CSC8499 Individual Project: A Reliable Benchmark Framework for Streaming Systems

Alexandros Rantos-Charisopoulos

MSc in Advanced Computer Science,
School of Computing Science, University of Newcastle, U. K.
b9058710@newcastle.ac.uk

Abstract Much scientific research is being conducted in order to develop and optimise complex systems that can gather and analyse enormous amounts of data. The complexity and technical difficulty of such systems are increasing by the fact that the whole process must happen almost instantly or as close as possible to real-time. For that particular reason, in the last decade, many systems have been developed neglecting, though, the development of a uniform benchmarking suite that can provide valuable feedback on the systems' performance characteristics. Unfortunately, previous attempts are lacking a few vital components that create a reliable and generic benchmark framework. In this thesis, we will investigate the key points of such a framework and set the grounds for future development.

1 Introduction

The importance of data is being highlighted by the fact that public and private enterprises prioritize data over other business assets. A very accurate point, regarding the shift of world's economy towards a data-driven economy, was brought up by The Economist when they published the article *The world's most valuable resource is no longer oil, but data* [1] back in 2017. This profound impact has become the prime concern of every business, thus they have heavily focused on the correct, fault-proof data acquisition and analysis. As a result, large businesses were in need of new complex systems that were capable of efficiently utilizing their produced data; the Big Data.

However, the so-called Big Data is not only a reference to the size of the information. It is mainly described as 3 **V**'s [2]. First comes the **V**olume which, as mentioned before, it relates to the most profound characteristic of Big Data; its vast size and magnitude. The rate that the information is being generated, is described as **V**elocity and also defines the rate that the data must be analysed. Lastly, **V**ariety discusses the diversity and the unstructured nature of the data; as often mentioned in Computer Science as heterogeneity. It is worth mentioning

that, later on, more characteristics were added to the above terminology such as *Veracity* by IBM, Oracle coined *Value* and SAS *Variability* and *Complexity*.

In order to take advantage of the Big Data, new systems have been invented that can process huge amounts of information fast, reliably and continuously. A big breakthrough was made with the discovery of cloud computing technologies [3, 4] that could be hosted in commodity hardware. Firstly, batch systems appeared to solve the above problem. These focused primarily on the throughput of the data, so they analysed the data in intervals which was defined as a certain amount of data. In other words, when the information hits a certain capacity or limit then the processing functions (batch jobs) were automatically fired without human intervention.

Later on, the long processing times of batch systems were no longer serving the purpose of fast data analysis. For this reason, new methodologies were designed that could process a vast amount of information almost instantly without a delay; the streaming systems. Streaming systems are a type of data processing engines that are designed with infinite datasets in mind [5] and has piqued not only the academic but also the business interest over the past years. These systems can provide among other things immediate fraud detection and of course a more immersive experience to the users as they observe live changes.

It is obvious that the aforementioned streaming systems are of utmost importance and the industry is currently in need of a platform that draws insightful comparisons between them. Extracting meaningful insights from raw data is a hard task to achieve due to the nature of Big Data. Moreover, data collection and afterwards data analysis are quite challenging themselves; two new computer science roles created and have a great demand in the last decade; Data Engineer and Data Scientist.

1.1 Aim and objectives

The main goal of this project is to develop a benchmarking suite for the aforementioned streaming systems, since there is a lack of a software that reliably tests their performance in terms of latency, throughput. The related work that has been conducted before, was focused on specific system applications and characteristics, such as CPU and RAM utilisation, rendering them inappropriate to test efficiently a wide variety of systems.

A combination of different technologies will be used for this project and it will be based on previous work that was done by Vasiliki Kalavri et al and was presented in Operating Systems Design and Implementation (OSDI) conference in 2018: DS2 [6]. Briefly, the technologies consist of:

- Flink [7] which will be the streaming system of choice.
- DS2 [6] is an automatic scaling controller for streaming systems that is integrated with Flink.
- finally, a benchmark suite NEXMark [8] of Apache Beam [9] will be used to provide a set of real-life scenarios environment for the system.

All in all, this dissertation will discover the various effects of input rates to the units of work that are being processed by the system, using the previous technologies. In addition, the project will suggest ways of selecting the optimal input arguments based on each purpose and discuss not only their correlation with the output but also the detected patterns that appear.

1.2 Structure of Dissertation

The dissertation is structured as described below:

Section 2: Background and Related Work. This Section will set the background and mention any related work done on the topic.

Section 3: Experimental Enviroment Setup. This Section will introduce the technologies and the systems that were used to undertake the research.

Section 4: Automation Tool. The thorough explanation of the automation tool that was designed and developed throughout the duration of this dissertation, can be found in this Section.

Section 5: Analysis and Outcome Evaluation. In this section, the results of this research will be presented.

Section 6: Conclusion and Future Work. In the end, a summary of the findings will be provided and also future work and improvements will be proposed.

2 Background and Related Work

This section aims to provide the necessary background knowledge on the topic of streaming systems and streaming processing. Furthermore, the technologies used will be discussed and the reasons behind their selection too. Finally, some of the related work on benchmarking will be also presented and briefly discussed.

2.1 Cloud Computing and MapReduce

In order to generate such computing power to process extensive bulk of data and keeping the latency low in a single computer, is impossible. For this reason, scientists consolidated Big Data and Cloud Computing, so they could achieve extraordinary processing results. The latter offers, on-demand, flexible and scalable services which are key for SREs to serve the great traffic fluctuations throughout the day; and the peak holiday periods. Apart from that, the connected network that consists of the cloud plays a vital role in delivering fault-tolerance systems because, in case of a component failure, reserved components take its place. Despite the architecture and the hardware, the computation part was performed by

the MapReduce algorithm which was distributed to many clusters in order to map the input data into key-value pairs and then reduce the values based on the created keys. The parallelization of the algorithm was an easy task because it was based on functional programming principles.

2.2 Streaming Processing Engines

Streaming systems can also be referred to as Streaming Processing Engines (SREs) and they are responsible to process large traffic fluctuations of incoming raw and unbounded data. Time plays a significant factor in their functionality and the way that process data based on the time manifests their efficiency. Generally, in such systems we observe two domains of time:

- **Event Time** which is the actual time that a unit of work needs in order to complete.
- **Processing Time** which is the observed time that the system needs in order to complete a unit of work.

In simpler words, due to external factors such as hardware resources, network latency and others, processing time exceeds event time and creates an asymmetry -skew- if we plot those two variables in a graph (as in Figure 1).

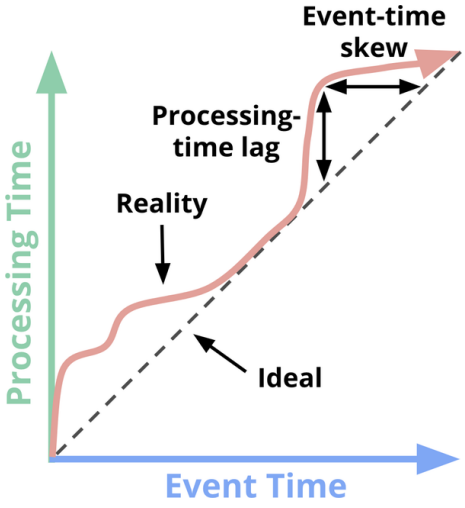


Figure 1: This figure shows the real-world correlation between Event Time (x-axis) and Processing Time (y-axis)). Adapted from Streaming Systems (p. 30), by Tyler Akidau et al [5].

The dashed line in 1 depicts the ideal scenario of processing time not being delayed by any external factor, thus being equal to event time. Also, the vertical

line (Processing-time lag) shows the set back that was caused by a number of factors that explained above and delayed the start of the processing. On the other hand, the horizontal line (referred to as Event-time skew) shows the time that has been actually lost in our pipeline. This creates the need to discuss processing data patterns because our data analysis must be a product of both of these time domains.

2.3 Processing Data Patterns of Streaming Data

Processing streaming data is directly connected to the time that a specific event took place. However, things get more complicated since, as explained in the previous subsection, the event time the system states is not reliable and could have a slight offset from the actual time of the event. Three distinct processing approaches will be to be mentioned below that tackle the aforementioned problem.

Time-Agnostic. Systems that follow this approach, as the name suggests, are not executing their tasks based on event or processing time but based on the arrival of more data; they are data-driven. *Filtering* is a fundamental concept, in Computing Science, which was selected to serve the above purpose, by collecting only the data that has certain properties of interest. The rest of the data can be ignored. Another time-agnostic approach is *Inner Join*, in which we combine two different data streams in order to select the piece of data that holds a certain value from the first stream and another value from the other stream.

Approximation Algorithms. Again another basic concept of Computer Science. It has a very straightforward application even though their implementation is quite a complex task. Their goal is to predict certain types of data based on the rest of the stream. Such algorithms work well with the SREs because they particularly excel when the possibilities of input values are very large.

Windowing. The idea behind this approach is to split a data stream into multiple finite datasets that are ready to be processed. A high level of understanding can be gained by looking at Figure 2. Its most basic form is *Fixed Windows*, in which the size of the window is preset for the whole duration of the processing. Next, we have *Sliding Windows*, an extension of fixed windows, because it slides the fixed window throughout the whole data stream over a fixed time period. Finally, *Sessions* which are a form of dynamic windows. These are purely data-driven and for this reason, they cannot be predefined. In general, windowing is used based on both time domains explained in the previous subsection.

2.4 Metrics

The main goal of streaming systems is to achieve the best combination of latency and throughput for each case. Generally, preserving the lowest possible latency while keeping the throughput at the maximum rate is the ideal. These are the main metrics used for evaluation the capability and performance of the systems.

