Big Data to Enable Global Disruption of the Grapevine-powered Industries

D4.4 - Resource Optimization Methods and Algorithms

DELIVERABLE NUMBER	D4.4
DELIVERABLE TITLE	Resource Optimization Methods and Algorithms
RESPONSIBLE AUTHOR	Franco Maria Nardini (CNR)

Co-funded by the Horizon 2020
Framework Programme of the European Union

GRANT AGREEMENT N.	780751
PROJECT ACRONYM	BigDataGrapes
PROJECT FULL NAME	Big Data to Enable Global Disruption of the Grapevine-powered industries
STARTING DATE (DUR.)	01/01/2018 (36 months)
ENDING DATE	31/12/2020
PROJECT WEBSITE	http://www.bigdatagrapes.eu/
COORDINATOR	Nikos Manouselis
ADDRESS	110 Pentelis Str., Marousi, GR15126, Greece
REPLY TO	nikosm@agroknow.com
PHONE	+30 210 6897 905
EU PROJECT OFFICER	Ms. Annamária Nagy
WORKPACKAGE N. | TITLE	WP4 | Analytics & Processing Layer
WORKPACKAGE LEADER	Franco Maria Nardini (CNR)
DELIVERABLE N. | TITLE	D4.4 | Resource Optimization Methods and Algorithms
RESPONSIBLE AUTHOR	Franco Maria Nardini (CNR)
REPLY TO	francomaria.nardini@isti.cnr.it
DOCUMENT URL	http://www.bigdatagrapes.eu/
DATE OF DELIVERY (CONTRACTUAL)	30 September 2019 (M21)
DATE OF DELIVERY (SUBMITTED)	30 September 2019 (M21)
VERSION | STATUS	2.0 | Final
NATURE	DEM (Demonstrator)
DISSEMINATION LEVEL	PU (Public)
AUTHORS (PARTNER)	Vinicius Monteiro de Lira (CNR), Cristina Muntean (CNR), Franco Maria Nardini (CNR), Raffaele Perego (CNR), Nicola Tonellotto (CNR), Roberto Trani (CNR), Salvatore Trani (CNR)
CONTRIBUTORS	
REVIEWER	Nikola Tulechki (ONTOTEXT)

VERSION	MODIFICATION(S)	DATE	AUTHOR(S)
0.1	Initial contributions	20/07/2019	Matteo Catena (CNR) Nicola Tonellotto (CNR)
0.2	Additional contributions	27/08/2019	Nicola Tonellotto (CNR)
0.3	Additional contributions	02/09/2019	Franco Maria Nardini (CNR) Raffaele Perego (CNR)
0.4	Additional contributions	05/09/2019	Nicola Tonellotto (CNR)
0.5	Additional contributions	10/09/2019	Raffaele Perego (CNR) Nicola Tonellotto (CNR)
0.6	Internal review	12/09/2019	Nikola Rusinov (ONTOTEXT)
0.9	Pre-final version	13/09/2019	Nicola Tonellotto (CNR)
1.0	Final version	30/09/2019	Nicola Tonellotto (CNR)
1.1	Update	07/09/2020	Franco Maria Nardini (CNR) Roberto Trani (CNR)
1.9	Pre-final version	20/09/2020	Franco Maria Nardini (CNR) Roberto Trani (CNR)
1.95	Internal Review	25/09/2020	Nikola Rusinov (ONTOTEXT)
2.0	Final version	25/09/2020	Franco Maria Nardini (CNR) Roberto Trani (CNR)

PARTICIPANTS	CONTACT
Agroknow IKE (Agroknow, Greece)	Nikos Manouselis Email: nikosm@agroknow.com
Ontotext AD (ONTOTEXT, Bulgaria)	Todor Primov Email: todor.primov@ontotext.com
Consiglio Nazionale Delle Ricerche (CNR, Italy)	Raffaele Perego Email: raffaele.perego@isti.cnr.it
Katholieke Universiteit Leuven (KULeuven, Belgium)	Katrien Verbert Email: katrien.verbert@cs.kuleuven.be
Geocledian GmbH (GEOCLEDIAN Germany)	Stefan Scherer Email: stefan.scherer@geocledian.com
Institut National de la Recherché Agronomique (INRA, France)	Pascal Neveu Email: pascal.neveu@inra.fr
Agricultural University of Athens (AUA, Greece)	Katerina Biniari Email: kbiniari@aua.gr
Abaco SpA (ABACO, Italy)	Simone Parisi Email: s.parisi@abacogroup.eu
SYMBEEOSIS LONG LIVE LIFE S.A. (Symbeeosis, Greece)	Konstantinos Rodopoulos Email: rodopoulos-k@symbeeosis.com

ACRONYMS LIST

DVFS	Dynamic Voltage and Frequency Scaling
OLDI	OnLine Data-Intensive
PDRQ	Per-Datastore Replica Queue
PPTQ	Per-Processing Thread Queue
FL	Federated Learning
LSTM	Long Short-Term Memory Network

EXECUTIVE SUMMARY

This accompanying document for deliverable "D4.4 Resource Optimization Methods and Algorithms" describes new research tools to manage distributed big data platforms that will be used in the BigDataGrapes (BDG) platform to optimize computing resource management.

The BigDataGrapes platform aims at providing Predictive Data Analytics tools that go beyond the state-of-the-art for what regards their application to large and complex grapevine-related data assets. Such tools leverage machine learning techniques that largely benefit from the distributed execution paradigm that serves as the basis for addressing efficiently the analytics and scalability challenges of grapevines-powered industries (see Deliverable 4.1). However, distributed architectures can consume a large amount of electricity when operated at large scale. This is the case of OnLine Data Intensive (OLDI) systems, which perform significant computing over massive datasets per user request. They often require responsiveness in the sub-second time scale at high request rates.

In this document, we illustrate the main operational characteristics of the OLDI systems and we describe the implementation and usage of our OLDI Simulator, a java library to simulate OnLine Data-Intensive systems. This simulator can be used to study the performance of OLDI systems in terms of latency and energy consumption. Then, we show how to use our simulator by using an example, in which we 1) we describe a system to simulate, 2) we model the system, and 3) we evaluate and compare various solutions by simulation means. Finally, we discuss how our simulator can be used to compare energy-efficient scheduling algorithms in distributed big data platforms, using a WSE with real-world dataset as motivating example. The scientific results exploited by our OLDI simulator and the energy-efficient scheduling algorithms discussed in this document have been shared with the research community in a BigDataGrapes paper [cikm2018].

In the second revision of this deliverable (due M33), we present a novel technique that enables an efficient training of machine learning models in distributed environments. The proposed technique can be used within the BigDataGrapes platform to improve the usage of distributed computing resources. The BDG platform relies on a distributed environment composed of several co-operating machines to support efficient and effective predictive data analytics over the extremely large and complex grapevine-related data assets. Such Predictive Data Analytics tools benefit from state-of-the-art machine learning techniques to extract evidences from the data and learn models of the various phenomenon. Scalable and distributed machine learning algorithms are needed to efficiently extract knowledge from such extremely large datasets. Moreover, data of different vineyards could be located on different machines that could be far from each other, e.g., close to the location of the vineyards where the data are gathered. For this reason, it is crucial to distribute and move the computation closer to the data to avoid slow and costly data-exchange across the globe. To this regard, the leading approach to learn machine learning models in this scenario is known as Federated Learning (FL). The updated version of this deliverable presents the application of quantization techniques, i.e., a form of lossy compression, in Federated Learning to improve the efficiency in terms of data exchange between the machines involved in the computation. We focus on learning neural network models as they achieve state-of-the-art performance on several time-series tasks such as stock forecasting, natural language processing, and sequential image processing. Experiments on a public dataset show that the introduction of quantization techniques to the Federated Learning of neural network allows to reduce up to 19 × the volume of data exchanged between machines, with a minimal loss of performance of the final model. The scientific results as well as the algorithms discussed in this document have been shared with the research community in an article that is currently under review.

TABLE OF CONTENTS

1 INTRODUCTION

The BigDataGrapes platform aims at providing Predictive Data Analytics tools that go beyond the state-of-the-art for what regards their application to large and complex grapevine-related data assets. Such tools leverage machine learning techniques that largely benefit from the distributed execution paradigm that serves as the basis for addressing efficiently the analytics and scalability challenges of grapevines-powered industries (see Deliverable 4.1). However, distributed architectures can consume a large amount of electricity when operated at large scale. This is the case of OnLine Data Intensive (OLDI) systems, which perform significant computing over massive datasets per user request. They often require responsiveness in the sub-second time scale at high request rates. Some examples of OLDI systems are search engines, memory caching systems, online advertising, machine translation, online analytics tools, etc.

Due to their distributed nature, OLDI systems are typically composed by thousands of physical servers. This infrastructure is necessary to have small latencies and to guarantee that most users will receive results in sub-second times. At the same time, such many servers consume a significant amount of energy, hindering the profitability of the OLDI systems and raising environmental concerns. In fact, datacenters can consume tens of megawatts of electric power and the related expenditure can exceed the original investment cost for a datacenter. The BigDataGrapes platform has all the characteristics of an OLDI system: it is a distributed system, it manipulates large-sized data assets, and it must provide information (e.g., analysis, predictions, etc.) to the users with a reasonable latency.

Therefore, it is important to study the performance of the BigDataPlatform platform (and similar systems) before deploying it on a real hardware infrastructure. In particular, we would like to estimate its latency and energy consumption, to assess before the deployment if its requirements are met, both in terms of responsiveness and energy expenditure. For this reason, we developed OLDI Simulator, a Java library to simulates both the latencies and CPU energy consumption of OLDI systems. The OLDI Simulator's classes permit to model OLDI systems and to simulate their performance. The library can also be used to evaluate different solutions while developing an OLDI system.

