Online Network Traffic Characterization

Deliverable
Progress on the ONTIC Big Data Architecture

ONTIC Project
(GA number 619633)

Deliverable D2.3
Dissemination Level: PUBLIC

Authors

Fernández Cantón Enrique, Martínez Bravo de Medina Álex, Ordozgoiti Rubio Bruno, Vakaruk Stanislav, Zhu Matias Bo (UPM);
Fernando Arias, Vicente Martín (EMC);
Stefanos Anastasiou, Athanasios Tzaferis, Panos Georgatsos, Harris Augoustidis, Konstantinos Galatianos, Vassilis Karavassilis (ADAPTIT);
Daniele Apiletti, Elena Baralis, Silvia Chiusano, Tania Cerquitelli, Paolo Garza, Luigi Grimaudo, Fabio Pulvirenti (POLITO)

Version
ONTIC_D2.3.2016.06.09.2
Version History

<table>
<thead>
<tr>
<th>Version</th>
<th>Modification date</th>
<th>Modified by</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>V.1</td>
<td>2015.12.04</td>
<td>Adaptit</td>
<td>Skeleton version</td>
</tr>
<tr>
<td>V2-s</td>
<td>2016.01.15</td>
<td>Adaptit</td>
<td>Review version edited contribution</td>
</tr>
<tr>
<td>V3-Polito</td>
<td>2016.01.20</td>
<td>Polito</td>
<td>Document review</td>
</tr>
<tr>
<td>V3-CNRS</td>
<td>2016.01.20</td>
<td>CNRS</td>
<td>Document review</td>
</tr>
<tr>
<td>V3-CNRS-UPM</td>
<td>2016.01.29</td>
<td>UPM</td>
<td>Clarifications on review comments</td>
</tr>
<tr>
<td>V4-fin</td>
<td>2016.01.29</td>
<td>Adaptit</td>
<td>Final version with review comments incorporated</td>
</tr>
<tr>
<td>V4-fin-bis</td>
<td>2016.05.24</td>
<td>Adaptit</td>
<td>Added section 8.3 on Google Cloud Platform selection following reviewers comments</td>
</tr>
<tr>
<td>V4-fin-bis-Ericsson</td>
<td>2016.06.06</td>
<td>Ericsson</td>
<td>Updated document review</td>
</tr>
<tr>
<td>V4-fin-bis-CNRS</td>
<td>2016.06.06</td>
<td>CNRS</td>
<td>Updated document review</td>
</tr>
<tr>
<td>V4-fin-upd</td>
<td>2016.06.09</td>
<td>Adaptit</td>
<td>Final updated version with review comments incorporated</td>
</tr>
</tbody>
</table>

Quality Assurance:

<table>
<thead>
<tr>
<th>Role</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality Assurance Manager</td>
<td>Alberto Mozo Velasco (UPM)</td>
</tr>
<tr>
<td>Reviewer #1</td>
<td>Fabio Pulvirenti (POLITO)</td>
</tr>
<tr>
<td>Reviewer #2</td>
<td>Juliette Dromand (CNRS)</td>
</tr>
<tr>
<td>Reviewer #3</td>
<td>Miguel-Angel Monjas (ERICSSON)</td>
</tr>
</tbody>
</table>
# Table of Contents

1. **ACRONYMS AND DEFINITIONS** ............................................. 8
   1.1 Acronyms ............................................................................. 8

2. **EXECUTIVE SUMMARY** ...................................................... 10

3. **SCOPE** ............................................................................... 11

4. **INTENDED AUDIENCE** ..................................................... 12

5. **SUGGESTED PREVIOUS READINGS** ................................. 13

6. **THE ONTIC BIG DATA ARCHITECTURE** ........................... 14
   6.1 Scope .................................................................................. 14
   6.2 Big Data Challenges .......................................................... 14
   6.3 System View ....................................................................... 15
   6.3.1 Overall Description ........................................................ 15
   6.3.2 System Deployment ........................................................ 16
   6.4 Technology View .................................................................. 18
   6.4.1 Big Data Technologies and Selection Criteria .................... 18
   6.4.2 Spark: The ONTIC Technology Choice .............................. 20