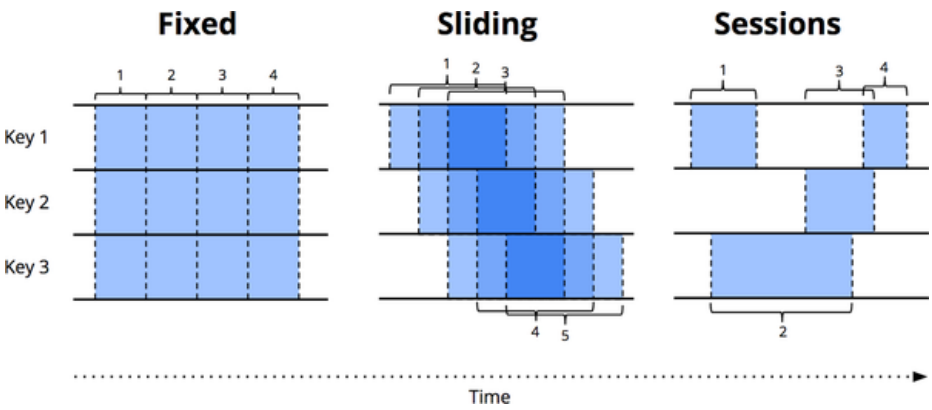


Figure 2: This figure depicts the different types of windowing. Adapted from Streaming Systems (p. 39), by Tyler Akidau et al [5].

Latency. This metric describes the time elapsed between the arrival of a unit of work till the end of its processing. It is hard to provide an accurate number for this metric in a distributed environment because the clock speeds are different per machine and other external factors such as resources utilization, I/O and network usage might play a role affecting the validity of the measurement. Extra caution is required to remove such "noise" when measuring latency.

Throughput. Generally, throughput is the average count of data processed per a defined time interval. Each system defines the data differently; it might be a single record or a fixed size of bytes. The same applies for time; it might be second or a fixed frequency or time period.

2.5 Related Work

The very first work was initiated by introducing *MediaBench* [10], a tool that was evaluating and synthesizing communications and multimedia systems. Later, *StreamBench* [11] introduced four different workload-suites to test different systems' characteristics and defined new metrics apart for the performance related ones (latency & throughput) for these experiments; these related to fault tolerance and they are throughput penalty factor (TPF) and latency penalty factor (LPF). In 2016 Yahoo! entered the field by publishing a set of experiments measuring the latency and throughput on three different Apache projects; Flink, SPARK and Storm [12]. They depended on two external technologies. Firstly on Apache Kafka [13] to read events and secondly, on Redis [14] to store the computed windows of data. However, later it was found by J. Karimov et al [15] that these technologies were a bottleneck for the benchmark system. They proposed a new benchmark framework that generated data from a custom generator and

they did not store the results as key-value pairs, replacing in that way the above technologies.

As can be seen, there is no framework that covers all scenarios. Each framework focuses on certain aspects of systems and uses specific cases at which it performs wells. Most previous studies also investigated certain SREs such as Apache SPARK, Apache Flink and Storm. However, last year a new study was released that was covering a wider spectrum of such systems [16] like Apache Heron, Kafka Streams and Hazelcast Jet.

In summary, this dissertation will evaluate the effect of input source rates and the operators' parallelism on the latency and throughput of streaming systems.

3 Experimental Enviroment Setup

Using Docker [17], a software that creates containers in which you can run your application at any Operation System, helped a lot with setting up the experimental system. An already created Dockerfile was used that was running Debian with Java and Rust, so the technologies that will be discussed below could run. However, few tweaks were made in the Dockerfile in order to execute the python scripts and their dependencies needed for the Automation Tool.

3.1 Flink [7]

This the processing engine that was used for our experiments and has native support for streaming, hence providing low-latency and high throughput. Flink is highly customizable, because of its DAG architecture, and developers can chain multiple custom processing functions and/or alter the order of the data pipeline. It can also support batch processing. Generally, it became famous for its ability of processing data stream in real-time. Its architecture resembles Hadoop MapReduce [18]; it is a master-slave distributed cloud system. The master is the JobManager Node and the workers/slaves are the TaskManagers. The Flink web dashboard can be seen in figure 3 while running Query2 of NEXMark.

3.2 DS2 [6]

DS2 is an automatic controller for streaming systems that estimates the true processing and output rates for each operator of the system and vary the systems' resources dynamically in order to avoid a streaming job to become over or under-provisioned. Instead of relying on external metrics such as CPU or memory utilization, it predicts how much and when to scale based on the true processing and output potential of each operator and the effect on the rest operators, so the system can meet the throughput SLO (Service Level Objective) which is the ideal throughput the service provider wants to reach. DS2 model defines three new terms on which our experiments will be based and lead by:

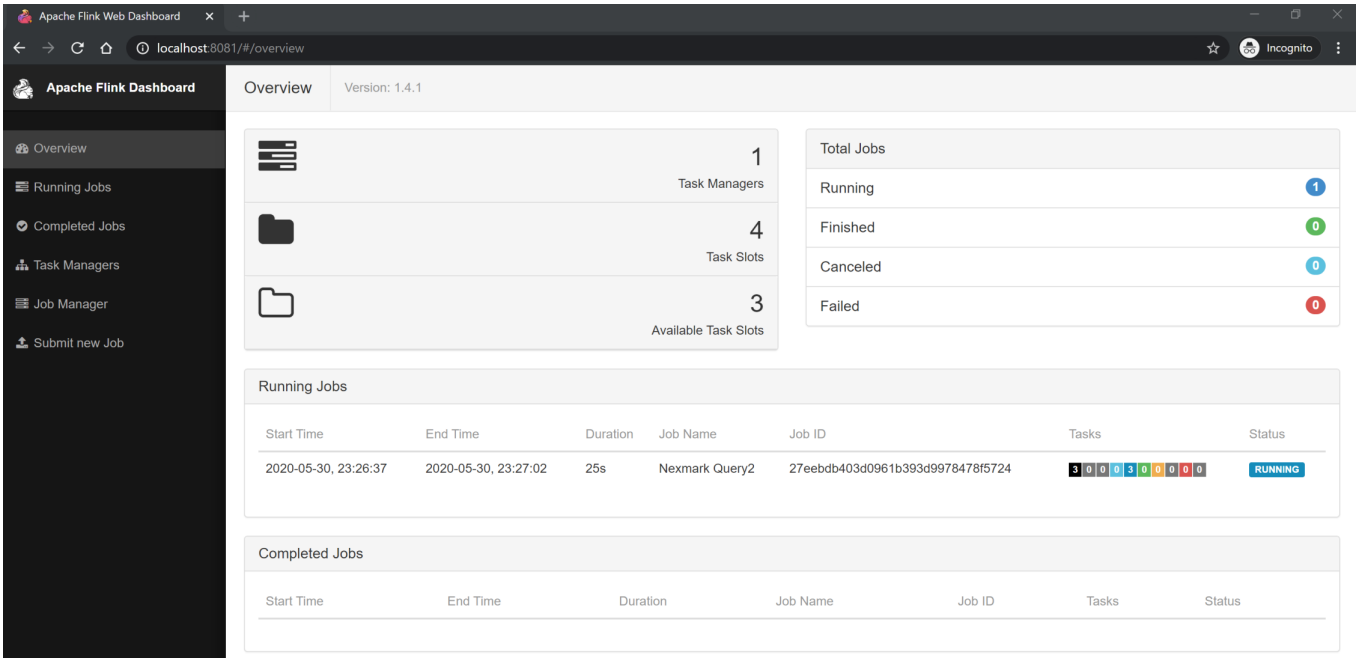


Figure 3: Screenshot of Apache Flink Web Dashboard running NEXMark Query2

- **Useful Time** is the notion of time spent only in useful operations for the system (deserialization, processing and serialization). In other words, it excludes the waiting time of any external factors such as resource bottlenecks. We also define the time metric that includes all the waiting times as **observed time**.
- **True Processing/Output Rate** the units of work processed in a unit of useful time (**ut**). It denotes the true work, in terms of processing and output rate, done by the system when the resources were allocated only for that job.
- **Observed Processing/Output Rate** the units of work processed in a unit of observed time (**ot**). It simply represents the amount of records that have been processed or output without excluding any unnecessary waiting time.

DS2 patch. In order for Flink to output the above metrics an already developed DS2 patch for Flink was used. This patch makes use of a *MetricsManager* that aggregates the measurements to reduce the overhead of Flink's output policy.

3.3 Beam [9]

In the need of a uniform model for batch and streaming systems, Beam was developed. It provides a high-level API that creates data pipelines that can be

executed by the processing engine of choice. Two are its main terms *PCollections* and *PTransforms*. The former represents the datasets which can be bound or unbounded data and the latter represents the transforming functions that given a collection of input, generates a collection of output. In addition, *PTransforms*, as seen in Figure 4, consists of:

- **ParDo** which is the map function of Beam and applies a specified function to every element.
- **GroupByKey** which is similar to reduce and groups the elements based on their key and then applies a function on each group.
- **Composite** which uses a combination of the above to produce an output.

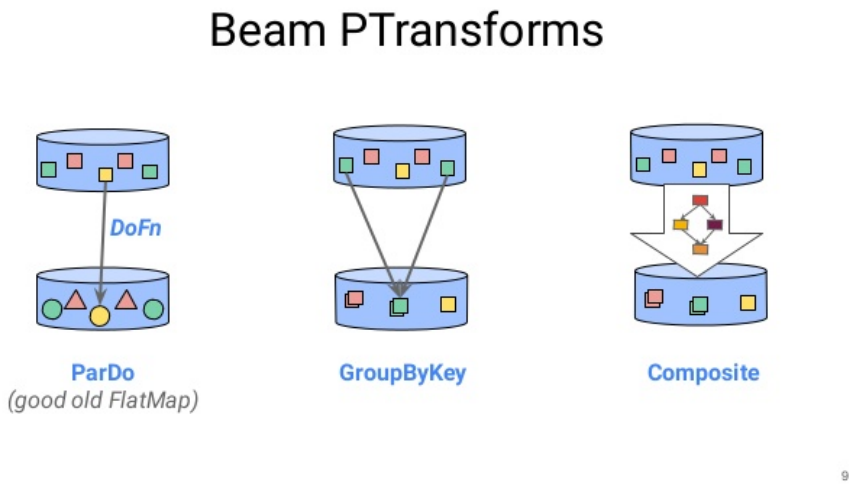


Figure 4: The different types of PTransforms in the Beam model. Retrieved from the presentation given by Etienne Chauchot, an Apache Beam Committer, [19].