In the second revision of this deliverable (due M33), we present a novel technique to improve the usage of the distributed computing resources when training the machine learning models used by the Predictive Data Analytics tools of the BigDataGrapes platform. The platform exploits a distributed architecture to execute the state-of-the-art machine learning techniques supporting such tools which extract evidences from the huge and complex grapevine-related data assets to learn models of the various phenomenon. For this reason, scalable and distributed machine learning algorithms must be employed to extract knowledge from such extremely large datasets in an efficient manner. Since data of different vineyards could be located on different machines that could be far from each other, e.g., close to the location of the vineyards where the data are gathered, the training of the models may involve the transfer of huge amount of data between the cloud machines, thus impacting the transfer costs to pay to the cloud provider. Therefore, it is crucial to distribute and move the computation closer to the data to avoid the slow data-exchange across the globe and the additional cloud costs due to the huge data-transfer. The leading approach to learn machine learning models in this scenario is known as Federated Learning (FL) and will be described in Section 5.1. In the updated version of this deliverable, we introduce the application of quantization techniques in Federated Learning to compress the data exchange between the machines involved in the training and reduce the most the volume of data-exchange. In particular, we focus on learning neural network models because they have been employed in other tasks of the BDG platform and, indeed, they achieve state-of-the-art performance on several time-series tasks such as stock forecasting, natural language processing, and sequential image processing. Experiments on a public dataset show that the introduction of quantization techniques to the Federated Learning of neural network allows to reduce up to 19 × the volume of data exchanged between machines, with a minimal loss of performance of the final model.

The rest of this document is structured as follows: in Section 2 we provide some background regarding OLDI systems and the challenges they incur into, in Section 3 we describe the OLDI Simulator library and how to use it with an example. Section 4 describes how our simulator can be used to compare energy-efficient scheduling algorithms in distributed big data platforms, using a WSE with real-world dataset as motivating example. In Sections 5.1 and 5.2 we describe the Federated Learning algorithm and our proposal concerning the introduction of compression techniques for reducing the amount of data-exchange. Then, in Section 5.3, we empirically assess the proposed techniques on a public dataset. Finally, Section 6 concludes the document.

2 BACKGROUND

In this section we will describe what is an *OnLine Data Intensive (OLDI) system* by illustrating its architecture and its fundamental components. We will then discuss about the energy consumption of such systems, which poses economic and environmental challenges for their adoptability.

2.1 OLDI SYSTEMS ARCHITECTURE

OLDI systems perform significant computing over massive datasets per user request. They often require responsiveness in the sub-second time scale at high request rates. Some examples of OLDI systems are search engines, memory caching systems, online advertising, machine translation, etc.

OLDI systems typically work as follows. Given a dataset, a *datastore* component is built. In search engines, for instance, the datastore is represented by the inverted index[1]. OLDI systems must manage massive datastores and process billions of *requests* per day with low latencies. Hence, OLDI systems often partition the datastore into smaller *shards* to process user requests. In fact, request processing times often depend on the datastore size. Since shards are smaller than the original datastore, this results in reduced processing times. After partitioning, shards are assigned to different *shard servers*. A shard server is a physical server composed by several *multi-core processors/CPUs* with a shared memory which holds the shard. A *request processing thread* is executed on top of each CPU core of the shard server. When a request is sent to an OLDI system, it is first received by a *request broker* which dispatches the request to every shard server. Each server computes the request results on its shard independently from the others. These partial results are sent back to the broker, which aggregates them. The aggregated results are the same that would be provided by a single datastore; since the computation is now distributed across several servers, request processing times are reduced. The request broker collects and aggregates the partial results from the shards, and the final results are sent to the issuing user. The set of shard servers holding all the shards is a *datastore replica*.

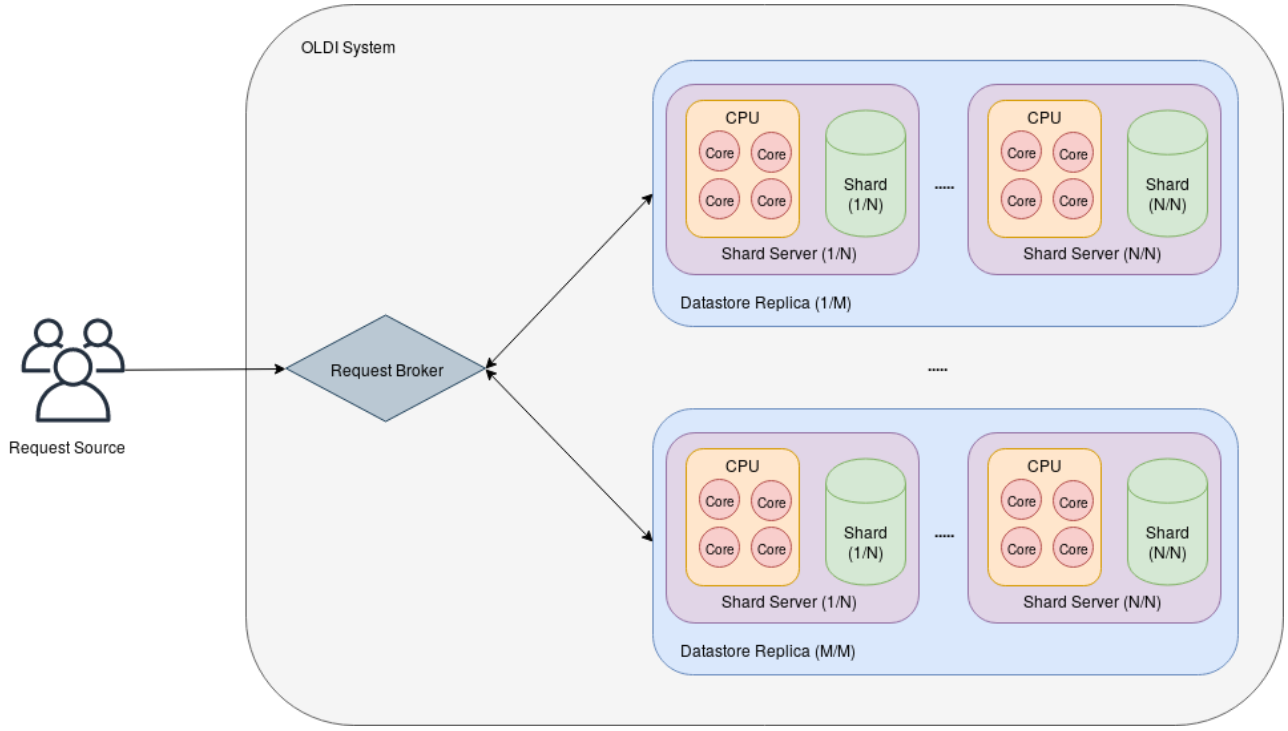

Figure 1 OLDI System with partitioned and replicated datastore.

1 https://en.wikipedia.org/wiki/Inverted_index

Figure 1 depicts an OLDI system where the datastore has been partitioned into *N* shards and it has been replicated *M* times. Therefore, such OLDI system has *M* datastore replicas, each composed by *N* shard servers, totaling *MxN* shard servers. When the request source (e.g., the system's users) send a request to the system, the request is processed as follows. Firstly, it is received by the request broker which dispatches it to a datastore replica according to some policy. For instance, the broker may decide to dispatch an incoming request to least loaded datastore replica (i.e., the one which is processing the least number of requests). Once dispatched to a datastore replica, the request must be processed – in a distributed fashion – by all its shard servers, since all shards are required to produce the request's results. At each shard server, the request is processed by a request processing thread (not shown in Figure 1), independently from the other servers. When all the shard servers have completed the request processing, the partial results are collected and merged together. The results are then sent back to the request broker, which can show the results to the request source.

A request processing thread runs on a CPU core. Since shard servers' CPUs have multiple cores, a shard server can process several requests in parallel. Nevertheless, when workloads are intense all the shard servers within an OLDI system may be busy processing requests. Therefore, new requests may be enqueued by the OLDI system in a *queue* (not shown in Figure 1). The first request in the queue is processed as soon as some computing resources becomes available. Where to place the request queue is an architectural choice. A popular option is the *Per-Datastore-Replica-Queue (PDRQ)*, where each datastore replica has its own request queue. Upon request arrival, the request broker push the request in the queue of the least loaded replica, which start processing it as soon as its shard servers are available. Another popular choice is the *Per-Processing-Thread-Queue (PPTQ)*, where each CPU core of each shard server has its own request queue. In this case, a request is processed as soon as the corresponding CPU core is free. Both architectural choices have their advantages. For instance, PDRQ typically leads to smaller latencies. However, PPTQ permits a fine-grained control of the CPU power management mechanisms.

2.2 OLDI SYSTEMS ENERGY CONSUMPTION

OLDI systems are typically composed by thousands of physical servers, hosted in large *datacentres*. This infrastructure is necessary to have small latencies and to guarantee that most users will receive results in sub-second times. At the same time, such many servers consume a significant amount of energy, hindering the profitability of the OLDI systems and raising environmental concerns. In fact, datacentres can consume tens of megawatts of electric power and the related expenditure can exceed the original investment cost for a datacentre.

In the past, a large part of a datacentre energy consumption was accounted to inefficiencies in its cooling and power supply systems. However, modern datacentres have largely reduced the energy wastage of those infrastructures, leaving little room for further improvement. On the contrary, opportunities exist to reduce the energy consumption of the servers hosted in a datacentre. In particular, the CPUs dominate the energy consumption of physical servers.

Modern CPUs usually expose two energy saving mechanism, namely *C-states* and *P-states*. C-states represent CPU cores idle states and they are typically managed by the operating system. C0 is the operative state in which a CPU core can perform computing tasks. When idle periods occur, i.e., when there are no computing tasks to perform, the core can enter one of the other deeper C-states and become inoperative. However, OLDI systems usually process a large and continuous stream of requests. As a result, their servers rarely inactive and experience particularly short idle times. Consequently, there are little opportunities to exploit deep C-states.

When a CPU core is in the active C0 state, it can operate at different frequencies (e.g., 800 MHz, 1.6 GHz, 2.1 GHz, ...). This is possible thanks to the *Dynamic Frequency and Voltage Scaling (DVFS)* technology which permits to adjust the frequency and voltage of a core to vary its performance and power consumption. In fact, higher core frequencies mean faster computations but higher power consumption. Vice versa, lower frequencies lead to slower computations and reduced power consumption. The various configurations of voltage and frequency available to the CPU cores are mapped to different P-states.

3 THE OLDI SIMULATOR LIBRARY

As illustrated in Section 2, OLDI systems require huge monetary investments to both be deployed and operated. In fact, OLDI systems are usually composed by a large number of physical servers which must be hosted in a datacentre, whose construction cost is not negligible. Additionally, OLDI systems require a huge amount of electricity to operate, resulting in large energy expenditure and negative environmental impact.

