................................ 17  
Table 2. Latency statistics, avg, min, max, and quantiles (90, 95, 99) in milliseconds ................................................. 17
## Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>SDP</td>
<td>Stream Data Processing</td>
</tr>
<tr>
<td>SDPS</td>
<td>Stream Data Processing System</td>
</tr>
<tr>
<td>SUT</td>
<td>System under test</td>
</tr>
<tr>
<td>DI</td>
<td>Driver Instance</td>
</tr>
<tr>
<td>DGI</td>
<td>Data Generator Instance</td>
</tr>
<tr>
<td>DQI</td>
<td>Data Queue Instance</td>
</tr>
<tr>
<td>HDPS</td>
<td>Hybrid Data Processing System</td>
</tr>
</tbody>
</table>
1 Introduction

Processing large volumes of data in batch is often not sufficient in cases where new data has to be processed fast to quickly adapt and react to changes. For that reason, SDP has gained significant attention. Apache Flink [1] is one of the most popular engines used for SDP. HDP is a special case of SDP which incorporates batch data processing into SDPS. In this demonstrator, we develop a benchmark framework for HDP on top of Apache Flink. Our solution is not specific to HDP and can be used for benchmarking any SDPSs. Therefore, throughout this demonstrator, we refer to our solution as benchmarking framework for SDPSs.

The application area of HDP in PROTEUS is detecting errors in coil production based on both historical and real-time data. Sensors in a coil production pipeline produce massive amounts of real-time data. PROTEUS HDP engine processes online sensor data in real time with defined business logic to detect if particular sensor data is defective by doing lookups from historical data. We build our benchmark framework based on this use-case.

In this demonstrator, we propose a benchmarking framework to accurately measure the performance of HDP built on top of Apache Flink. We measure the latency and throughput as the key performance indicators (KPIs). Latency, in SDP, is the time difference between data production at the source (e.g., the sensors in coil production) and the result output at the sink of the dataflow graph describing the stream processing operations. In this scenario, key performance indicators (KPIs). Latency, in SDP, is the time difference between data production at the source (e.g., the sensors in coil production) and the result output at the sink of the dataflow graph describing the stream processing operations. In this scenario, throughput determines the number of ingested and successfully processed records or events per time unit.

Even though there have been several benchmarks [1-4] of the performance of SDPS recently, none of them measure the latencies and throughput that can be achieved in a production setting. One of the repeating problems in previous evaluations is a missing definition and inaccurate measurement procedure for latency of stateful operators in SDPS. Another challenge is a missing separation between the system under test (SUT) and the benchmark driver. Frequently, the performance metrics are measured and calculated within the SUT; this means that the results will be influenced through the measurements and, thus, can be biased. We discuss these challenges as well as our solutions in detail in Section 2.

In this demonstrator, we address the challenges mentioned above. The proposed solution is generic, has a clean design with clear semantics, and can be applied to any SDPS as well as SDPS with HDP feature. The main goal is to create an environment in which we can measure the metrics more precisely and with minimum external influence.

The main contributions of this demonstrator are:

- We introduce a mechanism to accurately measure the latency of stateful operators in SDPSs. We apply our measuring techniques to PROTEUS use-case.
- We accomplish the complete separation of the test driver from the system under test.
- We measure the maximum sustainable throughput of the SDPS. Our benchmarking framework exploits system specific features like backpressure to measure the correct maximum sustainable throughput of a system.
- We use our benchmarking framework for an extensive evaluation of PROTEUS engine’s HDP feature with a practical use-case.

The benchmark source code is publicly available [5]. The remainder of this demonstrator is organized as follows. In Section 2, we discuss the detailed interpretation of stream benchmarking challenges and their importance. We provide the design of our benchmarking system, the use-cases, and the metrics in Section 3. After a detailed evaluation in Section 4, we conclude in Section 5.
2 Challenges

There are several challenges to be addressed when designing a benchmarking framework for SDPSs. In this section, we analyze these challenges and explain our solutions.