In the programming model of Beam the data pipeline is represented by a Directed Acyclic Graph (DAG), where each node of the graph is a transform function. The starting node (the one with no ingoing edges) is called a source and the last node (the one with no outgoing edges) is called a sink. The former is responsible to read the data, while the latter to write the output (e.g. to a file). They are I/O related nodes and in Beam mode, they are composite transforms.

3.4 NEXMark [8]

NEXMark (Niagara Extension XMark) is an extension to XMark which was a XML benchmark project. NEXMark is a benchmark framework that simulates

real-world processes of data streaming on the notion of an online auction system. It consists of multiple queries that expose streaming systems to different scenarios. All queries have three main entities:

- **Auction** represents a single item that is under auction.
- **Bid** represents a bid for a specific item that has been submitted for auction.
- **Person** represents the entity that is able to either submit an item or place a bid for auction.

In the context of this dissertation, the following queries will be used; Table 1 shows the actual SQL code of each query:

- **Query1** represents a Currency Conversion which in our terms translates to a simple Map function.
- **Query2** is a simple selection of auction with a particular number. This is a filter function.
- **Query3** returns all the persons who are selling an item in US states. This demonstrates an incremental join and filter.
- **Query5** returns every minute the item which has seen the most bids in the last hour. This is a sliding window operation.
- **Query8** monitors the new users since it selects all the users who submit an item the next 12 hours after they register to the auction. As Query5, this is a sliding window operation followed by a join operation.
- **Query11** selects all the bids that a user has placed in each of his/her active sessions. Demonstrates a session window operation.

Table 1: The SQL code of the NEXMark Queries.

QueryID	**SQL code**
Query1	SELECT auction, DOLTOEUR(price), bidder, datetime;
Query2	SELECT Rstream(auction, price) FROM Bid [NOW] WHERE auction = 1007 OR auction = 1020 OR auction = 2001 OR auction = 2019 OR auction = 2087;
Query3	SELECT Istream(P.name, P.city, P.state, A.id) FROM Auction A [ROWS UNBOUNDED], Person P [ROWS UNBOUNDED] WHERE A.seller = P.id AND (P.state = 'OR' OR P.state = 'ID' OR P.state = 'CA');
Query5	SELECT B1.auction, count(*) AS num FROM Bid [RANGE 60 MINUTE SLIDE 1 MINUTE] B1 GROUP BY B1.auction;
Query8	SELECT Rstream(P.id, P.name, A.reserve) FROM Person [RANGE 1 HOUR] P, Auction [RANGE 1 HOUR] A WHERE P.id = A.seller;

Unfortunately, due to technical difficulties, we removed Query3, Query3Stateful, which was and Query8 from our experiments. The reasoning behind this action

is briefly explained in the last section where we discuss the Future Work that can be done.

4 Automation Tool

In order to run a single Query the following steps had to be performed manually:

1. Set the required build and jar paths for Flink.
2. Configure the input rates and parallelism value of each operator.
3. Run the task through Flink in the background.

However, running multiple tests on 7 different queries was not a viable option. Inspiration was drawn by an already developed bash script that had automated a part of the work that needed to execute the *wordcount* Query. Specifically, the script was configuring the paths for Flink and parsing the input so it could set the corresponding parallelism for each operator. It was initiated by the following line:

```
./start-wordcount.sh "Source: Custom Source",1 \
                     #"Splitter FlatMap",1 \
                     #"Count -> Latency Sink",1 \
```

where between the quotes (“”) it is the name of the operator followed by a comma and its parallelism value. The hash symbol (#) is serving the purpose of the delimiter. By running the above command each mentioned operator was set to a single instance or alternatively parallelism value equal to 1. However, it is worth noted that increasing the parallelism value too much without having the sufficient available task slots in Flink, would result in an error due to the limited resources of the system. Furthermore, the DS2 patch for Flink writes its output on logs file and more specifically writes one line per file. For this reason, running multiple Queries simultaneously was not viable because there was no way to distinguish which Query produce which log file and if we did so, the results would be mixed and unreliable.

The goal was to fully automate the above process that was explained. For this reason, a combination of Python and Bash scripts were developed in order to:

1. run a set of Queries with dynamically changing inputs rates and parallelism values, as can be seen in Section 4.1.
2. monitor the output directory and save the results in the corresponding CSV file; Section 4.2.
3. analyse the results by creating helpful and insightful plots which is discussed in the last Section 4.3.

4.1 Query Generator

This class is responsible for generating sequentially a set of queries and “feeds” them to our system. To achieve this, we define a dictionary with starting values for each input argument of our query, as can been seen below:

```
QUERY_ARGS = {
"Query1": {"exRate":0.8, "srcRate":40000, "psource":1, "pmap":1},
"Query2": {"srcRate":40000, "psource":1, pmap":1},
"Query3": {"auctionRate":15000, "personRate":8000, "pAuctionSource":1,
           "pPersonSource":1, pjoin":1},
"Query3Stateful": {"auctionRate":15000, "personRate":8000,
           "pAuctionSource":1, "pPersonSource":1, "pjoin":1},
"Query5": {"srcRate":40000, "psource":1, pmap":1},
"Query8": {"auctionRate":15000, "personRate":8000, "pAuctionSource":1,
           "pPersonSource":1, "pjoin":1},
"Query11": {"srcRate":40000, "psource":1, "pmap":1},
}
```

We define a similar second dictionary which holds the values which we add to the previously mentioned dictionary, so we produce a different set of arguments for the running Query. To differentiate the value ranges of the input source rates with the operator's parallelism we define two variables (*RATE_ITERATIONS* and *PARALLELISM_ITERATIONS*) that control the times that each argument can increase. One rate iteration means that only one rate value has increased in the arguments, while one parallelism iteration means that the parallelism value of one of the operators has increased. However, for every parallelism iteration, multiple rate iterations will happen in order to generate different input arguments for the given parallelism combination. In general, the number of different queries we generate is equal to the product of these variables.

There are two main methods at which we generate the different arguments for each Query and it differs in the way we increase the parallelism operators each repetition:

1. Firstly, we increase one operator parallelism at a time while keeping the rest equal to their base value which is set to 1 5a.
2. Secondly, we increase uniformly all parallelism operators per repetition 5b.

Then, the generated queries from Figure 5 are given as input arguments to the bash script which is responsible of configuring the right paths for Flink, executing the given Query and also to fire the monitor script which we will discuss in the next section.

4.2 Polling and Storing Results

As was mentioned before, Flink writes its output log files in the directory of our preference. We opted to monitor that specified directory for any changes, and more specifically for any file creations, using the API library for python *Watchdog*. Every time this python package detects a change, it sends the file path to the database class that is responsible for storing all the results for the corresponding Query. The polling stops after two minutes of inactivity (no modification in the directory) and the script runs again the query with the next set of input arguments. Ideally, we wanted to use a cloud database since the system

```
srcRate,40000#psource,1#pmap,1
srcRate,80000#psource,1#pmap,1
srcRate,120000#psource,1#pmap,1
srcRate,160000#psource,1#pmap,1
srcRate,200000#psource,1#pmap,1
srcRate,40000#psource,2#pmap,2
srcRate,80000#psource,2#pmap,2
srcRate,120000#psource,2#pmap,2
srcRate,160000#psource,2#pmap,2
srcRate,200000#psource,2#pmap,2
srcRate,40000#psource,3#pmap,3
srcRate,80000#psource,3#pmap,3
srcRate,120000#psource,3#pmap,3
srcRate,160000#psource,3#pmap,3
srcRate,200000#psource,3#pmap,3
```

(a) Increasing all operators' parallelism per iteration

```
srcRate,40000#psource,1#pmap,1
srcRate,80000#psource,1#pmap,1
srcRate,120000#psource,1#pmap,1
srcRate,160000#psource,1#pmap,1
srcRate,200000#psource,1#pmap,1
srcRate,40000#psource,2#pmap,1
srcRate,80000#psource,2#pmap,1
srcRate,120000#psource,2#pmap,1
srcRate,160000#psource,2#pmap,1
srcRate,200000#psource,2#pmap,1
srcRate,40000#psource,3#pmap,1
srcRate,80000#psource,3#pmap,1
srcRate,120000#psource,3#pmap,1
srcRate,160000#psource,3#pmap,1
srcRate,200000#psource,3#pmap,1
srcRate,40000#psource,1#pmap,2
srcRate,80000#psource,1#pmap,2
srcRate,120000#psource,1#pmap,2
srcRate,160000#psource,1#pmap,2
srcRate,200000#psource,1#pmap,2
srcRate,40000#psource,1#pmap,3
srcRate,80000#psource,1#pmap,3
srcRate,120000#psource,1#pmap,3
srcRate,160000#psource,1#pmap,3
srcRate,200000#psource,1#pmap,3
```

(b) Increasing one operator's parallelism per iteration

Figure 5: This figure depicts the different sets of generated queries for Query2: $RATE\_ITERATIONS = PARALLELISM\_ITERATIONS = 4$

was deployed to multiple machines; however, we selected a very basic solution; to store the results in CSV files so it is easier for analysis with *pandas* framework. Git was also used for version control of this simplistic database. Apart from the output which DS2 produce that was mentioned in §3.2 we saved the input arguments that generated these results and the file name itself so we could identify the patterns based on the input arguments. An example of the stored values in the CSV files can be seen in Figure 2 when loaded into a dataframe using pandas framework. An explanation of each column follows:

- **operator** is the name of the operator that the row corresponds to.
- **id** is the unique number that relates to each operator instance. It starts from 0 and increases every time Flink outputs a new log file for that specific operator instance.