7. **THE SPARK DISTRIBUTED COMPUTING FRAMEWORK** 21
   7.1 Overview ............................................................................. 21
   7.2 Key Notions ........................................................................ 21
   7.2.1 Application Operations .................................................... 21
   7.2.2 Jobs, Stages and Tasks ..................................................... 22
   7.2.3 Degree of parallelism ....................................................... 22
   7.2.4 RDD Lineage ................................................................... 23
   7.2.5 Application Driver and Executors ..................................... 23
   7.2.6 Deployment Modes .......................................................... 24
   7.2.7 Cluster Manager ............................................................. 24
   7.2.8 Spark on Yarn ................................................................. 25
   7.3 Spark vs. Hadoop Map-Reduce ........................................... 25
   7.4 Working experience ............................................................ 25
   7.5 Performance Tuning ............................................................ 26
   7.5.1 Overall ............................................................................ 26
   7.5.2 Configuring Computing Resources for Spark on Yarn .......... 27
   7.6 Industry Adoption .............................................................. 31

8. **CLOUD-BASED ONTIC PLATFORM** ..................................... 33
   8.1 Scope ................................................................................. 33
   8.2 Requirements ..................................................................... 33
List of Figures

Figure 1: The ONTIC Big Data Architecture ................................................................. 15
Figure 2: ONTIC Big Data analytics in NSP domains .................................................. 17
Figure 3: Distributed computing frameworks for Big Data ......................................... 18
Figure 4: Big Data enabling technologies ................................................................. 19
Figure 5: ML support on Big Data frameworks ......................................................... 19
Figure 6: Spark stack .............................................................................................. 21
Figure 7: Example of Spark stages and tasks ............................................................ 22
Figure 8: Distributed execution of a Spark application ............................................ 24
Figure 9: ONTIC Google Cloud-based platform overview ...................................... 36
Figure 10: Data storage options .............................................................................. 36
Figure 11: The PICS algorithm ............................................................................... 42
Figure 12: Parallelized EM algorithm for PPCA ...................................................... 43
Figure 13: The PRANKS algorithm .......................................................................... 44
Figure 14: PPCSS with different number of workers ............................................... 45
Figure 15: PPCSS with different dataset sizes .......................................................... 45
Figure 16: Ratio to the SVD .................................................................................... 46
Figure 17: Accumulated leverage scores for the top-5 features .............................. 46
Figure 18: PICS with different numbers of workers ............................................... 47
Figure 19: PICS with different data set sizes ............................................................ 47
Figure 20: TM Forum Big Data Analytics Reference Architecture .......................... 48
List of Tables

Table 1: D2.3 Acronyms........................................................................................................... 8
Table 2: Yarn properties........................................................................................................... 28
Table 3: Spark computational properties ................................................................................. 29
Table 4: Algorithm test profile ............................................................................................... 37
Table 5: The top-10 chosen features ....................................................................................... 46
# 1. Acronyms and Definitions

## 1.1 Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Defined as</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ACL</td>
<td>Access Control List</td>
</tr>
<tr>
<td>ALS</td>
<td>Alternating Least Squares</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BI</td>
<td>Business Intelligence</td>
</tr>
<tr>
<td>BSS</td>
<td>Business Support Systems</td>
</tr>
<tr>
<td>CAPEX</td>
<td>Capital Expenditure</td>
</tr>
<tr>
<td>CEM</td>
<td>Customer Experience Management</td>
</tr>
<tr>
<td>CIF</td>
<td>Common Intermediate Format</td>
</tr>
<tr>
<td>CSSP</td>
<td>Common Subset Selection Problem</td>
</tr>
<tr>
<td>DAG</td>
<td>Direct Acyclic Graph</td>
</tr>
<tr>
<td>DoW</td>
<td>Description of Work</td>
</tr>
<tr>
<td>DPI</td>
<td>Deep Packet Inspection</td>
</tr>
<tr>
<td>DWH</td>
<td>Data Warehouse</td>
</tr>
<tr>
<td>ELT</td>
<td>Extract, load, transform</td>
</tr>
<tr>
<td>ES</td>
<td>ElasticSearch</td>
</tr>
<tr>
<td>ETL</td>
<td>Extract, transform, load</td>
</tr>
<tr>
<td>FCBC</td>
<td>Fast Correlation-Based Filter</td>
</tr>
<tr>
<td>HAWQ</td>
<td>HAdoop With Query</td>
</tr>
<tr>
<td>HDFS</td>
<td>Hadoop Distributed File System</td>
</tr>
<tr>
<td>HPC</td>
<td>High Performance Computing</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>IA</td>
<td>Information Architecture</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java Database Connectivity</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>MCMC</td>
<td>Monte Carlo Markov Chain</td>
</tr>
<tr>
<td>MOA</td>
<td>Massive On-line Analysis</td>
</tr>
<tr>
<td>ML</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>MR</td>
<td>Map-Reduce</td>
</tr>
<tr>
<td>NFV</td>
<td>Network Function Virtualization</td>
</tr>
<tr>
<td>NLANR</td>
<td>National Laboratory for Applied Network Research</td>
</tr>
<tr>
<td>OpenSOC</td>
<td>Open Security Operations Center</td>
</tr>
<tr>
<td>OPEX</td>
<td>Operating Expense</td>
</tr>
<tr>
<td>ORC</td>
<td>Optimized Row Columnar</td>
</tr>
<tr>
<td>OSS</td>
<td>Operations Support Systems</td>
</tr>
<tr>
<td>PCA</td>
<td>Principal Component Analysis</td>
</tr>
<tr>
<td>PCAP</td>
<td>Packet Capture</td>
</tr>
<tr>
<td>QoE</td>
<td>Quality of Experience</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>REST</td>
<td>Representational state transfer</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>RTT</td>
<td>Round Trip Time</td>
</tr>
<tr>
<td>SDN</td>
<td>Software Defined Networking</td>
</tr>
<tr>
<td>SPE</td>
<td>Streaming Processing Engines</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>SVD</td>
<td>Singular Value Decomposition</td>
</tr>
<tr>
<td>SVM</td>
<td>Support Vector Machines</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>WEKA</td>
<td>Waikato Environment for Knowledge Analysis</td>
</tr>
<tr>
<td>YARN</td>
<td>Yet Another Resource Negotiator</td>
</tr>
</tbody>
</table>
2. Executive Summary