Therefore, it is customary to analyse the performance of OLDI systems during their development and before deployment, to be sure they both meet the service level objective for which they are being designed (e.g., a tail latency below 500 milliseconds) and respect energy consumption constraint. To conduct performance analysis, it is possible to use simulations.

For this reason, we developed **OLDI Simulator**, a Java library to simulate both the latencies and CPU energy consumption of OLDI systems. The OLDI Simulator's classes permit to model OLDI systems like those described in Section 2, and to simulate their performance. The library can be used to evaluate different solutions while developing an OLDI system.

In the following, we will describe the library and its fundamental components, we will illustrate how to install it, and we will provide an example of its usage. OLDI Simulator has been successfully used in [cikm2018]. Its source code can be found at https://github.com/BigDataGrapes-EU/deliverable-D4.4.

3.1 LIBRARY DESCRIPTION

The OLDI Simulator library models several of the concepts introduced in Section 2. For instance, requests are represented by the it.cnr.isti.hpclab.request.Request class, whose instances can be generated by a it.cnr.isti.hpclab.request.RequestSource. A it.cnr.isti.hpclab.engine.RequestBroker instance can be used to simulate the dispatching of requests towards various datastore replicas, which are represented by the it.cnr.isti.hpclab.engine.DatastoreReplica class. This refers to multiple it.cnr.isti.hpclab.engine.ShardSever instances, which represent physical servers. Finally, shard servers hold a shard, which is represented as a it.cnr.isti.hpclab.engine.Shard instance, and run several request processing threads, which are represented by it.cnr.isti.hpclab.engine.RequestProcessingThread instances. The library currently implements the aforementioned classes for both the PDRQ and PPTQ architectures. Using these classes, the library permits to model and simulate an OLDI system, for example to estimate its latency.

The library permits to reliably estimate also the CPU energy consumption of OLDI systems. To this end, the library provides the it.cnr.isti.hpclab.cpu.CPU class to represent any CPU. A CPU has one or more cores, which are represented by the it.cnr.isti.hpclab.cpu.Core class. Finally, CPU instances are obtainable via the it.cnr.isti.hpclab.cpu.CPUBuilder. Currently, this library implements the aforementioned classes for the Intel i7-4770K[2] product, and it models the energy consumption of this CPU when running a search application (see Sec. 3.1.1 for further details). DVFS mechanisms are simulated within the Core class, which exposes the setFrequency() and getFrequency() methods to vary the core operating frequency and to query it.

To simulate an OLDI system, instantiate and configure the it.cnr.isti.hpclab.simulator.OLDISimulator class.

3.1.1 How to model the system latency

Among other factors, a system latency is influenced by the arrival rate of its requests and by their service times (i.e., the time needed by a processing thread to complete a request). In turn, service times are influenced by the CPUs operating frequencies. In fact, small frequencies lead to small energy consumptions but long

2 https://ark.intel.com/content/www/us/en/ark/products/75123/intel-core-i7-4770k-processor-8m-cache-up-to-3-90-ghz.html

processing times, while large frequencies have large energy consumption and short processing times. Service times depend also on which particular shard a request is processed. For instance, large shards may lead to long processing times while small shards to short ones. Therefore, to carefully simulate an OLDI system, one must know 1) how frequently requests arrive to the system, and 2) how long are the service times on a <shard x frequency> basis. Both information can be derived by monitoring an already running system.

For instance, in [cikm2018] we simulated a Web search engine. Query arrival rates were derived by analysing the MSN2006 query log[3], which reports the arrival time of its queries. Query service times, instead, were collected from real-world experiments carried out using the Terrier IR platform[4] on a dedicated server equipped with 32 GB RAM and an Intel i7-4770K processor. We used the ClueWeb09[5] (cat. A) collection, by indexing its Web pages and by partitioning the resulting inverted index into 5 shards. To derive the service times of the search engine, we randomly sampled 10,000 unique queries from the second day of the MSN2006 log. Then, we processed these queries using Terrier. Every query was processed on each of the five index shards and at each of the 15 core frequencies available on the i7-4770K. We recorded the query processing times to plug them into the simulation.

The information derived by this kind of experiment must then be encoded in Java to extend the it.cnr.isti.hpclab.request.RequestSource and it.cnr.isti.hpclab.engine.Shard. The RequestSource must be extended to implement the generate(Simulator simulator, Agent to) and isDone() methods. The first method generates instances of it.cnr.isti.hpclab.request.Request, which are forwarded to the Agent to (typically a it.cnr.isti.hpclab.engine.RequestBroker) and scheduled by simulator to be simulated. The latter method tells when the RequestSource must stop to generate requests.

In the Shard class, instead, we must implement the getServiceTime(long rid, int frequency) method. As the name suggests, this method tells us how long is the service time of a request (whose identifier is rid) when it is processed at frequency frequency kHz on that particular Shard instance.

3.1.2 How to model a CPU power consumption

A CPU consumes an amount of electric power that depends on the kind of tasks it is performing and on its configuration, i.e., the number of its cores which are actively performing some task and the frequencies at which they operate. Therefore, to accurately estimate the energy consumption of an OLDI system, one must first know which kind of CPUs it is going to adopt. Then, the CPU product must be carefully modelled by following its hardware specification and by studying its power consumption while performing the OLDI task (e.g., Web search, machine translation, etc.). Finally, one must extend the it.cnr.isti.hpclab.cpu.CPU, it.cnr.isti.hpclab.cpu.Core, and it.cnr.isti.hpclab.cpu.CPUBuilder to reflect these aspects.

For instance, in [cikm2018] we simulated a Web search engine. In our study, we assumed that all the search engine's CPUs were Intel i7-4770K. The i7-4770K has 4 physical cores which expose 15 operational frequencies ranging from 800 MHz to 3.5 GHz. Therefore, we represent the i7-4770K with the it.cnr.isti.hpclab.cpu.impl. Intel_i7_4770K class which extends it.cnr.isti.hpclab.cpu.CPU. The Intel_i7_4770K class refers to four it.cnr.isti.hpclab.cpu.impl. Intel_i7_4770K_Core(s), and it returns an array containing the 15 aforementioned frequencies when its getFrequencies() method is invoked.

Since the i7-4770K has 4 cores and 15 frequencies, it also has 3,875 possible configurations in terms of active cores and core frequencies. Therefore, we must map each of these configurations to their power consumption

3 http://www.sobigdata.eu/content/query-log-msn-rfp-2006
4 http://terrier.org/
5 http://www.lemurproject.org/clueweb09/

to accurately estimate the energy consumption of this CPU. As mentioned before, power consumptions must be measured while the CPU is busy performing the specific OLDI task we want to simulate.

To simulate the energy consumption of a Web search engine, in [cikm2018] we indexed the ClueWeb09 (cat. B) Web corpus using the Terrier IR platform. The resulting inverted index was kept in main memory by a dedicated server equipped with 32 GB RAM and the i7-4770K processor. This setting was then used to measure power consumptions as follows. For each possible CPU configuration, we launched a number of instances of the search platform equal to the number of active cores in the configuration. Each search instance was pinned to one of the cores, which operates at the frequency indicated by the CPU configuration. The instances continuously matched queries from the first day of the MSN2006 log against the aforementioned inverted index, to retrieve the top 1,000 documents. We used Mammut[6] to measure the energy being consumed by the CPU to derive its power consumption. These experiments allowed us to map each CPU configuration to its power consumption and to use this information for simulations. For instance, our simulated i7-4770K CPU consumes 0.8 Watts when idle, and up to 34.2 Watts when all its cores are busy processing queries at 3.5 Ghz.

The information derived by this kind of experiment must then be encoded in Java when extending the it.cnr.isti.hpclab.cpu.CPU class to represent a particular CPU product. In particular, the update(long timeInMicros) method must be implemented to invoke the updateEnergyConsumption(double power, long statusChangeTimeInMicros, double seconds) of it.cnr.isti.hpclab.simulator.OLDISimulator. During the simulation, this method is used to update the estimate of the amount of energy consumed by the simulated system. The method models the fact that, starting from statusChangeTimeInMicros (expressed in microseconds), for second seconds, the system was consuming power Watts. The value of the power parameter is decided according to the power consumption of the CPU product that we have previously modelled.

3.2 INSTALLATION AND DEPENDENCIES

OLDI Simulator requires Java SDK[7] >= 1.7, Maven[8], and Git[9] to be installed. To install OLDI Simulator, please open a terminal and type the following commands:

```
> git clone https://github.com/BigDataGrapes-EU/deliverable-D4.4.git
> cd deliverable-D4.4
> git submodule init
> git submodule update --remote
> mvn install package
```

In order to work, OLDI Simulator requires the following dependencies:

- jades[10]
- commons-math3[11]
- fastutil[12]
- guava[13]
- argparse4j[14]

The dependencies will be automatically managed by Maven.

6 http://danieledesensi.github.io/mammut/
7 https://www.oracle.com/technetwork/java/javase/downloads/index.html
8 https://maven.apache.org/
9 https://git-scm.com/
10 https://github.com/catenamatteo/jades
11 https://commons.apache.org/proper/commons-math/
12 http://fastutil.di.unimi.it/
13 https://github.com/google/guava
14 https://argparse4j.github.io/

3.3 LIBRARY USAGE

In this section, we will describe how to use the OLDI Simulator library by providing an example of how to model and simulate an OLDI system.
Imagine we want to analyse the performance of a search system whose datastore is composed by two shards. In this system, request service times have an exponential distribution; the mean service time is μ_A on shard A and μ_B on shard B. The service times on shard B are slightly larger than on shard A, i.e., $\mu_A \geq \mu_B$. We also know that the request arrivals occur at rate λ according to a Poisson process. Finally, we assume that all the shard servers are equipped with an Intel i7-4770K.
In the following, we will show how to model the described setting by using the OLDI Simulator library, and how to simulate such OLDI system to answer questions like a) how many datastore replicas are needed to keep the 95th percentile latency below τ milliseconds? b) How much energy is consumed if the CPUs operate at f kHz? and so on.