- **Exchange/Source Rate** the corresponding input rates values of the query that produced the results. These columns defer per query.
- **BidSource/PMap/LSink** the corresponding parallelism operator's values of the query. These columns defer per query.
- **op_instance_id** the operator instance.
- **total_op_instance** the total number of instances for the corresponding operator.
- **epoch_timestamp** this is
- **true_proc_rate** the aggregated true processing rate of the operator for the given timestamp. For some operators, it can be zero.
- **true_output_rate** the aggregated true output rate of the operator for the given timestamp. For some operators, it can be zero.
- **observed_proc_rate** the aggregated observed processing rate of the operator for the given timestamp. For some operators, it can be zero.
- **observed_output_rate** the aggregated observed output rate of the operator for the given timestamp. For some operators, it can be zero.

Table 2: Query1 stored values.
(Float values were rounded to three decimal places)

operator	id	exRate	srcRate	BidSource	PMap/LSink	op_instance_id	total_op_instances	epoch_timestamp	true_proc_rate	true_output_rate	observed_proc_rate	observed_output_rate
Source: Bids Source	0	0.8	40000	1	1	1	1	4919255864317175	0	43127.052	0	39792.979
Mapper	0	0.8	40000	1	1	1	1	4919255856315617	73754.032	73754.584	39398.341	39398.636
Latency Sink	0	0.8	40000	1	1	1	1	4919255862151575	257437.061	0.0	39424.940	0.0
Source: Bids Source	1	0.8	40000	1	1	1	1	4919266007030774	0	43707.027	0	40000.120
Mapper	1	0.8	40000	1	1	1	1	4919266025301793	76370.668	76371.049	40029.004	40029.204
Latency Sink	1	0.8	40000	1	1	1	1	4919266044636872	281685.476	0.0	40066.471	0.0

4.3 Data Analysis

The first approach was to analyse the impact of increasing each input rate on the observed and true processing rate and output. This was done by calculating the mean of the aforementioned output metrics from the piece of data that was

selected by keeping increasing the input rate of concern while keeping the rest at their starting values. However, this approach was found invalid because in data streams we do not care about mean values of output rates but rather the full spectrum of our data. This is because we can extract valuable insight from the valleys and peaks of a graph in such cases. For this reason, we compute the empirical cumulative distribution function (ECDF), which helps us depict all the collected data points based on their cumulative distribution. In simpler words, the x-axis represents the values of the input rate we analyse and the y-axis shows the cumulative probability of the specified value; for example, if the cumulative probability is 0.6 it means that 60% of our data produces output rate less and equal to the specified value at x-axis. In conclusion, an *Analyzer* class has been developed that reads a CSV file and creates a set of ECDF plots. A report of the implementation details is included in the following Section.

4.4 Settings

For the sake of expediency, a settings.py file was created. Via this file the following variables can be modified:

- The starting values of each Query.
- The value range of input variables by altering the maximum repetition at which the variables increase.
- The values that will be added to each Query's argument each run.
- The various directory paths such as the path that is being monitored, the storage path for the CSV files and others.
- The valid Query's names for the current version of our system.

4.5 Instrumentation Deployment

Each of the above was developed as a separate module. However, all are essential for our lightweight instrumentation tool. Figure 6 shows the dependencies of each module.

We simplified its initiation by creating a simplistic queue in which we store the Queries we want to generate via our generator. It is based on the fundamental fork system call of UNIX which is an operation that creates other subprocesses called child processes from the main process (parent process). This is mostly used in parallel computing, as the parent process creates subprocesses to take care of small tasks that could be run in parallel while the parent process continues its own computations. However, in our case, we stop the parent process until all of its subprocesses finish thus creating a queue. The overly simplified code for our queue:

```
import subprocess

subprocess.call(["python3", "generate_queries.py", "Query1",
                 "equallyIncreasingOperatorsv1"])
```

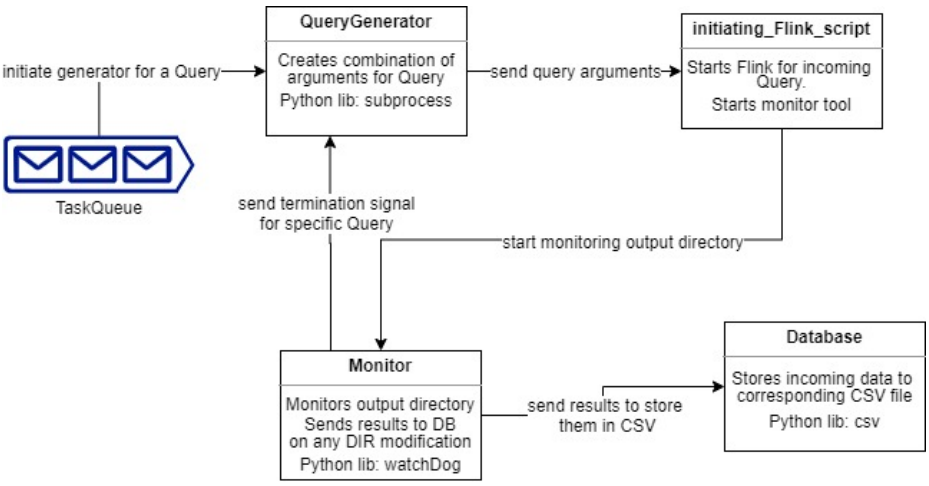


Figure 6: Dependencies of each module in our instrumentation tool. Diagram was created by using an online diagram maker *Flowchart Maker & Online Diagram Software* [20]

```
subprocess.call(["python3", "generate_queries.py", "Query1",
                 "metricsv1"])
subprocess.call(["python3", "generate_queries.py", "Query2",
                 "equallyIncreasingOperatorsv1"])
subprocess.call(["python3", "generate_queries.py", "Query2",
                 "metricsv1"])
```

We found *subprocess* is the most reliable python module to securely create new processes without any problems and we used it every time we wanted to spawn a process to perform a subtask as shown in 6. In addition, the last argument specify which generating function QueryGenerator has to initiate for the relating Query and is also appending that string to the file name of the CSV that is being created.

5 Analysis and Outcome Evaluation

The following evaluation covers only Flink with the DS2 patch in order to output the metrics that were discussed in §3.2. All experiments were run remotely on a single machine, that was provided by Newcastle University, with 2 Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz and 16GB RAM, running Ubuntu GNU/Linux 4.15.0-106-generic [21]. The Flink version that was used is Apache Flink 1.4.1 configured with 1 Task Manager which had 4 Task Slots, as seen in Figure 3.

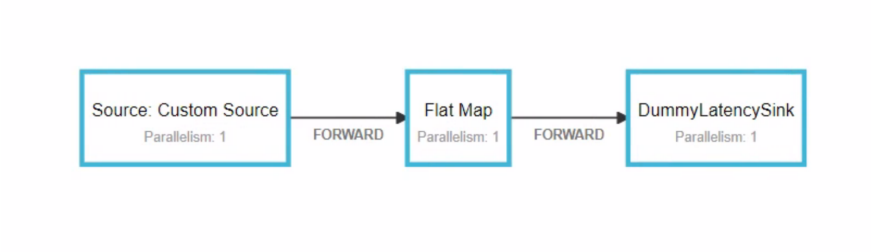
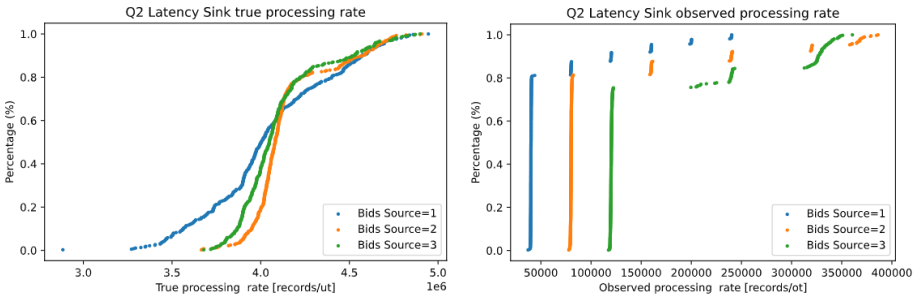


Figure 7: Data pipeline of NEXMark Query2. Screenshot was taken from the Apache Web Dashboard.

5.1 Data exploration

The decision behind storing the input values for each output was made because there was no data that could help us judge if different parallelism value of previous operators in the data pipeline could affect the current operator we are analysing. For demonstration purposes, the data pipeline of Query 2 is given in Figure 7. For example, we would like to explore if and by how much *Dummy Latency Sink* or *FlatMap* operator is affected by the parallelism value of *Source: Custom Source*. *Dummy Latency Sink* and *FlatMap* instances are exposed under the same variable in the source file so we could not explore the impact of the parallelism value between them; we leave this to future work. This was a new perspective we were able to explore by adding that piece of information into our dataset. It turned out this was a valid concern because in Figure 8 we observe an overall increase of both true and observed processing rate of *(Dummy) Latency Sink* by just adding more instances to *Source: Custom Source operator*. Based on DS2, the true processing (resp. output) rate relates to the maximum capacity of the operator which on numerous occasions the system cannot reach. However, the "real" rates of the system are represented by the observed rates. For these reasons, Figure 8b shows that, indeed, the instances of *Source: Custom Source operator* substantially affect the observed processing rate, even though Figure 8a depicts a small impact especially on the 60% of the true processing rate values. A simple interpretation of this behaviour could be that since there are more records produced by both of the instances of Custom Source operator *FlatMap* and *Latency Sink* operators have a greater input to process.

Another aspect we try to explore in the performance of our system is the effect of each input rate on each operator. In circumstances of multiple operator instances, we aggregate the processing and output rates of each parallel instance of that particular operator. This is done because in the case of parallel instances the workload is equally distributed among the total number of operator instances. In other systems this is a complex task as the only way to distinguish the "stage" of each parallel operator is by the timestamp. However, in our system, DS2 helps us to identify to which instances the same dataset was distributed by checking the unique *id*, which was mentioned in §4.2, of each



(a) Latency Sink true processing rate. (b) Latency Sink observed processing rate.