The ONTIC Big Data Architecture refers to a Big Data computing system tailored for network service provider (NSP) domains, which is able to run ML algorithms for traffic classification to the benefits of network performance and business intelligence. It constitutes an essential aspect and an important outcome of project work. Not only does it bind development to a common realization framework but it also provides practical means for putting the developed ML algorithms into operation in NSP domains.

Deliverable D2.3 describes the ONTIC Big Data Architecture in terms of system capabilities and realization technologies emphasizing on applicability and performance aspects. It consists of three main parts; the overview, the technology and the feature engineering part.

The first part presents the design of the architecture, its applicability in NSP domains and the adopted Big Data computing technology - it is noted that the research focus of the project is on ML analytics rather than on the underlying Big Data storage and processing technologies. The proposed architecture provides capabilities for capturing network traffic traces, their mediation and the engineering of a suitable set of features to fuel ML algorithms, and finally the distributed execution of the algorithms in an off-line and/or on-line fashion. It can be integrated with existing network management systems in NSP domains for enhancing traffic engineering capabilities. At the same time, it can form a sustainable information pipeline feeding operations and business levels with accurate and unbiased use/usage metrics and actionable KPIs. Apache Spark is the selected technology for realizing the required Big Data computing infrastructure on grounds of performance and versatility. Spark offers in-memory processing capabilities; it can seamlessly combine different data processing models; it supports a wide range of Big Data storage systems; and, it can run on already deployed clusters or in the cloud.

The second part elaborates on technological aspects. The key design aspects of the Spark cluster computing framework are presented, while the differences with Hadoop Map-Reduce design are highlighted. Drawing on gained knowledge and working experience, the critical aspect of optimizing application performance on a Spark cluster is discussed. The parameters for configuring the Spark engine and the underlying resource managers, according to application needs, are outlined and initial guidelines are provided.

The instantiation of the proposed architecture in the Google Cloud Platform is also described in terms of the technologies used. The project has committed, in its DoW, to using available enterprise clouds for the purpose of undertaking large-scale experimentation and exercising its architecture and developed algorithms in realistic production environments.

Last, the third part focuses on feature engineering. Note that all ML algorithms operate on an input set of ‘feature vectors’ - traces with defined attributes - and as such, their complexity is greatly affected by dimensionality. Appropriate procedures and suitable algorithms for feature reduction are presented along with experimentation results.

D2.3 is an update of D2.2, released January 2015, that presented the landscape of available Big Data computing frameworks, benchmark tests for aiding technology decision and an initial view of the architecture and feature engineering algorithms. D2.3 expands D2.2 in the latter aspect by presenting the evolution of the architecture in terms of capabilities, technologies adopted and feature engineering algorithms, also by describing its applicability in NSP domains, performance tuning and cloud instantiation. All these aspects will be continuously refined during project work and they will be finally documented in D2.6, due end of the project.
3. Scope

This Deliverable is part of WP2 of the ONTIC project. WP2 deals with the design of the ONTIC Big Data architecture and the realization of a system for provisioning, managing and making publicly available the ONTIC Network Traffic Summary (ONTS) dataset with traces from real ISP network flows.