3.3.1 How to model the request arrivals

To model the request arrivals, we need to extend the it.cnr.isti.hpclab.request.RequestSource class. In particular, we need to implement the generate(Simulator simulator, Agent to) and isDone() methods. The first method generates instances of it.cnr.isti.hpclab.request.Request, which are forwarded to the Agent to (typically a it.cnr.isti.hpclab.engine.RequestBroker) and scheduled by simulator for simulation. The second method tells when the RequestSource must stop to generate requests. In the following, we will call the new class RequestSourceImpl.

In our example, we want the arrivals occur at rate λ according to a Poisson process. Poisson distributions are implemented by the org.apache.commons.math3.distribution.PoissonDistribution class, which we instantiate in the RequestSourceImpl constructor. We also set a fixed seed for poissonDistribution, such that different simulation runs will generate the same request arrivals.

```
public RequestSourceImpl(int x, double lambda)
{
        long currentTime = 0;
        long end = TimeUnit.HOURS.toMicros(x);
        PoissonDistribution poissonDistribution = new PoissonDistribution(lambda);
        poissonDistribution.reseedRandomGenerator(42);
        generatedRequests = new LinkedList<>(); //Queue<Request>
        int uid = 0;
        while (currentTime <= end) {
                int numOfArrivals = poissonDistribution.sample();
                long intervalBetweenArrivals = TimeUnit.SECONDS.toMicros(1) / numOfArrivals;
                for (int i = 0; i < numOfArrivals; i++) {
                        currentTime += intervalBetweenArrivals;
                        if (currentTime <= end) {
                                uid += 1;
                                Request request =
                                        new Request(new Time(currentTime, TimeUnit.MICROSECONDS),
                                                        uid,
                                                        uid);
                                generatedRequests.add(request);
                        }
                }
        }
}
```

In our example, we want to simulate the system for *x* hours. To do so, we declare two long variables: end and currentTime. end indicates the time at which the class should stop to generate new requests. Instead, currentTime is used to keep track of the last generated arrival time. At each iteration of the while cycle, we determine the number of request arrivals by invoking the sample() method of poissonDistribution. The interval between arrivals (in seconds) is the reciprocal of the number of arrivals. For each arrival, we then increment currentTime by the interval between arrivals, and we generate a new Request. Requests are enqueued into the generatedRequests queue. We keep to generate requests until currentTime <= end. (Note: uid is a variable which is used to assign a unique identifier to each generated request).

During the simulation, requests will be fetched from the generatedRequests queue. Therefore, RequestSourceImpl will stop when the queue is empty. Consequently, we can implement isDone() as follows:

```
public boolean isDone()
{
        return generatedRequests.isEmpty();
}
```

We now need to implement the generate() method. The method just needs to fetch the first available request from the generatedRequests queue. The request is then wrapped into a RequestArrival (a jades' class which represents the arrival of a request to the simulated system). Finally, the request arrival must be scheduled on the simulator (an instance of jades' Simulator) which will advance the simulation.

```
public void generate(Simulator simulator, Agent to)
{
        Request request = nextRequest();
        if (request != null) {
                RequestArrival ra = new RequestArrival(request.getArrivalTime(), this, to, request);
                simulator.schedule(ra);
        }
}
```

nextRequest() is an abstract method inherited from jades' RequestSource. In our example, it is implemented to fetch requests from the queue as:

```
public Request nextRequest()
{
        if (isDone())
                return null;
        else return
                generatedRequest.poll();
}
```

3.3.2 How to model the service times

To model the request service times, we need to extend the it.cnr.isti.hpclab.engine.Shard class. In particular, we must implement the getServiceTime(long rid, int frequency) method. As the name suggests, this method tells us how long is the service time of a request (whose identifier is rid) when it is processed at frequency frequency kHz on that particular Shard instance. In the following, we will call the new class ShardImpl.

In our example, request service times have an exponential distribution and the mean service time is μ. Exponential distributions are implemented by org.apache.commons.math3.distribution.ExponentialDistribution, which we can instantiate in the ShardImpl

constructor. We also assume that the mean service time is equal to μ milliseconds when the CPU cores operate at maximum frequency, and that the service times are directly proportional to the frequency. Therefore, in the constructor, we initialize a maxFreq attribute to store the maximum available CPU frequency. This information will be used in getServiceTime() to scale the service time of a request according to the actual core frequency. Such information can be obtained by passing a CPUBuilder instance to the ShardImpl constructor, as follows

```
public ShardImpl(CPUBuilder cpuModel, int mu)
{
        maxFreq = Integer.MIN_VALUE;
        for (int freq : cpuModel.getFrequencies()) maxFreq = Math.max(maxFreq, freq);
        expDistribution = new ExponentialDistribution(mu);
}
```

We can now implement getServiceTime(long rid, int frequency). In this method, we use expDistribution to sample a service time from the exponential distribution. Such service time is then scaled according to frequency and returned as a Time object (a jades' class used to represent time quantities).

```
public Time getServiceTime(long rid, int frequency)
{
        expDistribution.reseedRandomGenerator(rid + Short.MAX_VALUE);
        double minServiceTime = expDistribution.sample();
        long serviceTime = (long) ((maxFreq * minServiceTime) / ((double)frequency));
        return new Time(serviceTime, TimeUnit.MILLISECONDS);
}
```

(Note: expDistribution is reseeded with rid at every invocation of getServiceTime(long rid, int frequency), such that the same service time is returned for the same rid).

3.3.3 How to simulate the modelled system

Once we have implemented RequestSourceImpl and ShardImpl, we can finally program and run our simulation. To do so, we have to write a Simulation class with a main method to invoke in order to start the simulation.

```
 public static void main(String args[]) throws IOException
{
        CPUBuilder cpuBuilder = new Intel_i7_4770K_Builder();

        int x = 2;
        int lambda = 40;
        RequestSource source = new RequestSourceImpl(x, lambda);

        int muA = 80; int muB = 90;
        Shard shardA = new ShardImpl(cpuBuilder, muA);
        Shard shardB = new ShardImpl(cpuBuilder, muB);

        OLDISimulator simulator = new OLDISimulator("output.gz", x);
        int r = 2;
        RequestBroker broker = new it.cnr.isti.hpclab.engine.pdrq.RequestBroker(
                simulator, cpuBuilder, r, shardA, shardB);

        int f = 3500000;
        simulator.setRequestSource(source);
```

```
        simulator.setBroker(broker);
        simulator.setFrequency(f);

        source.generate(simulator, broker);

        System.out.println("Simulating...");
        simulator.doAllEvents();
        System.out.println("...done!");
}
```

In our example, we assume that all the shard servers are equipped with an Intel i7-4770K CPU. Therefore, in the main() we instantiate a CPUBuilder for that product. We then instantiate a RequestSourceImpl with parameters x (simulation duration, in hours) and lambda (mean arrival rate). We also instantiate two ShardImpl(s) to represent shard A and shard B, and we pass to their constructors the cpuBuilder and the mean service time of the shard. After that, we create an instance of OLDISimulator, which is the class which actually performs the desired simulation. Its constructor takes, as parameters, the path of the file where to write the results of the simulation (gzipped), and the simulation duration (in hours). The last object we need to create is the RequestBroker. In this example we will assume that the system adopts a PDRQ architecture, so we choose the request broker from the it.cnr.isti.hpclab.engine.pdrq package. To be instantiated, the broker needs the references to the simulator, to the cpuBuilder, and to all the shards. Also, the broker needs to know how many datastore replicas to instantiate (parameter r). Finally, we tell to the simulator which are its request source, request broker, and at which frequency f its CPUs must operate, using its setter methods. To conclude, we must instruct the source to start to generate the requests, and the simulator to process all the simulation events.

3.3.4 How to analyse the output

Once the simulation is completed, an output file (gzipped) will be generated. The file has two kinds of entries:

- *broker*, and
- *energy*.

For each simulated request, we will have a broker line in the output, like:
[broker] 4 14.000
This tell us that the simulated system has received a request at second 4 (2nd field) and that its *completion time* was 14 milliseconds (3rd field). Also, for each (simulated) second we will have an energy line in the output, like:
[energy] 86395 47.728
This tell us that the simulated system has consumed 47.728 Joules at second 86395 of the simulation.

The output file can be munged to a) separate the energy information from the rest of the file, and b) to get the 95th-tile latency. To get the energy data, it is sufficient to open a terminal and to type:
> zcat output.gz | grep energy | cut -f3 -d' ' > output.energy
This will generate a 'output.energy' file containing only the data regarding the simulated energy consumption. To get the 95th-tile latency (one entry per second, 30-seconds moving 95th-tile latency in milliseconds), we use the mungetime-gzip.py script, which can be found in the OLDI Simulator's repository. To get the latency data, open a terminal and type:
> python3 mungetime-gzip.py output.gz > output.95th-tile
This will generate a 'output.95th-tile' file containing the tail latencies of the simulated system. Finally, the generated files can be plotted using the plot_times.py and plot_energy.py scripts included in the library repository. The script plots the 30-seconds moving 95th-%tile latency (resp. The energy consumption) of the system, both per second and with an order 3 Savitzky-Golay smoothig over a ten minutes time window,

Regarding our example, let's assume that λ=40 requests/second, μ_A=80 milliseconds/request and μ_B=90 milliseconds/request, τ=400 milliseconds. We simulate four system configurations:

- PDRQ-18S (small), with 2 datastore replicas and CPUs@1.8GHz,
- PDRQ-18L (large), with 4 datastore replicas and CPUs@1.8GHz,
- PDRQ-35S (small), with 2 datastore replicas and CPUs@3.5GHz,
- PDRQ-35L (large), with 4 datastore replicas and CPUs@3.5GHz.

Once the respective simulations have been completed, we plot their latencies using the aforementioned script. As we can see in Figure 2, the two configurations which operate their CPUs at 1.8GHz cannot meet the desired latency of 400 milliseconds. On the other hand, the desired latency can be obtained by operating the CPUs at 3.5GHz, indifferently by the number of datastore replicas. The number of datastore replicas, however, has an impact on the energy consumption of the system. Indeed, in Figure 3, we can see how PDRQ-35L consumes more energy than PDRQ-35S while exhibiting the same latency.