Figure 8: An ECDF plot of Query2 Latency Sink operator's different processing rate. The various scatter plots represent the cumulative distribution of each processing rate while we increase the parallelism of Bids Source operator. The legend shows the parallelism value of Bids Source.

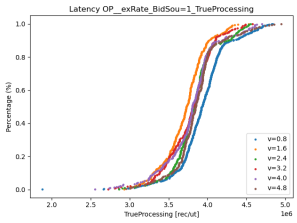
instance. Since we evaluate the operator as a single unit we group the various operator instances with the same *id*.

5.2 Experimental Results: Rates Effect

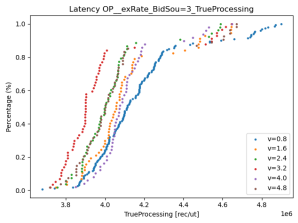
In this subsection, we provide our findings concerning the effect of input rates on the processing and output rate of each operator.

Query1. Figure 9 shows the effect of *exchange rate* and *source rate* on the three different operators (*Source, Mapper, Latency Sink*). We concluded that *exchange rate* is inversely proportional to the processing and output rate of all operators regardless their parallelism. In addition, we observed that increasing the parallelism value of the *Source* operator did not have any significant impact on the processing rate of Latency Sink (see Figure 9a & 9b) or on the true output rate of Mapper (see Figure 9g & 9h). This might be because the maximum capacity of the operators does not significantly change regarding the *exchange rate*. However, increasing the parallelism of Mapper and Latency operators tripled the true processing and output rate when we added two more instances to each operator in contrast to when we only had a single one (Figure 9c & 9d and 9e & 9f). A completely different pattern can be seen when looking at the observed output rates and it applies for all operators. The remaining Figures (9i, 9j, 9k and 9l) shows that if we increase the lowest *exchange rate* (0.8), the observed output rate "locks" to a very limited range of output values and generally the overall throughput is far lesser.

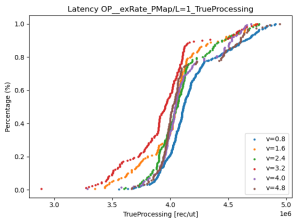
On the other hand, *source rate* had the opposite effect on the operators rates as can be seen in Figure 10. The biggest impact was observed on the true output rate of *source* operator (Figure 10e). Comparing it with Figure 10f we also observe that the parallelism value of the operators that are after our



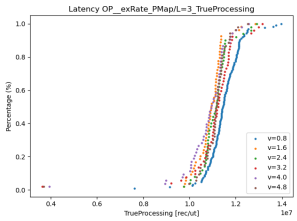
(a) Latency Sink true processing rate with SourceP = 1.



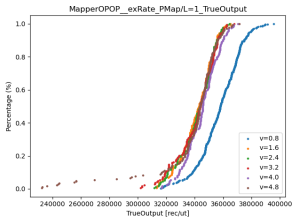
(b) Latency Sink true processing rate with SourceP = 3.



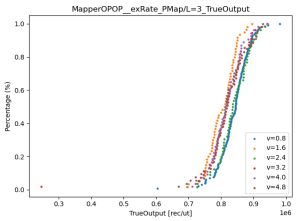
(c) Latency Sink true processing rate with MapperP = LatencyP = 1.



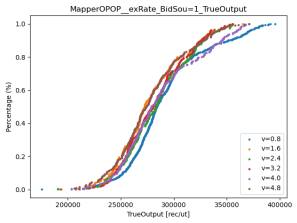
(d) Latency Sink true processing rate with MapperP = LatencyP = 3.



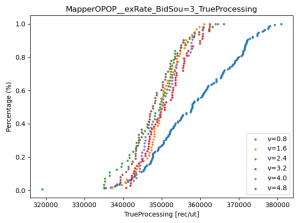
(e) Latency Sink true processing rate with MapperP = LatencyP = 1.



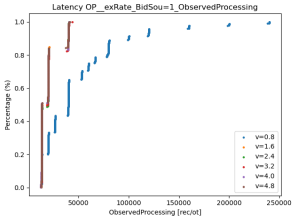
(f) Mapper true output rate with MapperP = LatencyP = 3.



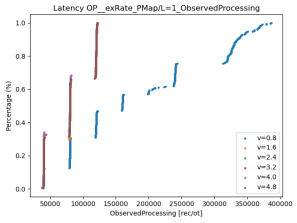
(g) Mapper true output rate with SourceP = 1.



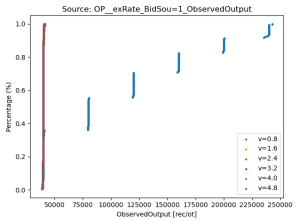
(h) Mapper true output rate with SourceP = 3.



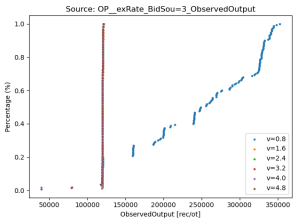
(i) Latency Sink observed output rate with SourceP = 1.



(j) Latency Sink observed output rate with MapperP = LatencyP = 1.



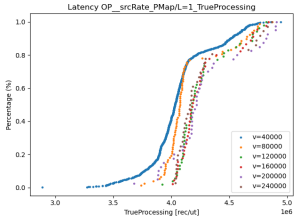
(k) Source observed output rate with SourceP = 1.



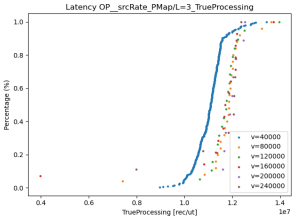
(l) Source observed output rate with SourceP = 3.

Figure 9: Query1's effect of increasing exchange rate on processing and output rates of each operator. The legend shows the values of exchange rate.

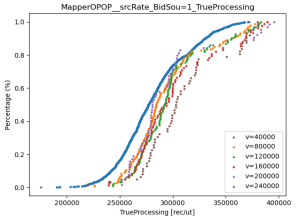
current operator in the data pipeline is not impacting its true capacity. Looking to the observed output rates in the case of *Source* operator we see that when the parallelism is equal to one the graphs look very similar (Figures 10f and 10g) but when parallelism increases to three, the values *source rate* greater than $120,000$ stops increasing the observed output rates (Figure 10h). Also, we observed that *Mapper* observed output (resp. processing) rates and *Latency Sink* observed processing rate produced the same identical graphs regardless of the input rates and operator parallelism.



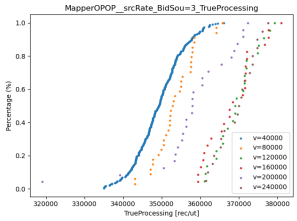
(a) Latency Sink true processing rate with MapperP = = LatencyP = 1.



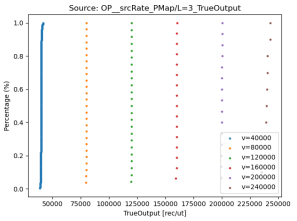
(b) Latency Sink true processing rate with MapperP = = LatencyP = 3.



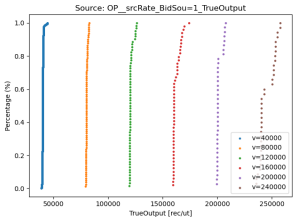
(c) Mapper true processing rate with MapperP = = LatencyP = 1.



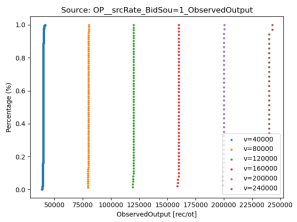
(d) Mapper true processing rate with MapperP = = LatencyP = 3.



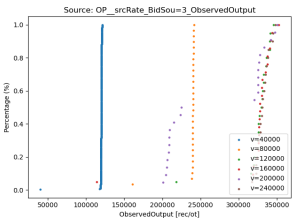
(e) Source true output rate with MapperP = LatencyP = = 3.



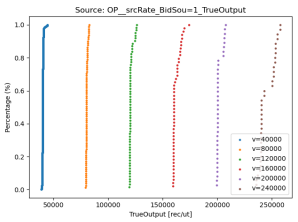
(f) Source true output rate with SourceP = 1.



(g) Source observed output rate with SourceP = 1.



(h) Source observed output rate with SourceP = 3.



(i) Source true output rate with SourceP = 1

Figure 10: Query1's effect of increasing source rate on processing and output rates of each operator. The legend shows the values of source rate.

Query2. Likewise, *source rate* had the same overall impact as in Query1. Figure 11 shows that *source rate* raise the true (resp. observer) processing and out rates. At high values of *source rate* the performance boost seems to degrade in *Latency Sink*, especially when we increase the instances of any operator (Figure 11b). The same scatter pattern was observed for *Flat Map* as increasing the parallelism of the operators made the scatter of high values of *source rate* converge. We provide only the graphs with parallelism value equal to 1 in Figures 11e, 11f, 11g and 11h. Lastly, all graphs related to *Source* operator has the same pattern as Figure 11i, and by only increasing the parallelism of the same operator we see an increase of 100% at each instance we add.

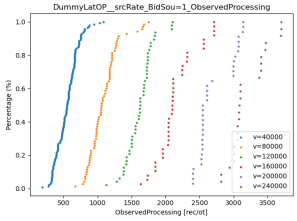
Query5. In general, we found that *source rate* did not play a role in the performance of the system, as the majority of scatter plots are overlapping (Figure 12). Even the plots for true rates are almost identical with the graphs that corresponds to the observed rates. Nonetheless, an important comment to make is that in most of the Figures (except Figure 12l) the tail of the graphs shows that approximately lowest 10% of the rates are produced by either the highest *source rate* $(240,000)$ or the lowest $(40,000)$.