Deliverable D2.3 refers to the architectural aspect of WP2 work, updating D2.2 where an initial view of the ONTIC Big Data architecture was presented following a comprehensive survey of related frameworks and benchmark tests. D2.3 presents a revised view of the architecture and the technology selected for its realization, emphasizing on deployment and performance aspects. Enhanced algorithms for feature selection, replacing the introductory algorithms presented in D2.2, and related results are also included. D2.3 sets the grounds for the development work in the project on off/on-line ML algorithms (WP3, WP4) and demonstration use cases (WP5).

The work along architectural aspects will be concluded and documented in the final WP2 Deliverable, D2.6, due to end of the project.
4. Intended audience

This Deliverable is of particular interest to all partners of the ONTIC project, not only those working on architectural and data capture issues but equally those working on the specifications, development and testing of ML algorithms and use cases. This is because the Deliverable presents the grounds on which the project is being developed - the algorithms and use cases should fit the specified overall system architecture, its application scenarios and the selected realization technology.

Furthermore, the Deliverable targets data scientists and software engineers by presenting the intrinsic design and technology aspects underlining the Big Data architecture proposed by ONTIC.
5. Suggested previous readings

The information provided in this Deliverable is explained in such a way that no previous specific knowledge is needed apart from a basic background in the fields of information management, software engineering and networking. Familiarity with available Big Data projects and technologies could be useful.

For avoiding information duplication explicit references are made to the following project Deliverables detailing specific topics related to the ONTIC Big Data Architecture.


6. The ONTIC Big Data Architecture

6.1 Scope

The ONTIC project aims at developing, assessing and demonstrating the applicability of innovative Big Data mining and machine-learning algorithms for network traffic classification/analysis. Broadly speaking, traffic classification refers to a set of algorithms for categorizing traffic or identifying normal/abnormal patterns and trends.

The work of ONTIC falls into the realm of Big Data analytics that is growing at fast pace over the last years across several sectors such as telecoms, healthcare, IoT and genomics to name a few. Specifically, in the telecoms sector, the focus of ONTIC, the market of Big Data analytics is expected to grow 40% per year on average until 2017 according to credible business reports.\textsuperscript{1}

Machine-learning (ML) algorithms for traffic classification are a continuously developing research area [17]-[23]. However, their integration into the everyday network and service provisioning operations in network service provider (NSP) domains is at an early stage. Their application is mainly exhausted in characterizing application traffic (e.g. personal, social, spam emails) and in recommendation systems. As an indicative example, Google produced last year a white paper on how machine-learning could be used to optimize data center operations\textsuperscript{2}.

In the above scenery of increased business potential, the ONTIC Big Data Architecture aims at providing a readily deployable system at NSP domains for running ML algorithms for traffic classification on Big Data to the benefit of network performance and business intelligence. The definition of such architecture entails at a large extent a technology selection: the technology for realizing the required Big Data storage and processing capabilities.

The ONTIC Big Data Architecture is described from a system viewpoint in the following, while the technology adopted for its realization is presented after. First, the challenges stemming out from handling Big Data in a telecoms environment and the datasets to be used by the project are discussed.

6.2 Big Data Challenges

The term ‘Big Data’ is widely used to stress the underlying properties of the data that nowadays become available for analysis. The so-called ‘three Vs’ model defines these properties as follows: \textit{volume}, amount of data; \textit{velocity}, increasing rate of input data; and, \textit{variety}, diversity of data ranging from unstructured to structured data. It is the combination of these three Vs that makes Big Data different from what was before.\textsuperscript{3}

Whereas data processing was driven by pre-defined schemas (detailed information models) before, Big Data processing drives the schemas. Multiple data analytics algorithms may need to apply to the same data collection or arbitrary combinations of them or even to newly available data collections. Light-weight information models, as required by each algorithm, need to be dynamically attached to input data for avoiding the complexity and overhead of a chain of explicit mediations as it was usually the case before. The velocity property necessitates the processing of streams of data in addition to batches, ideally through the same infrastructure means. \textit{A new breed of data storage, processing and analytics systems and strategies is required.}

\textsuperscript{1} WiKibon, Big-Data Analysis Report, August 2013.
\textsuperscript{2} https://googleblog.blogspot.gr/2014/05/better-data-centers-through-machine.html
\textsuperscript{3} Note that additional V’s may be attributed to Big Data such as ‘variability’, denoting change in other data characteristics (http://bigdatawg.nist.gov/_uploadfiles/NIST.SP.1500-1.pdf).
Telecoms, the area of concern to ONTIC, fully fledge all Big Data properties. Data indeed comes in variety; in unstructured forms from sensors or social-media feeds; in semi-structured forms from measurement probes or various Web-server logs; as well as in structured forms from more traditional data sources, like accounting servers, mediation devices or CRM and billing systems. The proliferation of mobile devices, sensors of any kind, content and data-demanding services and applications available in the Web fuels data volume and velocity; Internet traffic is expected to grow on average by 23% every year from 2014 to 2019⁴.