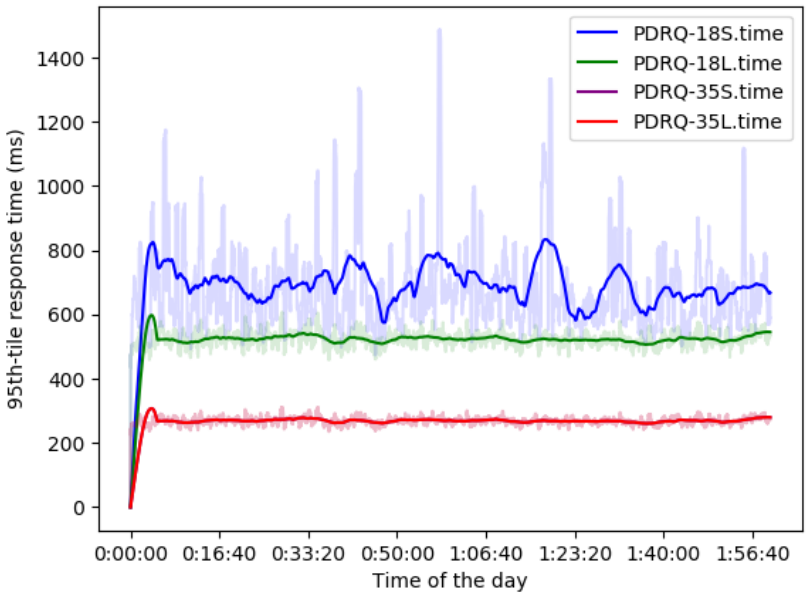

Figure 2 Tail response time (in milliseconds).

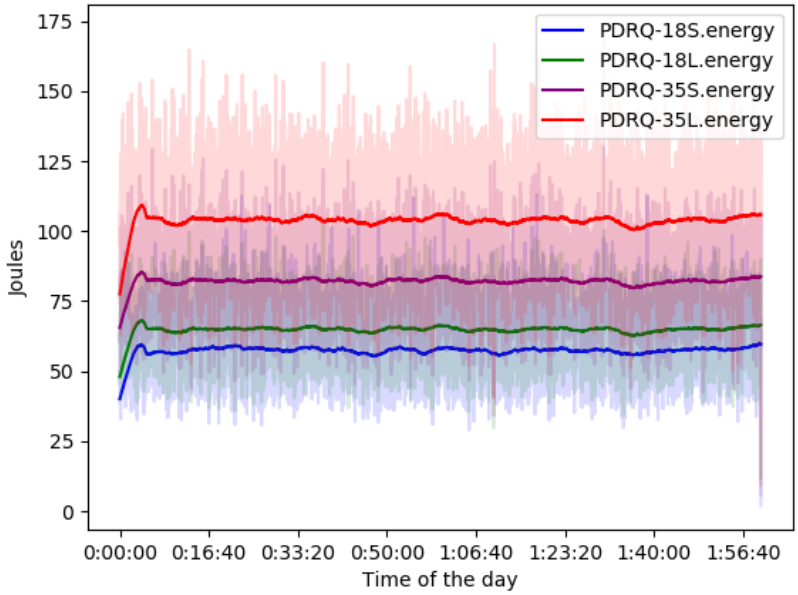

Figure 3 Energy consumption (in Joules).

4 EFFICIENT ENERGY MANAGEMENT IN DISTRIBUTED WEB SEARCH

The distributed architecture adopted by OLDI systems ensures that most user request will quickly be processed, in sub-second times (e.g., 500 ms) [arapakis2014]. However, such many servers consume a significant amount of energy, mostly accountable to the power consumption of their CPUs. The resulting electricity expenditure can hinder the profitability of OLDI systems, as they can consume tens of megawatts of electric power. Therefore, energy efficiency is an important aspect for the economic successfulness of OLDI systems [barroso2013].

OLDI systems can trade-off performance (i.e., longer response times) for lower energy consumptions when this does not affect the user experience. Such trade-offs are feasible by varying the frequency and voltage of CPU cores in WSEs' servers via Dynamic Frequency and Voltage Scaling (DVFS) technologies. Thanks to DVFS, OLDI systems can save energy by processing data no faster than necessary. State-of-the-art implementations of this principle are PEGASUS and PESOS.

PEGASUS (Power and Energy Gains Automatically Saved from Underutilized Systems) [pegasus2014] is a technique that aims at improving the energy efficiency of large-scale systems such as WSEs. Experimentally shown, PEGASUS reduces by up to 20% the power consumption of a Google Search production cluster, while keeping its latencies within an acceptable service level objective (SLO). Differently, the PESOS (Predictive Energy Saving Online Scheduling) algorithm [pesos2017] is designed to reduce the CPU energy consumption of single servers. Experimentally evaluated, PESOS can reduce the CPU energy consumption of a query processing server by almost 50%, while response times are kept below a desired time threshold.

While PESOS results are significant, their validity is limited, as PESOS has only been tested on a single server configuration. Instead, the de facto standard for WSEs is to rely on a distributed architecture, as PEGASUS correctly assumes. In this Section we fill the gap between PESOS and PEGASUS by evaluating the performance of PESOS on a distributed WSE, and comparing PESOS and PEGASUS in terms of energy consumption and their success in meeting response time requirements. To this end, we use our OLDI simulator to simulate the behaviour of PESOS when deployed on thousands of servers.

To evaluate the performance of PESOS and PEGASUS at a realistic scale, we simulate a distributed Web search engine. We focus on a distributed web search engine since both the data to be processed (e.g., textual documents) and the user processing requests (textual queries) are available in the form of standard scientific datasets for research. Their volume (millions of documents and hundreds of thousands of queries), variety & variability (different query and document characteristics) and velocity (very small deadlines for query processing) are greatly relevant for the BigDataGrapes project, and they are a good representative of different workloads and operations on a distributed OLDI system. Our distributed architecture includes 1,000 servers processing both shared and replicated data. On each server, the data fits into the main memory. We simulate a real-time flow of query processing requests using 500,000 queries distributed along 24 hours, and every request is assigned a deadline of 500 milliseconds.

In doing so, we want to investigate the following research questions (RQs):

(1) Does PESOS help to reduce the CPU energy consumption of a distributed WSE?

(2) Does PESOS provide acceptable latencies in a distributed WSE?

(3) How does PESOS compare to PEGASUS in term of both latencies and energy consumption?

To answer RQ1 and RQ3, we simulate the energy consumption (measured in megawatt hours, MWh) of a WSE's CPUs, while we simulate their tail latencies (computed as the 95th percentile of response times distribution) to answer RQ2 and RQ3. To better understand the benefits of PEGASUS and PESOS for WSEs, we will also compare their performance with a WSEs which always operates its CPUs' cores at maximum frequency for the sake of low latencies. We refer to this configuration as PERF.

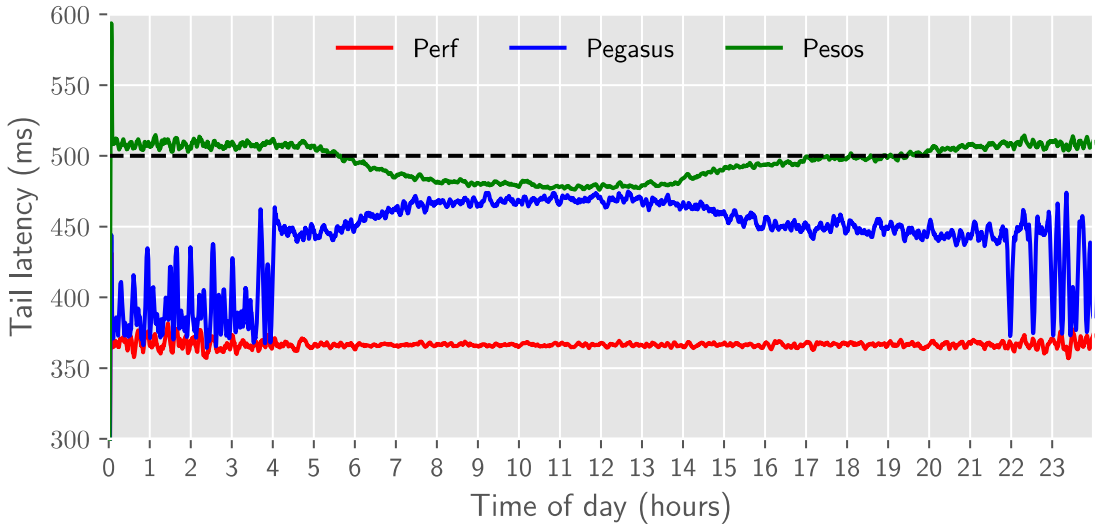

Figure 4 The 30-seconds moving 95th-%tile latency of PERF, PEGASUS, and PESOS, both per second and with a order 3 Savitzky-Golay smoothig over a ten minutes time window.

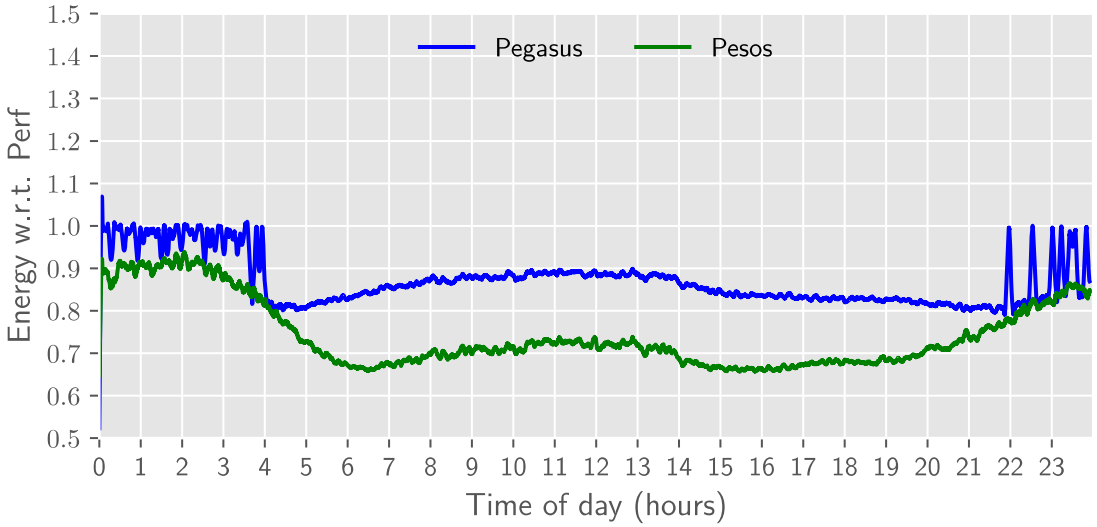

Figure 5 The 30-seconds moving 95th-%tile latency of PERF, PEGASUS, and PESOS, both per second and with a order 3 Savitzky-Golay smoothig over a ten minutes time window.