Query11. There are some controversial findings for this query in Figure 13. Firstly, we see in Figure 13a the scatter subplots for values of *source rate* greater than $40,000$, overlap; however in Figure 13c we observe that the subplots of the true processing rates do not have that behaviour and in fact the minimum value of *source rate* yields the best results and they are in a way inverted. However, this behaviour is lost when we increased the instances of *Source* operator and we got the same plots (Figures 13b & 13d) regardless if the processing rate was based on useful or observed time. The same pattern can be found in the next three Figures (13e,13f and 13g) but here we start with the same graph when the Session Window parallelism is equal to one (13e) and when we increase it to three the observed processing rate plot (13f) shows that the minimum value *source rate* has the worst rate but when we observer (13g) we see that the same value has the best rate. The remaining Figures (13h,13i,13j,13k,13l) shows the difference between observed and true output rate on *Source* operator while we increase the instances of that operator.

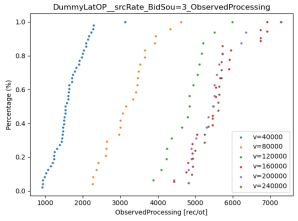
5.3 Experimental Results: Parallelism Effect

Here, we present the results of our analysis regarding the effect of increasing the parallelism value of all operators at once on the processing and output rate of all the system's operators.

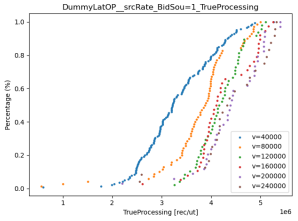
Query1. Generally, we do observe a improvement on performance while using multiple instances on all operators in Figure 14. However, in most plots the orange and green line converges showing that the maximum processing and output rates do not increase while increasing the parallelism value. Finally, the



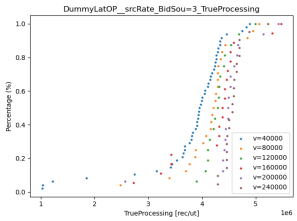
(a) Latency Sink observed processing rate with SourceP = 1.



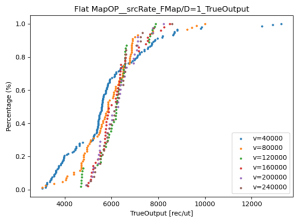
(b) Latency Sink observed processing rate with SourceP = 3.



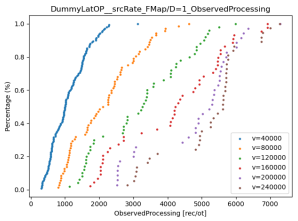
(c) Latency Sink true processing rate with SourceP = 1.



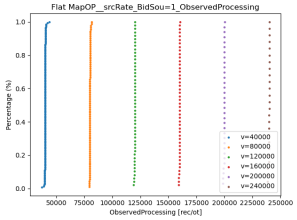
(d) Latency Sink true proc. rate with SourceP = 3.



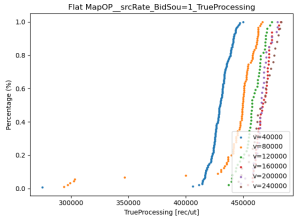
(e) Flat Map true output rate with FlatMapP = 1.



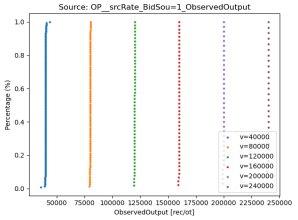
(f) Flat Map true output rate with FlatMapP = 1



(g) Flat Map observed proc rate with SourceP = 1.



(h) Flat Map true proc. rate with SourceP = 1.

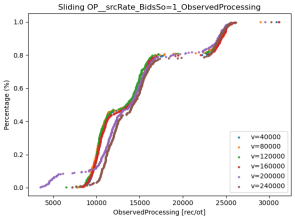


(i) Source observed output rate with SourceP = 1

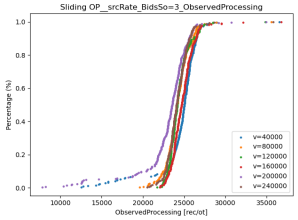
Figure 11: Query2's effect of increasing source rate on processing and output rates of each operator. The legend shows the values of source rate.

statistical dispersion of some graphs are identical (14c with 14d & 14e with 14f) which means that there is no difference between observed and true rates.

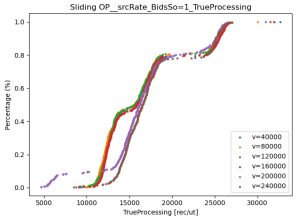
Query2. Figures 15g and 15h shows the importance of the notion of useful time and true processing and output rate we introduced in §3.2. We could have draw the wrong conclusion by just looking 15g since the highest parallelism value yields the worst results; however, this is not the case when looking over at 15h. In general, we see an increase in performance as the parallel instances of all operators are increasing but the performance boost is falling off while the number of instances is going up.



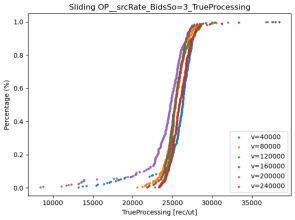
(a) Sliding Windows observed processing rate with SourceP = 1.



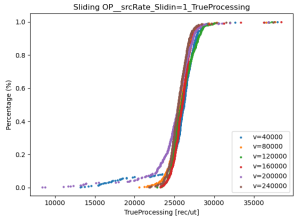
(b) Sliding Windows observed processing rate with SourceP = 3.



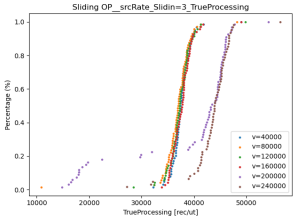
(c) Sliding Windows true processing rate with SourceP = 1.



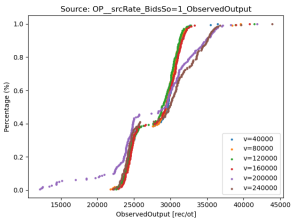
(d) Sliding Windows observed processing rate with SourceP = 3.



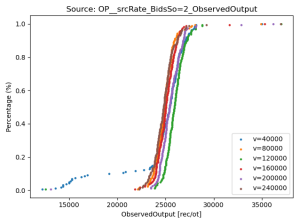
(e) Sliding Window true processing rate with SlidingWP = 1.



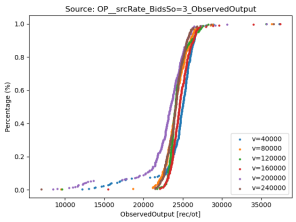
(f) Sliding Window true processing rate with SlidingWP = 3.



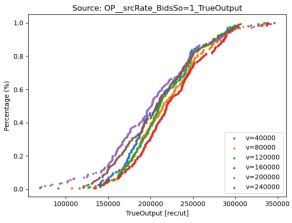
(g) Source observed proc. rate with SourceP = 1.



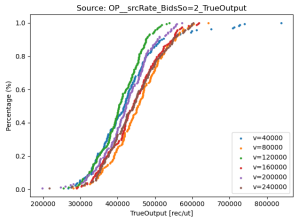
(h) Source observed proc. rate with SourceP = 2.



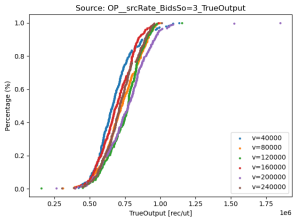
(i) Source observed proc. rate with SourceP = 3.



(j) Source true output rate with SourceP = 1.

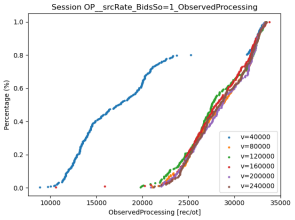


(k) Source true output rate with SourceP = 2.

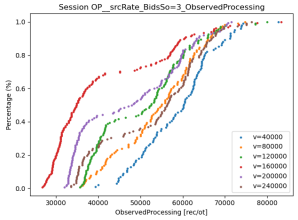


(l) Source true output rate with SourceP = 3.

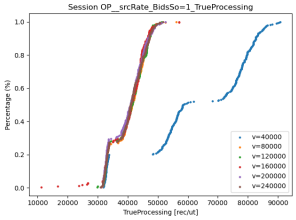
Figure 12: Query5's effect of increasing source rate on processing and output rates of each operator. The legend shows the values of source rate.



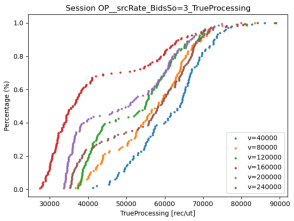
(a) Session Window observed proc. rate with SourceP = 1.



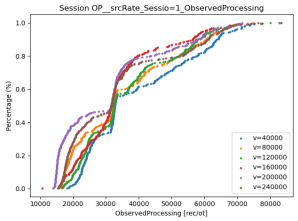
(b) Session Window observed proc. rate with SourceP = 3.



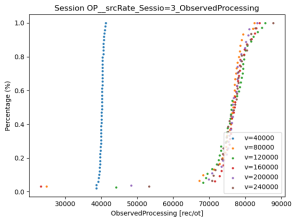
(c) Session Window true proc. rate with SourceP = 1.



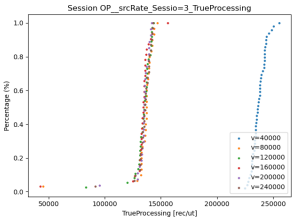
(d) Session Window observed proc. rate with SourceP = 3.



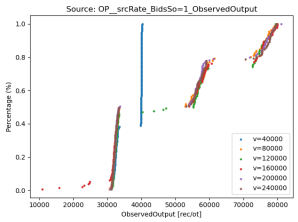
(e) Session Window observed proc. rate with SessionWP = 1.



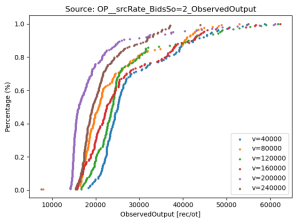
(f) Session Window observed proc. rate with SessionWP = 3.