ONTIC considers datasets containing IP network traffic traces. The project has already started to collect such a dataset, the ONTS (ONTIC Network Traffic Summary) dataset, from the operational environment of an ISP [31]. In addition, the project will use reference network traffic datasets available in the public for comparatively testing its developed algorithms.

The off-line traffic classification algorithms (WP3) have to deal with the volume property of Big Data whereas the on-line ones (WP4) with the velocity property. For exemplifying the utility of the developed algorithms, a number of use-cases are developed (WP5) addressing typical areas of concerns to service providers - anomaly detection, congestion avoidance and adaptive QoE management. All these will be implemented in the ONTIC Big Data Architecture which is described in the following.

6.3 System View

6.3.1 Overall Description

Figure 1 presents an overall view of the ONTIC Big Data Architecture depicting the flow of information from raw data to analytics results. As it can be seen, its Y-form, which draws from the Lambda architecture software paradigm [16], allows the seamless combination of off- and on-line analytics. Moreover, it adheres to the TM Forum reference architecture for Big Data analytics (Figure 20, Annex A) as its components can directly map to the specified functional layers of data ingestion, management and analysis.


The Data Capturing block obtains raw traffic information from the core network by capturing IP packets. This block is part of the ONTIC Provisioning System and is described in detail in [31]. Captured raw data is then processed for creating an optimized set of features (attributes) according to application needs, and subsequently, the off/on-line algorithms apply for deriving the desired analytics.

Depending on application case, Pre-Processing mediates the information included in the raw data to create the necessary observation points and their features as required by the ML algorithms to apply. It may involve things like mappings (e.g. packets/flows may be mapped to particular traffic types) and aggregations, numeric values normalization and scaling. A typical pre-processing example is the first part of the unsupervised anomaly detection algorithm (UNADA), developed by WP4, where packets may be aggregated per destination and/or source IP address. Essentially, the Pre-Processing block allows domain experts to customize the application of ML algorithms in terms of their input information.

The Pre-Processing block is an addition over the previous version of the architecture. Its design should be modular enough to allow the flexible incorporation of different pre-processing functions. It is not of algorithmic nature; however, the project will look into factoring out generic pre-processing functions to the end of providing a complete realistic solution for ML-based traffic classification analytics in NSP domains.

The Feature Engineering module cleans the features extracted from the raw data by removing any redundant information that may be identified. Subsequently, it selects the key features and returns the best possible set for processing. Feature selection and thus reduction is of increased value as the complexity of ML algorithms grows exponentially with the number of features. Suitable algorithms developed by the project for feature selection are presented in chapter 9. The clinched data is now ready for processing.

Data processing can be performed either on-line (in real-time as data arrives) or off-line. For on-line processing, the data is continuously fed into the On-line Analytics Engine, with the pre-processing and feature engineering functions being performed in real-time. For off-line processing, the data is stored in a Distributed File System which should be resilient. From there the Off-line Analytics Engine will retrieve the data as required by the algorithms that runs. Note that the stored data can at any time be streamed to the On-line Analytics Engine. This store-and-stream processing mode may be useful for performance reasons e.g. for not starving cluster resources from on-line processing, for application reasons per se e.g. for applications intrinsically requiring both off- and on-line analytics and for testing the on-line algorithms before put into production.

The project will use readily available platforms for providing the required distributed storage and processing environment as its focus is on the development of ML analytics for traffic classification, not on the underlying technology as such. The issue of selecting an appropriate distributed computing technology is dealt in the following. Next, the system applicability in NSP domains is discussed.

6.3.2 System Deployment

The proposed architecture can be realized in NSP domains as depicted in Figure 2.
As it can be seen, the ONTIC Big Data Analytics System can be largely deployed outside the network, typically in a data center of the NSP. Its lower part, the Data Capturing module, can be embedded into the network, directly attached to key network interfaces for capturing a representative set of traffic traces as required by the nature of the application cases of concern. Deliverable D2.5 [31] describes in detail the packet capturing process.

The above Figure presents a reference deployment scenario. Multiple instances of the system may exist e.g. per geographical and/or per application area. This advocates a sort of peer and/or hierarchical parallel processing paradigm, which, however, falls outside project scope; it is only mentioned for highlighting future research dimensions.

The ONTIC Big Data Analytics System can be part of closed-loop (with feedback) control schemes by appropriately interfacing with existing NMS (network management systems) usually employed in NSP domains, while at the same time it can form, in its own right, an open-loop (non-feedback) control system for providing unbiased operations and business intelligence.