To answer RQ2 and RQ3, we initially discuss the latencies of our simulated WSE in its various configurations. Then, we analyse the CPU energy consumption of PEGASUS and PESOS w.r.t. PERF to answer RQ1 and RQ3.
In Figure 4 we observe that both PEGASUS and PESOS are closer than PERF to the 500 ms SLO. PERF exhibits a tail latency of ~350 ms at the cost of a large energy consumption. While PESOS tail latencies are closer than PEGASUS' ones to the target SLO, we also observe that PESOS leads to small latency violations in the early and late hours. This happens because the WSE receives fewer queries during night-time than during day-time; therefore PESOS tends to select small core frequencies. This results in previously unobserved latency violations, but which emerge at scale due to the variability of response times across different shard servers. Nevertheless, such limited violations (on average 509 ms) do not negatively affect the user experience.
In the same periods of the day, PEGASUS processes queries much faster than necessary, likely because PEGASUS "is a conservative policy that aims to slowly reduce the power limit without causing any SLO violations along the way". Therefore, PEGASUS fails to capture full advantage of small workloads to save energy, to not

hinder the system responsiveness. This is why the CPU energy consumption of PEGASUS is similar to PERF in early and late hours, as shown in Figure 5. Energy savings up to ~20% w.r.t. PERF are observable during the rest of the day

Conversely, PESOS can save ~10% of CPU energy consumption w.r.t. PERF early and late in the day. However, this comes at the cost of small latency violations as shown in Figure 1. During the rest of the day, PESOS remarkably reduces the CPU energy consumption by ~30% w.r.t. PERF, while keeping the WSE's tail latency just below 500 ms. In fact, query workload is intense during midday, and PESOS correctly selects large core frequencies in such situation.

Given these results, we can conclude that PESOS helps reducing the CPU energy consumption of a distributed WSE (RQ1). In our simulations, the CPUs in a day consume 254.02 MWh using PERF, while with PESOS the consumption reduces to 179.26 MWh (-29%). The same CPUs consume 218.40 MWh with PEGASUS, meaning that PESOS consumes 18% less energy than PEGASUS (RQ3). However, such energy savings comes at the cost of negligible latency violations when workload is scarce (RQ2), while PEGASUS never violates the target SLO (RQ3)

5 FEDERATED LEARNING IN DISTRIBUTED PLATFORMS

In this section, we detail the new content added as an update of this deliverable at M33. We first describe Federated Learning (FL), i.e., the leading approach to learn machine learning models in distributed environments under minimal data transfer requirements, to support the learning of the models behind the Predictive Data Analytics tools of the BigDataGrapes platform. We then propose the novel application of quantization techniques in Federated Learning to reduce the volume of data exchanged between the machines to reduce the cloud costs of the data-transfers. We finally evaluate the proposed contributions on a public dataset to assess the efficiency and effectiveness impact of our proposals when learning state-of-the-art machine learning models.

5.1 FEDERATED LEARNING

The huge amount of data collected through the BigDataGrapes platform asks for effective and efficient solutions to learn the machine learning models supporting the Predictive Data Analytics tools of the platform. Moreover, since the data of the different vineyards may be located on different machines that could be far from each other to keep the storage machines close to where the data are gathered, e.g., through devices such as drone units or various sensors placed inside the vineyard, it is convenient to limit the volume of data transferred through the network to reduce the cloud costs and avoid that the construction of the machine learning models slowdowns also the Predictive Data Analytics tools of the BDG platform. A recent approach to overcome this problem is to decentralize the computation where data are, i.e., on edge machines gathering the data directly from the close devices. Edge computing extends the Cloud computing paradigm, according to the above decentralized approach: data processing and storage capabilities are not exclusive characteristics of centralized data centers. Instead, an additional layer, called edge, is placed in the middle between the Cloud and the devices. Federated Learning (FL) is the leading approach adopted in this scenario to learn machine learning models exploiting the locality of the data. Currently, neural network models have been successfully applied to time-series tasks such as stock forecasting, natural language processing, and sequential image processing, where they achieve state-of-the-art performance. Moreover, the neural network training algorithm is well suited for being applied in a Federated Learning approach. For this reason, we focus on neural network models in the following discussion. Algorithm 1 and Figure 6 illustrate the main phases of the Federated Learning algorithm when applied to neural network models. The main steps are: every edge machine i) receives a partially trained neural network model W from a cloud node, ii) refines the model τ times based on the data provided by the local devices, then iii) sends the locally-refined model back to the cloud node. After the tree steps, the cloud node iv) aggregates all the locally-refined models and generates a new global model, then v) sends the global model back to the edge machines for a new round of local training. By doing so, the complex fusion of machine learning models on a cloud node is decoupled from the storage of training data and the heavy-computation performed by the edge machines. Moreover, it avoids the transmission of the huge amount of data within the cloud as the only information transferred between the machines is the machine learning model.

Learning state-of-the-art models using Federated Learning is challenging because the neural network models may occupy several GBs of space, e.g., certain state-of-the-art neural network models for natural language processing and image recognition. For this reason, we investigate the introduction of quantization techniques, i.e., a particular class of techniques for compressing neural network models, in the Federated Learning scenario to improve the efficiency in terms of data exchange between the edge machines and the cloud node.

Input : n local datasets $D_1, \ldots, D_n$ at edge servers $C_1, \ldots, C_n$
a number of rounds T
a number of epochs τ
Output: A matrix W of NN weights
FL$(D, D_1, \ldots, D_n, T, \tau)$:
1 Matrix W is randomly initialized
2 **for** T rounds **do**
3 **for** $i \leftarrow 1$ **to** n **do**
4 S sends W to C_i as W_i
5 **for** τ epochs **do**
6 $W_i \leftarrow \text{TRAIN}(W_i, D_i)$
7 C_i sends W_i to S
8 $W \leftarrow \sum_{i=1}^{N} \frac{|D_i|}{|D|} W_i$
9 **return** W

Algorithm 1 Federated Learning algorithm.

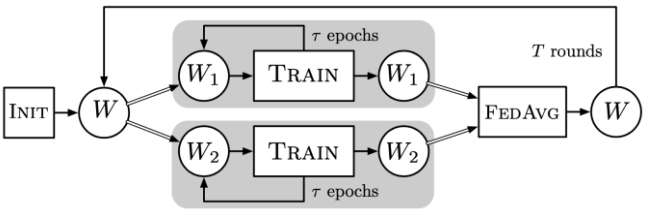

Figure 6 Main phases of the Federated Learning algorithm on two edge machines and a cloud node.

5.2 FEDERATED LEARNING WITH QUANTIZATION

The introduction of Federated Learning algorithms to learn the machine learning models employed by the Predictive Data Analytics tools of the BigDataGrapes platform poses new challenges in term of volume of data exchanged between the machines involved in the computation. Since cloud providers also bill according to the volume of traffic, this aspect can easily become a relevant cost for the BDG platform. For this reason, here we explore new solutions to reduce the volume of data exchanged between the machines in the Federated Learning scenario to reduce the cloud costs of the data-transfers.

Neural network (NN) models are made up of multiple hidden layers of neurons connected through weighted connections amplifying or weakening each signal from one neuron to the other. The weights of the connections are trained so that the whole network accurately approximate a given target function. The weights connecting all neurons of a layer to all neurons of the next layer are represented with weighting matrices of floating-point numbers. However, neural networks are often over-parametrized to ease the training process, e.g., state-of-the-art natural language processing models employ hundreds of millions of connections and occupy several GBs of space. Neural network quantization is a class of approaches aiming at computing a representation $\widehat{W}$ of the neural network weights W with a smaller memory footprint. Such approaches constrain the precision of floating-point weights to a fixed- point representation, by using a small number of bits. In particular, k-bit binary quantization maps each weight of a NN model to $\{-1, +1\}^k$. For instance, the binary quantization introduced by Courbariaux et al. [courbariaux2015] assumes $k = 1$ and the NN model W is represented using only one bit per weight plus two floating points identifying the bits zero and one. This quantization scheme can be easily extended to k bits, and an effective algorithm achieving this goal is IterQ by Xu et al. [xu2018].

We propose to improve the efficiency of FL by reducing the amount of data being transferred from the edge machines to the cloud node. To do so, we propose to apply quantization schemes to the models being transferred during the execution of the Federated Learning algorithm illustrated in Algorithm 1Figure 6. In particular, we propose to perform a first quantization before the global model is transferred from the cloud node to the edge machines and a second quantization after the model is updated by the edge machines and sent back to the cloud node. Algorithm 2 illustrates our proposed federated learning with quantization (FLQ) algorithm. FLQ aims at obtaining a NN model W while reducing the volume of data (in bytes) transferred among the cloud node and the edge machines. In the FLQ algorithm, a Federated Learning round is now made up of 6 steps: i) the cloud node applies quantization to its global model (line 3), ii) the cloud node sends its quantized global model to every edge machines (line 5), iii) each edge machines refines its own local model, starting from the received quantized global model, on its own data (line 7), iv) each edge machines applies quantization to its locally trained model (line 8), v) each edge machines sends its quantized locally trained model back to the

cloud node (line 9), and vi) the cloud node applies aggregates the quantized locally-refined models (line 10). The procedure `QUANTIZE(W)` performs the quantization of the matrix W encoding the NN model. In principle, it may implement any of the quantization approaches available in the literature.

Input : n local datasets $D_1, \ldots, D_n$ at edge servers $C_1, \ldots, C_n$
a number of rounds T
a number of epochs τ
Output: A matrix W of NN weights
FLQ $(D, D_1, \ldots, D_n, T, \tau)$ **:**

1 Matrix W is randomly initialized
2 **for** T rounds **do**
3 $W \leftarrow \text{QUANTIZE}(W)$
4 **for** $i \leftarrow 1$ **to** n **do**
5 S sends W to C_i as W_i
6 **for** τ epochs **do**
7 $W_i \leftarrow \text{TRAIN}(W_i, D_i)$
8 $W_i \leftarrow \text{QUANTIZE}(W_i)$
9 C_i sends W_i to S
10 $W \leftarrow \sum_{i=1}^{n} \frac{|D_i|}{|D|} W_i$
11 **return** W

Algorithm 2 The proposed FLQ algorithm.