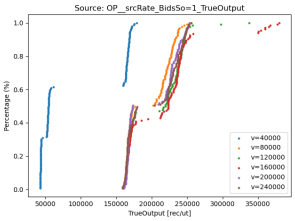
(g) Session Window true proc. rate with SessionWP = 3.



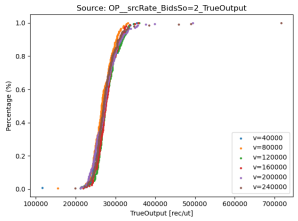
(h) Source observed output rate with SourceP = 1.



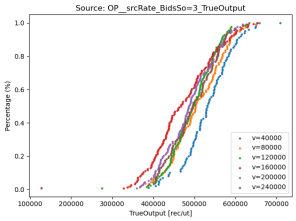
(i) Source observed output rate with SourceP = 2.



(j) Source true output rate with SourceP = 1.

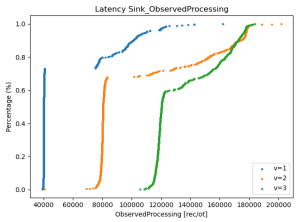


(k) Source true output rate with SourceP = 2.

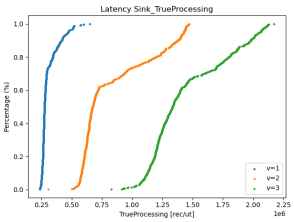


(l) Source true output rate with SourceP = 3.

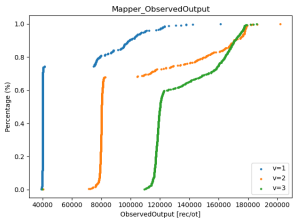
Figure 13: Query11's effect of increasing source rate on processing and output rates of each operator. The legend shows the values of source rate.



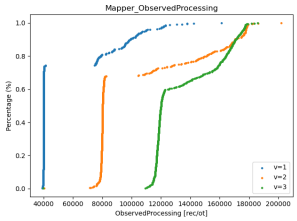
(a) Latency Sink observed processing rate



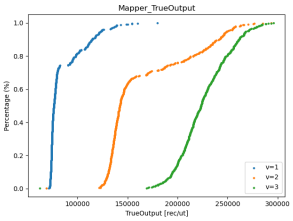
(b) Latency Sink true processing rate



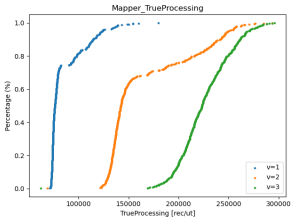
(c) Mapper observed output rate



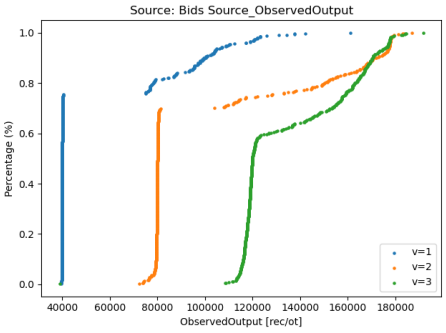
(d) Mapper observed processing rate



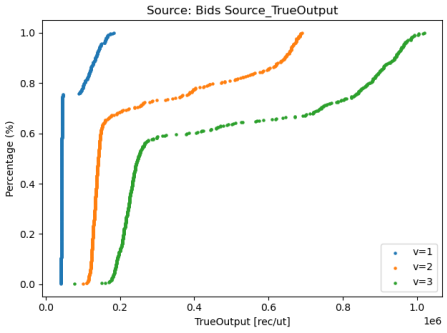
(e) Mapper true output rate



(f) Mapper true processing rate



(g) Bids Source observed output rate

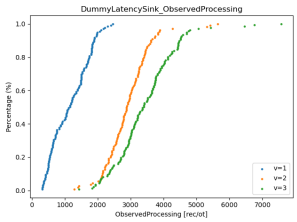


(h) Bids Source true output rate

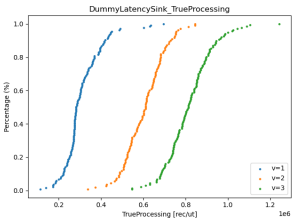
Figure 14: Query1's effect of increasing all operators' parallelism values simultaneously on processing and output rates. The legend shows the parallel instances of the operators.

Query5. Observing Figure 16, we observe that all operators significantly benefit from the increase of their parallel instances. In addition, the green line which corresponds to three parallel instances for each operator is not steep as the rest and yields result almost three times better than using a single instance.

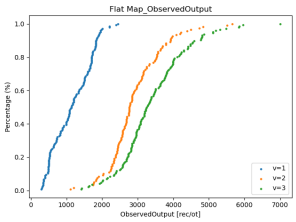
Query11. Looking at Figure 17, we observe that all operators significantly be-



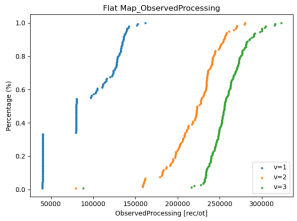
(a) Latency Sink observed processing rate



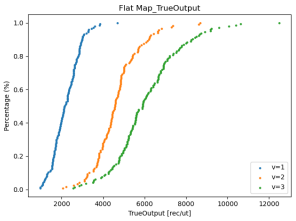
(b) Latency Sink true processing rate



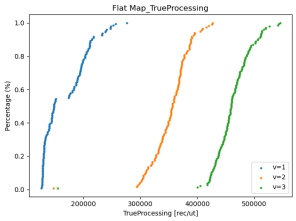
(c) Flat Map observed output rate



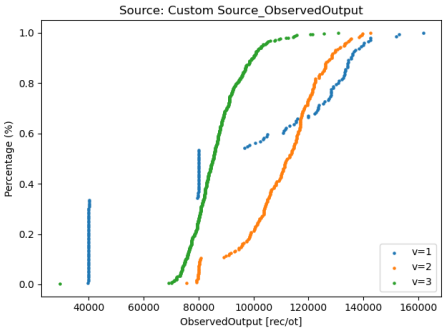
(d) Flat Map observed processing rate



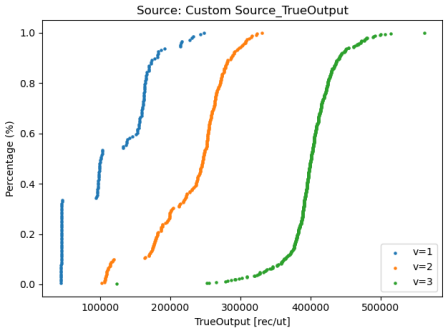
(e) Flat Map true output rate



(f) Flat Map true processing rate



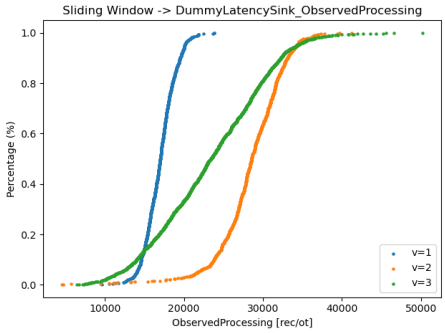
(g) Custom Source observed output rate



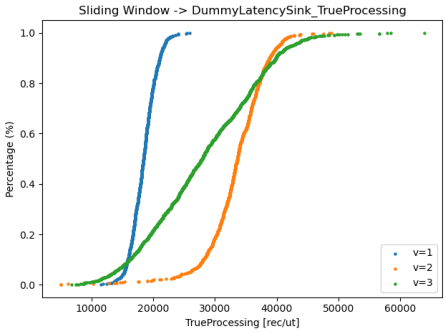
(h) Custom Source true output rate

Figure 15: Query2's effect of increasing all operators' parallelism values simultaneously on processing and output rates. The legend shows the parallel instances of the operators.

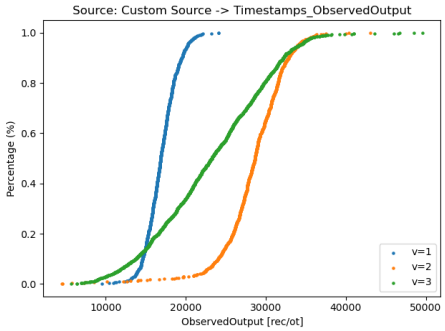
nefit by the increase of their parallel instances. In addition, the green line which corresponds to three parallel instances for each operator, is not steep as the rest and yields results almost four times better than using a single instance.



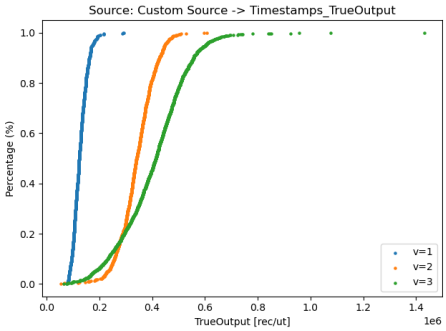
(a) Latency Sink observed processing rate



(b) Latency Sink true processing rate



(c) Custom Source observed output rate



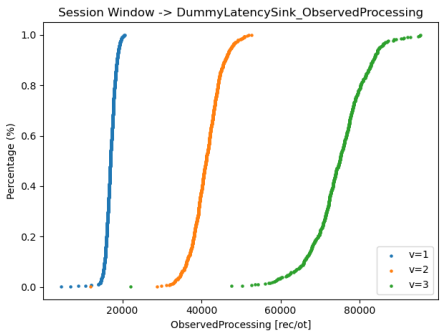
(d) Custom Source true output rate

Figure 16: Query5's effect of increasing all operators' parallelism values simultaneously on processing and output rates. The legend shows the parallel instances of the operators.