By analyzing captured traffic traces through ML techniques, the ONTIC Big Data Analytics System can provide enhanced TE (traffic engineering) intelligence -for optimizing network performance while guaranteeing QoS/E at desired levels- by allowing closed-loop controls to be exerted at various time-scales and/or by utilizing information on evolution of various traffic mixes. This is to the contrary of current practices, which utilize total/average load information over specific measurement intervals mainly relying on traditional, single metric, time-series analysis. Note that the concurrent exercise of congestion control at multiple time-scales is considered in the telecoms literature as a promising approach when dealing with long-tail (highly-variable) traffic that it is the typical case in telecom environments.

At the same time, the ML-elaborated traffic analysis of the ONTIC system can lead to providing accurate statistics and forecasts on the use and usage of network resources incurred by various traffic mixes. This can be achieved by avoiding simplifying averaging calculations and man-in-the-middle interventions, as it is usually the case today. In other words, the ONTIC system, by operating on the wealth of information hidden in raw traffic data, can provide an automated unbiased information pipeline spanning engineering, middle-management and executive levels of network, service, commercial and business operations.

---

The use cases considered by the project (undertaken in WP5) for exemplifying the utility of the developed ML algorithms (in WP3 for off-line and WP4 for on-line algorithms) correspond to the above application uses. The anomaly detection case is an example of open-loop control, whereas the congestion avoidance and adaptive QoE management cases are examples of closed-loop controls, with the first applied at the edge/core and the latter at the access parts of the network.

Last, the following point is worth-noting. The advent of SDN (software defined networking) and NFV (network function virtualization) technologies and the emerging fog⁶ (lower than the cloud) computing/networking paradigm amplify the need of having a traffic analysis system such as the ONTIC Big Data Analytics System. With these technologies, the network becomes a distributed set of software controllable devices where network/service functions can be dynamically instantiated at network core, edge, access, even IoT end-points as conditions warrant so. Such an environment requires on one hand a distributed information processing infrastructure, beyond the level of a local cluster though, and, on the other hand, traffic data-driven intelligence for appropriately orchestrating the required network/service functionality; both these aspects are considered by the project, even if not in this particular network technology context. Although outside of its research scope, this application case highlights the exploitation potential of the project in the evolving telecoms scenery.

6.4 Technology View

6.4.1 Big Data Technologies and Selection Criteria

For coping with volume and velocity, Big Data requires a powerful distributed computing (storage and processing) platform. As stated, ONTIC focuses on the development of ML analytics on top of a Big Data platform, not on the development of the platform itself.

Generally speaking, three main distributed computing frameworks for Big Data can be distinguished (Figure 3): Map-Reduce, advocating a partition-aggregation processing paradigm; Graph-oriented processing, building upon graph models for data representation and analysis; and, Parallel DBs, offering SQL-like Big Data query interfaces.

Figure 3: Distributed computing frameworks for Big Data

Figure 4 presents a glimpse of the landscape of related technologies, mostly those supported by open source Apache projects. Each technology presents its own set of data abstractions, APIs, file formats and related management and utility tools [1]-[15]. Please refer to Deliverable D2.2 [30] for a comprehensive description of these technologies, including also relevant benchmark tests.

---

It is noted that the majority of Big Data computing technologies provide support for ML analytics through libraries implementing commonly-used ML algorithms. Figure 5 depicts known ML libraries available in the open source, distinguishing between distributed/non-distributed implementations and off-line/on-line analytics. The fact that there exist different ML libraries for different parallel processing technologies shows that the efficient development of distributed versions of ML algorithms remains a key research topic, while it raises the question of which Big Data technology attains the best performance overall. The ambition of ONTIC is to find its algorithms and/or its implementations integrated in the ML suite of the chosen technology.

ONTIC is primarily interested in the Map-Reduce paradigm because of its maturity that made it become the common place of parallel implementations of ML algorithms. Evidently, the technology to be selected should meet at least the following common-sense criteria:

- efficiency in performance and robustness in operations,
- cost-effectiveness by elastically scaling up/down on commodity hardware,
- rich set of APIs facilitating the development of ML algorithms, and
- manageability and support by an ecosystem of open tools.

The technology choice of the project is presented next.

6.4.2 Spark: The ONTIC Technology Choice

ONTIC selected Apache Spark [2] for realizing the distributed Big Data computing platform of its architecture. The choice is primarily based on the performance potential and versatility of Spark. More specifically:

- Spark provides for in-memory processing capabilities, which makes it outperform the classical Hadoop Map-Reduce implementation especially for iterative data processing applications; typically, ML algorithms, including the ones to be developed by the project, are of iterative nature. The project has made related benchmark tests during the first year [30] and the results verify the performance benefits of Spark (see also section 7.4). Chapter 7 discusses key performance-improving aspects of the Spark design.