Input : n local datasets $D_1, \ldots, D_n$ at edge servers $C_1, \ldots, C_n$,
a number of rounds T
a number of epochs τ
Output: A matrix W of NN weights
ΔFLQ $(D, D_1, \ldots, D_n, T, \tau)$ **:**

1 Matrix W is randomly initialized
2 **for** $i \leftarrow 1$ **to** n **do**
3 S sends W to C_i as W_i
4 $\Delta W \leftarrow 0$
5 **for** T rounds **do**
6 $W \leftarrow W + \Delta W$
7 $\Delta W \leftarrow \text{QUANTIZE}(\Delta W)$
8 **for** $i \leftarrow 1$ **to** n **do**
9 S sends ΔW to C_i as ΔW_i
10 $W_i \leftarrow W_i + \Delta W_i$
11 $W_i' \leftarrow W_i$
12 **for** τ epochs **do**
13 $W_i' \leftarrow \text{TRAIN}(W_i', D_i)$
14 $\Delta W_i \leftarrow \text{QUANTIZE}(W_i' - W_i)$
15 C_i sends ΔW_i to S
16 $\Delta W \leftarrow \sum_{i=1}^{n} \frac{|D_i|}{|D|} \Delta W_i$
17 **return** W

Algorithm 3 The proposed ΔFLQ algorithm.

Since quantization can introduce large errors in the model weights in later rounds due to the smaller and smaller changes applied to the weights during training, we also propose ΔFLQ, a variant of the FLQ algorithm, illustrated in Algorithm 3. The ΔFLQ algorithm applies the FLQ algorithm not to the full models, but on their difference before and after local training, at the edge machines, and before and after federated averaging, at the cloud node. Initially, a random matrix is initialized on the cloud node and distributed to all edge machines (lines 1- 3). Then, at every round, the ΔW matrix is quantized (line 7) and transferred to the edge machines (line 9). Each edge machine sums this matrix to its local model W_i (line 10), and then it performs the local training on a copy W_i' of its local model (lines 11-13). At the end, each edge machine computes the ΔW_i matrix, i.e., the difference between the local model after and before the training, quantizes it and transfers it back to the cloud node (lines 14-15). Finally, the cloud node computes the weighted average of the received differences and stores it in the ΔW matrix (line 16).

In the next section, we experimentally measure the accuracy of the final learned models produced by FLQ and ΔFLQ with respect to other training algorithms, such as local learning and federated learning, and we evaluate the benefits of our FLQ and ΔFLQ algorithms at reducing the data transmitted among the cloud node and edge machines.

5.3 ASSESSMENT ON PUBLIC DATA

We now present a comprehensive analysis of the performance of the FLQ and ΔFLQ algorithms by investigating two main research questions (RQs):
RQ1. What is the impact of the FLQ and ΔFLQ algorithms on the accuracy of the learned models, measured in terms of validation and test losses?
RQ2. What is the reduction in terms of data transmitted between edge machines and cloud node by the FLQ and ΔFLQ algorithms w.r.t. standard FL approaches, i.e., without quantization?

To investigate our RQs, we explore the following three scenarios:

1. **Local Learning (LL)**: the model is trained locally on the cloud node using all training data, which thus require to transfer all data to the cloud node.
2. **Federated Learning (FL)**: the model is trained on all edge machines using the local the training data applying the standard FL algorithm.
3. **Federated Learning with Quantization (FLQ)**: the model is trained on all edge machines using the local the training data applying the FLQ and ΔFLQ algorithms.

We first assess the performance of the Federated Learning scenario (FL) with respect to the Local Learning scenario (LL) to evaluate the impact of federated learning on the model performance, i.e., the effect of working on partitioned training data. We then apply quantization in the federated scenario (FLQ) to assess the impact of quantization techniques. In particular, we first assess the performance of our FLQ algorithm, then we assess the performance of our ΔFLQ algorithm. Finally, we compare the size of the models transmitted in the federated scenarios to quantify the data volume reductions yielded by the quantization schemes considered.

5.3.1 Experimental Setup

We experiment FLQ and ΔFLQ on the next word prediction task that, given a sentence context, i.e., a sequence of words, aims at learning a language model, i.e., a probability distribution over the words conditioned on a given context, to predict the next most likely word that appears after the context. This task can be seen as a time series prediction problem where data points are words in a given vocabulary, and given the first n words, we aim at predicting the $(n+1)-$th word. It is a common use case used in mobile applications like, for example, predictive keyboards or personal voice assistants. In our scenario, we assume that every edge machine has a private collection of text written by the user, and we aim at learning a global language model without disclosing the private collections.

Dataset. We conduct the experimental evaluation using the public dataset WikiText-2, which is composed of 720 text articles: 600 articles in the training set and 60 in both the validation and the test sets, respectively. The training set consists of 2,088,628 tokens, while the validation and test sets consist of 217,646 and 245,569 tokens, respectively. The vocabulary consists of 33,278 words. In the following analysis, we use the training set to train our models while the validation and test sets are used for early stopping the training of the model, and to measure its final performance, respectively.

Neural Architecture. We perform next word prediction by training a Long Short-Term Memory Network (LSTM), a recurrent neural network usually employed for solving the next word prediction task. The LSTM network is the core model employed as a predictive analytics tool in one pilot of the BDG, i.e., the price prediction task of the food safety pilot. In our experiments, we train a LSTM with an input embedding layer consisting of 200 neurons, two hidden LSTM layers, each one composed of 512 neurons, and an output linear layer with 33,278 outputs, one per word in the vocabulary. We use a batch size of 100 words. The training of the network is regularized with learning rate decay, i.e., if the validation loss does not decrease in an epoch, the learning rate, initially set to 20, is decreased by a factor of 4, up to a minimum value of 10^{-4}. For regularization purposes, we also employ a dropout strategy by setting the dropout rate to 0.5. Moreover, to address the gradient explosion problem when training the LSTM, we set the gradient clipping to 0.25, the gradient norm clipping to 0.3 and

the weight clipping to 1.0. The training of the LSTM is performed by minimizing the cross-entropy loss until the learning rate decreases below the minimum value of 10^{-4} (early-stopping condition) or for a maximum of 80 epochs.

Implementation Details. The experimental framework is implemented in PyTorch 1.4.0. Experiments are performed using a Tesla T4 GPU on an AMD EPYC CPU clocked at 2.2GHz and 24GB of RAM. The machine works as both cloud node and edge machine and we experiment with 2 edge machines. In the evaluation of federated scenarios, edge machines are emulated by training the LSTM models locally on the machine on equally-sized and disjoint partitions of the training set. All the quantization strategies tested work by independently quantizing the weight matrices of the LSTM. No quantization is performed on the input embedding layer.

5.3.2 Experimental Evaluation

We perform our experiments by reporting the effectiveness of the LSTM in terms of: i) test loss on the final model and ii) validation loss observed during the training process to analyze the convergence speed of each method. The size of the models is computed by counting the number of bits of both the quantized and un-quantized parameters of our LSTM. Local and federated scenarios employ different portions of the training data, depending on the number of workers and perform different local training epochs and federated training rounds. The total wall-clock time required to train the different models depends on four factors: i) the number of workers processing their own portions of the dataset, ii) the number of learning epochs used by each worker to locally learn its model, iii) the number of federated rounds used to collect the local models and to compute their federated averaging, and iv) the quantization process. In the following analysis, we measure the training time in time units. We assume that the time required to perform a single training epoch on a single worker on the whole dataset corresponds to 1 time unit. On w workers, the whole dataset, divided among all workers, requires $1/w$ time units to be processed in a single learning epoch in parallel on w workers. On w workers, τ training epochs require τ/w time units to perform the local training. Finally, by measuring the time required by the quantization, and the model transmission processing, we found out that their overhead is negligible with respect to the time required to perform a single learning epoch. For this reason, we do not take it into account.

Local Learning and Federated Learning Scenarios. We now propose an experimental analysis of the performance of the LSTM network when trained both in LL and FL scenarios. Figure 7 reports the validation loss of the LSTM during the training in the FL scenario according to Algorithm 1 by varying τ in $\{1, 2, 4, 8, 16\}$, i.e., the number of epochs performed by two edge machines before transmitting the model to the cloud node for federated averaging. When training in Local scenario (LL, bold red line), the validation loss drops significantly very soon, reaching the minimum value of 4.73 after 29 units of time. Moreover, the plot shows a clear trend: the more epochs are performed locally to each worker in a round of federated learning (FL), i.e., larger values of τ, the more time units are needed to converge to the minimum validation loss. The rationale of this behaviour lies in the role of federated averaging, i.e., (FEDAVG), which allows to share the knowledge learned by the two workers in isolation in a single LSTM model that is then used locally by edge machines to continue the learning. Moreover, in the FL scenario, the neural network achieves a better performance in terms of validation loss than the LL corresponding loss with small values of τ, i.e., τ in $\{1, 2, 4, 8\}$. This does not hold for $\tau = 16$, where the FL minimum validation loss does not outperform the corresponding loss of the LL scenario. The rationale behind the better performance shown in the FL scenario with respect to the LL scenario relies in the federated averaging step that introduces a regularization effect in the resulting merged model, thus achieving a better generalization of the learned model. To conclude, federated learning allows to gain effectiveness on the final LSTM model with the big advantage of keeping data stored on edge machines in a decentralized manner.