2.1 Efficiency is key

The first challenge is to design an efficient, simple benchmarking framework to avoid additional and often hidden overheads of complex benchmarking. For example, the connection between the data generator and the SUT can cause large additional overheads. To test the stream data processing engine, a data generator is an essential component which provides large amounts of streaming data. Figure 1 shows three possible cases to link the data generator and the SUT. The simplest design is to connect the SDPS directly to the data generators as shown in Figure 1a. Although this is a perfectly acceptable design, it does not match real-life use-cases. In large-scale setups, stream data processing engines do not connect to push-based data sources but pull data from distributed message queues. A pull-based design, where data sources and SUT are connected through queues, is shown in Figure 1b. A common bottleneck of this option is the throughput of the message queuing system. Moreover, this adds a (de-)serialization layer between the SUT and the data sources. Therefore, we use a third option, which is a hybrid of the first two. As can be seen in Figure 1c, we embed the queues as a separate module in the data generators. This way, the throughput is bounded only by the network bandwidth and the systems work more efficiently as there are no (de-)serialization overheads of intermediate layer.

2.2 Separate driver and system under test

The second challenge is to isolate the driver from the SUT as much as possible. In previous works, researchers measured the throughput either inside the SUT, or used internal statistics of the SUT. However, if the measurements and SUT are not separated, then the measurement computations can influence the results of benchmark. In our benchmarking framework, we separate the driver and SUT for each measurement metric and perform measurements separately from the SUT.

The first metric that we measure is throughput. If the system can ingest and process all the data produced by the driver instances during the whole experiment, then the system can sustain the given throughput and we call it sustainable throughput. If the system sustains the given throughput and cannot process more, then we call it maximum sustainable throughput. In this paper, we analyze the maximum sustainable throughput of SDPSs which is sum of the throughputs of all driver instances. We keep the throughput assessment inside the driver instances and sum them at the end of each experiment.

The second metric is latency. We define the latency of a tuple to be the interval between the tuple’s event-time and the emission time from sink operator of SUT. We analyse both metrics in more depth in Section 3.1.1.

2.3 Separate driver and system under test

The third challenge is to measure the metrics with close to real-life scenarios. One example is the throughput measurement. In the previous SDPS benchmarks, the throughput of a SUT is measured by either taking quantiles over the test duration, or showing max, min, and average results. From a user’s perspective on the other hand, the system’s throughput is the upper limit throughput that it can sustain in a realistic setup. We propose user-defined sustainability policies in our benchmarking framework. For example, the benchmarking framework can take into account the backpressure of the SUT.

Another example for this challenge is latency measurement. If there are additional systems between the SDPS and data source (driver), then those are likely to add further latency for each tuple. One solution is to measure
the latency in the SDPS the same way as it is measured in batch data processing systems. In this case, the number and complexity of the systems between SDPS and data source is irrelevant as the latency is the interval between tuple ingestion and output from SUT. However, this measurement of latency is too optimistic and does not result in the real latency of the tuple. The reason is that the actual latency is based on tuples event-time. For example, in PROTEUS use-case, the event-time can be a sensor’s measurement time, the latency then is the time interval between measurement time of coil and the result being emitted from the SDPS. Therefore, the usual method of measuring the latency within SDPS does not conform to real world scenarios especially in pull-based SDPSs. Depending on their backpressure mechanism, a pull-based SDPS will reduce the input rate on high loads, which will not be reflected in the measured latency. To solve this, we define the latency for SDPSs to be the difference between tuple event-time and output time. Because SDPS is just one part of the whole PROTEUS environment, the actual latency is the time between sensor’s measurement of coil and user’s seeing the corresponding change in dashboard. Because the focus of this demonstrator is benchmarking HDP feature of Flink, we concentrate only on the latency of stream data processing part of the PROTEUS pipeline.

Another factor which can lead to unrealistic latency measures is the data generation rate. If the input data rate is higher than the maximum sustainable throughput of SDPS and we measure the event-time latency, the results may not exhibit the real latency. Initially the system will try to ingest as much data as possible. As a result, the processing time of SUT will take longer and the latency will increase for every adjacent tuple. To solve this issue, we conduct experiments with the maximum sustainable throughput for each SDPS.
3 Benchmarking framework design

In this section, we discuss the design of the benchmarking framework. As shown in Figure 2, there are two main components of a test deployment: 1) the SUT and 2) the driver. The driver has two subcomponents: i) the Data Generator and ii) the Data Queue.