- Spark seamlessly supports through a common set of basic abstractions different data processing models even within the same application, batch and stream, SQL-like and graph-oriented processing, while it provides for an up-to-date ML library (MLlib). This application versatility makes Spark a truly multi-purpose distributed processing platform, accelerating the development of ‘culturally’ diverse analytics through the same technology. This aspect is of significant importance to the project as in telecoms, the target application domain of the project, data processing and analytics diversification is amplified by the dynamics of input data and the required agility in multiple decision levels.

- Spark retains the rich and well-proven Hadoop- and Apache-based ecosystem of Big Data file systems (HDFS, Hive, HBase, Cassandra) and formats, and associated querying tools (HiveQL, Pig), while it provides native interfaces to other distributed storage or file systems like Amazon S3 and Elasticsearch. This is another key facet of Spark’s versatility that enables porting to and interoperation with already and newly deployed clusters. It is essential for the project as it facilitates sustained deployment of the developed algorithms and the management of their input/output datasets. The default choice of the project regarding the cluster file-system is HDFS.

- Spark decouples application processing and cluster management concerns by interfacing to a variety of resource managers. This operational versatility enhances the capability of optimizing application performance along with cluster resources. It is essential for the project for assessing the performance of the developed algorithms in various cluster configurations and for ‘best’ tuning the performance of the algorithms in target deployment clusters. Please see section 7.5 for more information.

Spark being an active Apache project is constantly developing in terms of capabilities and interfaces with emerging file systems and analytics libraries. Furthermore, Spark is well supported in terms of documentation and distributions; all major Hadoop distributions available as open source or enterprise editions, like Cloudera’s CDH, Hortonwork’s HDP and Pivotal’s PHD, support Spark.

Finally, the choice of relying on Spark is justified by current business practices. As it becomes apparent from section 7.6, Spark has already received intense industrial attention. In five years, Spark has moved from the lab to become part of commercial offerings and an integral component of the core computing platforms of large-scale service providers.
7. The SPARK Distributed Computing Framework

7.1 Overview

As stated in its official Web-page [2], “Apache Spark is a fast and general engine for large-scale data processing”.

On the speed side, Spark has been optimized to perform not only on disk but also for in-memory computations. It allows datasets to be loaded in the memory once and queried many times. This characteristic makes Spark well suited for iterative machine-learning algorithms, outperforming in these cases Hadoop’s disk-based Map-Reduce engine.

On the generality side, Spark provides a unified stack of data processing models - batch and streaming, SQL and Graph-based. While previously, all these different workloads would require separate distributed engines, in Spark they are tightly integrated around the same core engine and the same basic programming abstractions (Figure 6). As a result, they can be combined in a streamlined and inexpensive way, which is mostly required in production data analysis pipelines. Furthermore, Spark can run on a variety of cluster managers and can access diverse Big Data sources such as Hadoop HDFS, Apache HBase, Hive and Cassandra, OpenStack Swift, Amazon S3 or Elasticsearch.

Spark is designed to ease application development by offering a comprehensive set of APIs and rich built-in libraries with bindings in Scala, Python, Java and R.

![Spark Stack](image)

Figure 6: Spark stack

Drawing on gained working experience, the following sections elaborate on the Spark cluster computing framework from the standpoint of an application developer. The intention is not to offer a programming guide but to highlight the intrinsic design and configuration aspects influencing application performance. For a seminar treatment of Spark and its APIs refer to the documentation in the official site [2] and available textbooks [24]-[27].

7.2 Key Notions

7.2.1 Application Operations

A Spark application consists of a set of operations for manipulating and processing datasets, arranged in a particular order according to the specific logic of the application. Spark operations fall into two general categories: transformations and actions.

Transformations mediate the information included in datasets thus they are used to generate new (transformed) datasets. Typical examples of transformations that can apply to a single dataset include filter (for selecting data), map (for creating new data types) and variousByKey operations (for performing aggregations on key-value paired datasets). Transformations may also apply to a pair of datasets for performing basic set operations such as the union, join (for keyed datasets), intersect, subtract and cartesian transformations.
Actions process datasets for ‘doing something’ - compute values e.g. sums, return data to the ‘main’ function of the application or output data in a specified file-system such as HDFS. Typical examples of actions include count (for getting the number of elements in a dataset), take/collect (for getting a part or the entire dataset), reduce/aggregate (for computing a result value based on a supplied function), lookup (for getting the values of a specific key) and various saveAs operations (for storing a dataset).