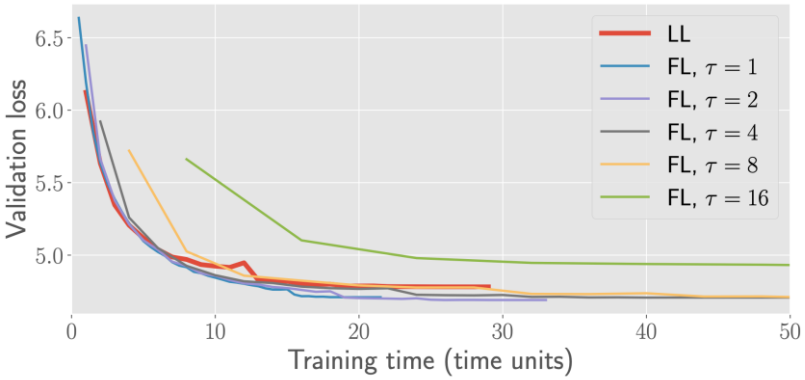

Figure 7 Validation loss of the LSTM network in the FL scenario by varying the number of epochs τ in a round of federated learning. We also report the performance on the LL scenario (red bold line).

Federating Learning with Quantization: FLQ and ΔFLQ. We now focus our analysis on the impact of quantization on the effectiveness of the model when employed in the Federated Learning scenario. For this assessment, we employ the quantization algorithm IterQ using $k = 2$ bits per neural network weight. We already observed in the previous analysis an improvement of the effectiveness of the LSTM model when employed in this setting (FL). We now evaluate the impact of the intrinsic error due to the compact approximated representation of the neural network weights (FLQ) and of the neural network weights deltas (ΔFLQ).

Table 1 Minimum validation and test losses of the LSTM network model in the Local Learning (LL), Federated Learning (FL), Federated Learning with Quantization (FLQ) and Federated Learning with Delta Quantization (ΔFLQ) scenarios, by varying the number of local training epochs τ.

	LL	FL					FLQ					ΔFLQ				
τ	-	1	2	4	8	16	1	2	4	8	16	1	2	4	8	16
Val. loss	4.78	4.71	4.69	4.71	4.70	4.81	6.95	6.90	6.89	6.85	6.83	6.13	5.74	5.43	5.17	5.05
Test loss	4.73	4.64	4.64	4.63	4.66	4.76	6.89	6.86	6.85	6.80	6.77	6.06	5.66	5.36	5.10	4.99

We report in Table 1 the minimum validation loss and the test loss achieved by all algorithms explored so far. We also report the performance of the various Federated Learning algorithms by varying the number of local training epochs τ in $\{1, 2, 4, 8, 16\}$. The reported results show that while FL does not achieve better performance increasing the number of local training epochs, the FLQ shows better results when increasing this parameter. In detail, the performance of the models trained with FLQ (test loss 6.77) are worse than the performance of the models trained with FL (test loss 4.76). We ascribe this result to quantization, since it can introduce errors in the model weights. In particular, during the training of the LSTM, the high variance in the small changes applied to the weights at later rounds can have a negative impact on the whole training and increasing the local training epochs only partially overcomes this problem. To mitigate this effect, in the previous subsection, we proposed the ΔFLQ algorithm, a variant of the FLQ algorithm. The ΔFLQ algorithm quantizes the changes in the network weights exchange between the edge machines and the cloud node. The introduction of quantization of the deltas improves the performance in the FLQ algorithm. Indeed, ΔFLQ outperform the performance achieved using FLQ for all values of τ considered. Moreover, when increasing τ, the validation loss gets closer to the value achieved in the LL and FL scenarios.

To conclude on RQ1, we experimentally showed that while FLQ (the simplest form of federated learning with quantization) underperforms with respect to the local (LL) and the federated learning (FL) scenarios, the proposed ΔFLQ algorithm allows to train an LSTM network with performance similar to an LSTM network trained in the local scenario (LL), thus overcoming the problems introduced by the quantization. In particular, the FLQ and ΔFLQ algorithms train the LSTM model with a 14% and 5% degradation in the test loss, respectively, which is a negligible cost if they allow to reduce the volume of data transferred in the cloud.

Analysis of Model Transmission Costs. We now evaluate the reduction in terms of data transmitted between the edge machines and the cloud node by the FLQ/ΔFLQ algorithms with respect to the FL algorithm without quantization. Our LSTM model is composed of 27,249,264 parameters in total. As reported above, we do not apply quantization to the input embedding layer, whose weight matrix contains 6,655,600 elements. The other layers store 20,593,664 model weights, which are represented according to our quantization schemes. By storing each parameter as a 32-bit floating point number, the LSTM network requires 0.10GB, while the 2-bit quantization scheme IterQ, which uses 2-bit per model weight, stores the quantized network in 0,03GB with a space reduction of 3.4 ×.

Table 2 Rounds to reach the minimum validation loss and volume of data exchanged between the cloud node and the edge machines (in GB) for the training of the LSTM network model in the Local Learning (LL), Federated Learning (FL), Federated Learning with Quantization (FLQ) and Federated Learning with Delta Quantization (ΔFLQ) scenarios, by varying the number of local training epochs τ.

	LL	FL					FLQ					ΔFLQ				
τ	-	1	2	4	8	16	1	2	4	8	16	1	2	4	8	16
Rounds	27	37	25	19	27	28	17	12	7	8	7	5	5	5	5	5
Data vol.	-	14.8	10.0	7.6	10.8	11.2	2.04	1.44	0.84	0.96	0.84	0.6	0.6	0.6	0.6	0.6

We report in Table 2, for each number of local training epochs τ, the number of rounds required to reach the minimum validation loss. This number represents the optimal number of rounds to obtain the best LSTM network performance during training according to the validation loss, i.e., before the validation loss starts increasing and the local models start overfitting on the local data. The number of rounds determines how many times the LSTM model is transferred from an edge machine to the cloud node and vice-versa. In general, the amount of data transferred in the cloud is as follows: $model\_size \times num\_workers \times num\_rounds \times 2$.
Table 2 also reports the total amount of data exchanged among the cloud node and the edge machines in the Federated Learning scenarios, i.e., FL, FLQ and ΔFLQ. Regarding the FLQ algorithm, both the number of rounds to achieve the best performance and the volume of data exchange decrease when increasing τ, i.e., the number of local training rounds performed on each edge machine. With $\tau = 16$, FLQ attains a test loss of 6.77 in 7 rounds (56 time units) with a total of $0.03\text{GB} \times 2 \times 7 \times 2 = 0.84\text{GB}$ transferred, while FL attains a test loss of 4.76 in 28 rounds (28 time units) with a total of $0.10\text{GB} \times 2 \times 28 \times 2 = 11.2\text{GB}$ transferred. Regarding ΔFLQ, it always produces a LSTM network with performance close to the FL algorithm without quantization in very few rounds, namely 5, with just 0.6GB of data transferred. With $\tau = 16$, ΔFLQ gets a test loss of 4.99 in 5 rounds (20 time units) with a total of $0.03\text{GB} \times 2 \times 5 \times 2 = 0{,}6\text{GB}$ transferred.

To conclude on RQ2, we experimentally showed that both FLQ and ΔFLQ algorithms are able to largely reduce the data transferred between the cloud node and the edge machines in a federated learning scenario. In particular, when used with the 2-bit IterQ quantization scheme and using $\tau = 16$ epochs for local training at each worker: i) the FLQ algorithm trains the LSTM model with a 14% degradation in the test loss while reducing by a factor of 13 × the total data transmitted over the network during federated learning, and ii) the ΔFLQ algorithm trains the LSTM model with just a 5% degradation in the test loss while reducing by a factor of 19 × the total data transmitted over the network during federated learning.

6 SUMMARY

The BigDataGrapes platform has all the characteristics of an OnLine Data-Intensive (OLDI) system. Indeed, the platform is distributed in nature, it works on large datasets, and it must be responsive. During the development of the platform, it would be desirable to test different algorithms and solution, to optimize resource usage and to be sure that the system has low latencies and low energy consumption before its deployment on a real hardware infrastructure.

For this reason, in this document we introduced OLDI Simulator, a java library to simulate OnLine Data-Intensive systems to study their performance in terms of latency and energy consumption.

In this document we illustrated the characteristics of the OLDI systems and we describe our java library, which can be used to model and simulate them. Then, we showed how to use the library by using an example, in which 1) we describe a system to simulate, 2) we model the system, and 3) we evaluate and compare various solutions by simulation means. Finally, we discussed how our simulator can be used to compare energy-efficient scheduling algorithms in distributed big data platforms, using a distributed Web search engine as an exemplary OLDI system, and real-world datasets, which are a good representative of different workloads and operations on a distributed OLDI system such as the BigDataGrapes platform.

The second revision of this deliverable (due M33) presents a novel research tool enabling the efficient training of machine learning models in a Federated Learning scenario, i.e., a family of distributed learning algorithms avoiding the transfer of huge amount of data within the cloud nodes, to improve the usage of distributed computing resources. We explored the Federated Learning paradigm for learning predictive data analytics tools and we proposed two different federated learning quantization algorithms, namely FLQ and ΔFLQ, to reduce the amount of traffic exchanged among edge machines and the cloud nodes. As an example case study, we conducted experiments on public data by training a state-of-the-art neural network model to solve the next word prediction task, which is a well-known task in Natural Language Processing. The assessment shows that our contribution reduces the data exchanged between the cloud machines up to 19 × with a minimal impact on the test loss of the final model of about 5%. Such promising results allow us to reduce the most the cloud costs of the BigDataGrapes platform due to the data transfer among the cloud nodes at the charge of a negligible loss in the effectiveness of the trained models.

References

[arapakis2014] I. Arapakis, X. Bai, and B. B. Cambazoglu. 2014. Impact of Response Latency on User Behavior in Web Search. In Proc. SIGIR. 103–112.
[barroso2013] L. A. Barroso, J. Clidaras, and U. Hölzle. 2013. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines (2nd ed.). Morgan & Claypool Publishers.
[courbariaux2015] M. Courbariaux, Y. Bengio, and J.-P. David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In Proc. NIPS, 3123–3131.
[cikm2018] M. Catena, O. Frieder, N. Tonellotto. 2018. Efficient Energy Management in Distributed Web Search. In Proc. CIKM.
[pesos2017] M. Catena, N. Tonellotto. 2017. Energy-Efficient Query Processing in Web Search Engines. IEEE TKDE 29, 7, 1412–1425.
[pegasus2014] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. 2014. Towards Energy Proportionality for Large-scale Latency-critical Workloads. In Proc. ISCA. 301–312.
[xu2018] C. Xu, J. Yao, Z. Lin, W. Ou, Y. Cao, Z. Wang, and H. Zha. 2018. Alternating multi-bit quantization for recurrent neural networks. in Proc. ICLR.