The Driver Instance (DI) is a combination of the Data Generator Instance (DGI) and the Data Queue Instance (DQI). The driver is the combination of all DIs. Similarly, the Data Generator Subcomponent (DGS) and Data Queue Subcomponent (DQS) are the combination of all DGIs and DQIs, respectively. The driver is responsible for generating and queuing the data. It is composed of finite number of instances, which are distributed evenly to the worker nodes in the cluster. The driver nodes are separate from the SUT nodes in the cluster deployment. The DGI and the DQI reside in the same machine to avoid any network overhead and to ensure data locality. The data is kept in memory to avoid disk write/read overhead. The first subcomponent, the DGS, generates data with a constant speed. Because of the bottlenecks explained in Section 4, we avoid using mediator data queueing systems between the driver and SUT but we use distributed local queues (DQS). Data is queued in first-in/first-out manner in the DQS. As explained earlier, this approach is not different from implementing centralized and distributed message queues between the SUT and the driver. The following analogy can be made: the queue topic in a distributed message queuing system is analogous to DQS, and the queue partition is analogous to DQI.

The data generator timestamps every event. The latency of event of event is calculated from this point, i.e., the longer it stays in a queue, the higher its latency. The number of DIs can be arbitrary and the overall throughput that can be generated is only bounded by the network bandwidth.

![Figure 1. Design of the benchmark framework](image-url)
3.1 Use-case: detecting defective coils

We use PROTEUS use-case for this benchmark with a dataset provided by ArcelorMittal. The main goal is to detect defective coils in real-time. To reach this goal, the engine performs join operations between real-time sensor records and historical data of defective coils and computes similarity among the records. If the similarity is above a given threshold, then we mark the particular sensor record as defective. The batch join operator is efficiently ingested into SDPS. It is of crucial importance to run this pipeline in real-time, because high latency of the engine leads to late detection of deficient coils, which in turns increases the production costs of the manufacturer. The main goal is to detect defective coils in real-time.

3.2 Metrics

The metrics for this benchmark are latency and throughput. In this section, we give the definition of each metric and explain our solution to measure it.

As we defined above, we use the maximum sustainable throughput to measure system workload. Throughout the experiments, the data generation speed in all DIs is equal and constant. To examine if a system can sustain a given throughput, we divide the queue used in the DQI into three parts: $q^a$, $q^b$ and $q^n$. Figure 3, shows the example partitioning of the queue. If the size of the queue is less than or equal to $q^a$ throughout the experiment, then this is acceptable, meaning the SUT can sustain the given throughput. If the queue size is between $q^a$ and $q^b$ on the other hand, the SUT cannot sustain the given data rate, but the driver might tolerate it for some time. Slow system initialization and backpressure can create long queue. However, if the queue size is longer than $q^b$ then the SUT cannot sustain the given throughput and the latency is expected to continuously increase. In this case, we end the experiment.

Our system supports user-defined policies to test the sustainability. For example, one policy tolerates the $q^b$ part of the queue for a given time period (time based policy). Another policy tolerates the $q^b$ part of the queue for a given number of pushes into the queue (count based policy). This policy is customizable and can be easily plugged into the driver.

We choose a policy that is a combination of time and count based policies. Once the queue size exceeds the $q^a$ and is less than $q^b$, we wait until $\frac{q^b-q^a}{2}$ elements are pushed into the queue and check again. If the queue size is still bigger than $q^b$ we end the benchmark concluding the SUT cannot sustain the given throughput. This process is done continuously in all DIs and if the SUT fails to sustain one instance of the driver, then the experiments are halted meaning the SUT cannot sustain the given throughput. That is, the SUT can sustain a given workload if it can sustain the throughput of all DIs.

![Figure 2. Basic intuition behind the backpressure-compatible queue](image-url)
4 Evaluation

In this section, we discuss our experimental results. We run experiments in 2-, 4- and 8-node clusters to measure the scale out feature of the SUT. The experiments are done with the maximum sustainable throughputs. Throughout the experiments, we use 16 parallel DIs in the cluster. Each experiment duration is approximately 30 minutes. We allocate 16 CPUs and 16GB RAM for every node. The system clocks of all nodes in the cluster are synchronized via an NTP server. The network bandwidth is 1Gb/s. We select q^a to be 5% and q^b 10% of the overall input. We use approximately 25% of the real time data as a warmup. We enable the backpressure to ensure the durability of experiments. That is, we do not want the systems to ingest more input than they can process and crash during the experiment. In preliminary experiments, measuring the maximum throughput without backpressure mechanisms lead to various exceptions. If the SUT drops one or more connections to the DI, then the experiment is halted with the conclusion that the SUT cannot sustain the given throughput. Similarly, in real-life if the system cannot sustain the user feed and drops connection, then this is considered as a failure. The system clocks of all nodes in the cluster are synchronized via an NTP server. We use 2 historical data sets. DS1 contains 5.2M entries, DS2 contains 52M entries.