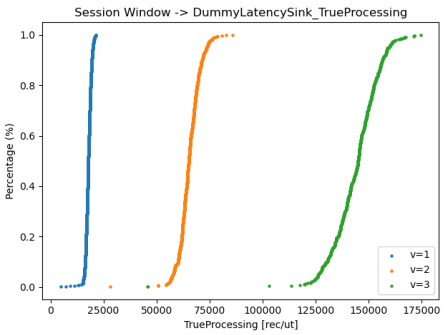
5.4 Summary

Based on our previous evaluation we concluded that following general statements:

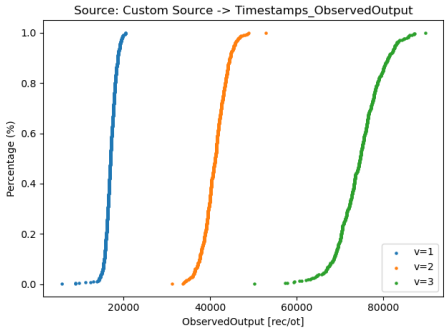
1. In most cases, the true processing (resp. output) rate differs with the observed ones. In addition, the former is greater by a big margin from the latter. However, there were few occasions that these rates were identical but we believe this was due to the fact that the experiments were performed in a single machine minimizing the negative effects of various external factors.
2. In a data pipeline the parallelism values of the operators that precede the current operator we analyze affect its performance. For example in Figure 7 *Source* affects the rest but *Flat Map* and *Dummy Latency Sink* cannot have an effect on *Source*.
3. If we increase all the operators' instances we almost certainly see a considerable performance improvement. However, we believe that the more we



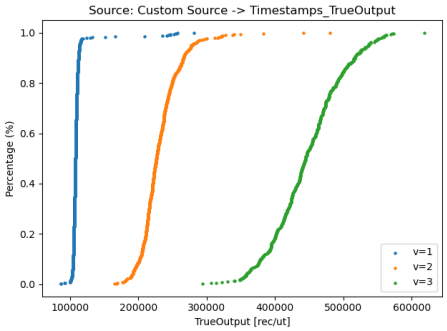
(a) Latency Sink observed processing rate



(b) Latency Sink true processing rate



(c) Custom Source observed output rate



(d) Custom Source true output rate

Figure 17: Query11's effect of increasing all operators' parallelism values simultaneously on processing and output rates. The legend shows the parallel instances of the operators.

increase the parallelism value of the operators that boost is becoming secondary and may not be worth pursuing.

4. Regarding the input rates, choosing the optimal ones comes down to identifying the workload of each operator as some operators benefit more than others. However, as we propose in §**??**, extending the range of input rates, in both direction, might reveal new motifs.

6 Conclusion

Coming with a concrete solution to benchmarking Streaming Systems is a quite complex task. Many factors come into play that cannot be controlled. The ideal benchmark suite must cover all systems characteristics uniformly and draw the corresponding results per case.

For this reason, in this paper, we presented a new way of benchmarking systems like these. It is heavily based on the notion of useful and observed time

which was discussed in §3.2. This way we evaluate each operator separately and filter out any noise generated by external factors. In addition, we observed a set of patterns, analysed them per case and then reported some general conclusions.

In particular, in this project we built an automation tool in order to generate the experimental data we needed to evaluate; we used two different ways of data generation. We, then, stored all results for analysis.

However, more time was needed to be dedicated to this project due to the multiplex nature of streaming systems. That is why we propose some suggestions in the following subsection.

6.1 Future Work

As future work, it would be useful to extend this benchmark workflow to include more streaming systems. Due to the fact that this system is compatible with Beam model, this should be a relatively easy task. On top of that, DS2 is already implemented on top of Timely DataFlow [22] and Apache Heron [23], so these two options are a good start.

In addition, Query8 needs to be further investigated, as it produces an error that renders it unable to be executed on Flink. The error message is related with the DS2 *MetricsManager* and the Tumbling Window of Latency Sink and can be seen below:

```
java.lang.NumberFormatException: For input string:
"TumblingEventTimeWindows(10000), ListStateDescriptor{serializer=
org.apache.flink.api.common.typeutils.base.ListSerializer@91811454},
EventTimeTrigger(), WindowedStream.apply(CoGroupedStreams.java:303))
 -> DummyLatencySink (1"
at java.lang.NumberFormatException.forInputString
(NumberFormatException.java:65)
at java.lang.Integer.parseInt(Integer.java:580)
at java.lang.Integer.parseInt(Integer.java:615)
at org.apache.flink.runtime.util.profiling.MetricsManager.
<init>(MetricsManager.java:68)
at org.apache.flink.runtime.taskmanager.RuntimeEnvironment.
<init>(RuntimeEnvironment.java:133)
at org.apache.flink.runtime.taskmanager.Task.run(Task.java:669)
at java.lang.Thread.run(Thread.java:748)
```

Another technical issue was produced by Query3 and Query3Stateful. After a few completed jobs, Flink stopped generating the output files forcing the automation tool to stop, since it was not detecting any changes in the output directory. On top of that, the automation tool was submitting the next query on Flink and eventually Flink was running out of available tasks slots.

On the other hand, it is crucial to explore further the performance of Flink by expanding the ranges of the input rates and the parallelism value of the operators on the Automation Tool. This can be easily done by tweaking the corresponding

global variables in the *settings.py* file. Due to limited time, we restricted the instances of each operator to a maximum of three and we did not experiment with high input rates or low. Furthermore, it is worth exploring the option of creating new ways of input combinations as we were limited to only two ways. For example, we were only increasing parallelism value by one at a time where more variety is needed to draw a more general conclusion; similarly with input rates. A wider input variety might unveil the operators that bottleneck the system. It would be also useful if the parallelism value of *Mapper* and *Latency Sink* was exposed as different variables in the NEXMark queries source file because this way we could not measure the impact of changing the parallelism the former to the latter.

Another possible future work might be adjusting the parameters of the execution of a streaming query at runtime. However, such work is limited to streaming systems that have a savepoint mechanism like Flink. Savepoint allows us to store the state of the system and resume its execution afterwards with different settings. Changing input arguments on runtime might reveal new patterns or behaviours in the processing and output rates of each operator.

Finally, in the future, it would be beneficial to deploy this setup to a distributed environment and perform the same experiments. Since the experimental evaluation was done based on a single machine many complexities of streaming systems were diminished. We acknowledge the fact that in our experiments delays of external factors, such as CPU utilization in multi-tenant architecture or general hardware failures, were not taken into consideration so we leave that to future work.

Acknowledgements. First and foremost, I would like to express my gratitude to my supervisor Matthew Forshaw who provided valuable guidance and technical advice throughout the progress of this project. Secondly, I would like to thank my family for believing in me and supported me in my educational career all these years.

References

[1] The Economist. *The world's most valuable resource is no longer oil, but data*. URL: `https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data` (Accessed 01/08/2020).

[2] Amir Gandomi and Murtaza Haider. "Beyond the hype: Big data concepts, methods, and analytics". In: *Elsevier* (2014). DOI: `http://dx.doi.org/10.1002/andp.19053221004`.

[3] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

[4] Fay Chang et al. "Bigtable: A Distributed Storage System for Structured Data". In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2006, pp. 205–218.

[5] Tyler Akidau, Slava Chernyak and Reuven Lax. *Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing*. O'Reilly Media, 2010. ISBN: 978-1491983874.

[6] Vasiliki Kalavri et al. "Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows". In: ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/system/files/osdi18-kalavri.pdf (Accessed 14/08/2020).

[7] Paris Carbone et al. *Apache Flink™: Stream and Batch Processing in a Single Engine*. Tech. rep. URL: http://asterios.katsifodimos.com/assets/publications/flink-deb.pdf (Accessed 14/08/2020).

[8] Pete Tucker et al. *NEXMark - A Benchmark for Queries over Data Streams DRAFT*. Tech. rep. OHSU. URL: http://datalab.cs.pdx.edu/niagara/pstream/nexmark.pdf (Accessed 19/05/2020).

[9] The Apache Software Foundation. *Beam Apache*. URL: https://beam.apache.org/ (Accessed 14/08/2020).

[10] Chunho Lee, Miodrag Potkonjak and William Mangione-Smith. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems." In: Dec. 1997, pp. 330–335.

[11] R. Lu et al. "Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks". In: *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. 2014, pp. 69–78.

[12] S. Chintapalli et al. "Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, pp. 1789–1792.

[13] Jay Kreps. "Kafka : a Distributed Messaging System for Log Processing". In: 2011.

[14] Josiah L. Carlson. *Redis in Action*. USA: Manning Publications Co., 2013. ISBN: 1617290858.

[15] J. Karimov et al. "Benchmarking Distributed Stream Data Processing Systems". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 1507–1518.

[16] E. Shahverdi, A. Awad and S. Sakr. "Big Stream Processing Systems: An Experimental Evaluation". In: *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. 2019, pp. 53–60.

[17] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux journal* 2014.239 (2014), p. 2.

[18] Konstantin Shvachko et al. "The Hadoop Distributed File System". In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. MSST '10. USA: IEEE Computer Society, 2010, pp. 1–10. ISBN: 9781424471522. DOI: 10.1109/MSST.2010.5496972. URL: https://doi.org/10.1109/MSST.2010.5496972.

[19] Etienne Chauchot. *Universal Metrics with Beam*. presentation. ApacheCon North America 2018. URL: https://www.slideshare.net/EtienneChauchot/universal-metrics-with-apache-beam (Accessed 12/08/2020).

[20] 2020. URL: https://app.diagrams.net/ (Accessed 16/08/2020).

[21] Canonical Ltd. *Ubuntu Image version 4.15.0-106-generic*. URL: https://packages.ubuntu.com/xenial/linux-image-4.15.0-106-generic (Accessed 14/08/2020).

[22] Martín Abadi and Michael Isard. "Timely Dataflow: A Model". In: *Formal Techniques for Distributed Objects, Components, and Systems*. Ed. by Susanne Graf and Mahesh Viswanathan. Cham: Springer International Publishing, 2015, pp. 131–145. ISBN: 978-3-319-19195-9.

[23] Sanjeev Kulkarni et al. "Twitter Heron: Stream Processing at Scale". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 239–250. ISBN: 9781450327589. DOI: 10.1145/2723372.2742788. URL: https://doi.org/10.1145/2723372.2742788.