Tuning configuration parameters of the engine is important to get a good performance for the given use-case. There are several properties for each SDPS, that need to be tuned to customize the given use-case. For example, the buffer size has to be adjusted properly to ensure a good balance between throughput and latency. Although selecting low buffer sizes can result in low system latency, the actual latency of tuples may increase as they will be queued in the DQI instead of the buffers inside the system.

![Figure 3. Latency distributions in histogram.](image)

Figure 4 shows the latency distributions in benchmark results. We conducted the experiments with maximum sustainable throughput. We can clearly see that for both datasets, DS1 and DS2, scaling-out lowers the overall latency distribution. The main reason is that the engine distributes the historical data among the nodes by key, so the engine performs join operations in parallel throughout the cluster nodes. Because, the latency distribution figures for 8-node cluster were indistinguishable with the same scale we used in Figure 4, we didn’t put them.
Figure 4. Latency distributions in time series. x axis shows the event time of tuple and y axis shows the corresponding latency.

Figure 5 shows the latency distribution in time series format in benchmark results. We conducted the experiments with maximum sustainable throughput. Similar to Figure 4, we can see that there is significant latency improvement when scaling out. In Figure 5b, we see behaviour of backpressure in underlying system. The backpressure feature prevents the underlying system from exceptions (like memory exception) in runtime. We can see a clear latency drop while scaling from 4- to 8-node. We also can see much less latency peaks. The main reason is that system gets stable with more nodes and more buffers for this use-case. As a result, less peaks in latency happens and data is processed in a flow continuously.
<table>
<thead>
<tr>
<th></th>
<th>DS1</th>
<th>DS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-node cluster</td>
<td>272K</td>
<td>70K</td>
</tr>
<tr>
<td>4-node cluster</td>
<td>400K</td>
<td>112K</td>
</tr>
<tr>
<td>8-node cluster</td>
<td>550K</td>
<td>180K</td>
</tr>
</tbody>
</table>

**Table 1.** Sustainable throughput of SUT. The unit time is 1 second

Table 1 shows the maximum sustainable throughputs of SUT throughout the experiments. We can see approximately 40% increase in sustainable throughputs when scaling out. Table 2 shows the **avg, min, max and quantile (90, 95, 99)** values for latency distributions. We can see that with larger data sets there is more improvement over **avg** latency when scaling-out. Moreover, although there are significantly high max latencies, from quantile values we can see that they can be regarded as outliers.

<table>
<thead>
<tr>
<th></th>
<th>DS1</th>
<th>DS2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>avg</strong></td>
<td><strong>min</strong></td>
</tr>
<tr>
<td>2-node</td>
<td>512</td>
<td>2</td>
</tr>
<tr>
<td>4-node</td>
<td>483</td>
<td>2</td>
</tr>
<tr>
<td>8-node</td>
<td>370</td>
<td>9</td>
</tr>
</tbody>
</table>

**Table 2.** Latency statistics, **avg, min, max, and quantiles (90, 95, 99)** in milliseconds
5 Conclusion

Responding to an increasing use of real-time data processing in industry and in academia, we build a novel framework for benchmarking stream data processing engines and in particular the ones with hybrid data processing capability. Even though there have been several benchmarks of the performance of SDPS recently, these do not measure the latencies and throughput that can be achieved in a production setting. One of the repeating problems in previous evaluations is a missing definition and inaccurate measurement procedure for latency of stateful operators in SDPS. Another challenge is a missing separation between the system under test (SUT) and the benchmark driver. Frequently, the performance metrics are measured and calculated within the SUT; this means that the results will be influenced through the measurements and, thus, can be biased. We identify current challenges in previous work and solve them in our solution. First, we give the definition of latency of a stateful operator and a methodology to measure it. The solution is lightweight and does not require the use of additional systems. Second, we completely separate the systems under test from the driver. Third, we introduce a simple and novel technique to conduct experiments with the highest sustainable workloads. As a result, we can conclude that HDP feature of PROTEUS SDP engine deals with above challenges successfully. Real-time data processing with HDP enabled will empower the manufacturer (ArcelorMittal) to detect the errors in coil production fast and eliminate costs.
6 References