The Spark engine executes the application operations in parallel across a cluster of locally interconnected nodes, with each node having its own computing (CPU, RAM) and storage resources.

### 7.2.2 Jobs, Stages and Tasks

Spark adopts a lazy-evaluation approach whereby application execution is driven by actions. That is, transformations are not performed until an action is encountered; the actions force the evaluation of the transformations required for producing the datasets where they apply. As such, the execution of an application in Spark involves the execution of a number of jobs each corresponding to an action.

The execution of a Spark job evolves through a number of sequential or parallel stages each encompassing a subset of the transformations required towards computing the dataset to which the corresponding action applies. Sequential stages are designated by transformations that require data-shuffles i.e. transformations that require moving data around cluster nodes. Piped-lined transformations like filters and maps are included in the same stage, whereas data-shuffling transformations like group-by transformations initiate a new stage. Parallel stages are distinguished by transformations that involve multiple datasets, like joins.

The execution of a stage involves the execution of a number of tasks which actually implement the operations included in the stage. Transformations and actions are split into a number of tasks according to the number of partitions into which the datasets that they are called upon are divided -note that in a distributed environment datasets are not stored in their entirety in a single node but in partitions across different nodes. The tasks per operation are executed in parallel across the nodes of the cluster.

Figure 7 exemplifies the above by presenting the stages of a job involving typical Spark operations.

### 7.2.3 Degree of parallelism

Based on the above, in the context of an application, the degree/level of parallelism is essentially determined by the number of partitions of the datasets being processed. Each stage
corresponds to a fixed number of partitions and the tasks -for implementing the specified
operations during the stage- are executed in parallel on these partitions.

Evidently, the degree of parallelism can affect application performance. More parallelism will
usually lead in faster execution times, however too much parallelism may create bottlenecks
especially at input/output operations as well as significant overhead that may result from the
accumulation of the small overheads associated with partition management and task scheduling
in a distributed environment.

Spark enables applications to regulate the partitions of the processed (input and derived)
datasets and therefore the degree of application execution parallelism. First, by allowing specify
the desired number of partitions for the dataset to be produced in transformations involving
data shuffles -not pipelined transformations like filters and maps. And second, through explicit
transformations: repartition (for creating a specific number of partitions), coalesce (for
decreasing the number of partitions) and partitionBy (for hash- or range-partition a key-value
dataset with a supplied function).

7.2.4 RDD Lineage

As outlined, the execution of a Spark application results in the parallel execution of tasks over
stages per job (action) specified by the application. To this end, Spark builds and maintains the
lineage of each dataset included in the computations i.e. its dependencies from previous
datasets and the respective transformations, in the form of a DAG (directed acyclic graph) and
schedules the execution of the required tasks over stages for realizing the required
transformations and finally the actions.

Because Spark maintains the lineage of each dataset, datasets can be recomputed in case (part
of) their data is lost e.g. on node/disk failures, provided that the input dataset can be retrieved
from the file system -note that HDFS is a reliable file system offering a reliability level of 2 or 3.
For emphasizing this design feature, Spark uses the term reliable distributed datasets (RDDs)
for characterizing datasets.

7.2.5 Application Driver and Executors

To put the reliable and lazy-evaluation approach for running applications into effect, Spark uses
the master/slave paradigm, with one central coordinator, called the driver, and many
distributed workers, called the executors.

The driver runs the main functionality of the application. It splits application functionality into
jobs, stages and tasks and schedules the execution of the resulting tasks in the distributed set of
executors based on data closeness criteria. It also builds and maintains the lineage of the
datasets (RDDs) involved for ensuring resilient execution.

The executors run the individual tasks over stages per job as scheduled by the driver, exchanging
data between them (in case of data-shuffle operations), or communicating data back to the
driver (in case of actions) or interfacing with the file-system for data input/output.

Spark allows applications cache in-memory and/or disk intermediately derived RDDs for later
use, through a service called the Block Manager that lives within each executor.

The driver and executors of a single or different applications run in separate JVMs. Their
sizing in terms of CPU and RAM, may be specified by the application at run-time (through the
sparkConf object) or through command line options when invoking the application. If such
specifications are not provided, Spark uses the defaults settings.
7.2.6 Deployment Modes

While executors are instantiated inside cluster nodes, the driver may not necessarily run in the cluster, but in a machine (locally) connected to the cluster. In the client deployment mode, Spark launches the driver on the machine where the application was invoked, whereas in the cluster deployment mode, the driver is shipped to run on a node in the cluster. The latter mode is typical for running applications in cloud-based Spark clusters.

7.2.7 Cluster Manager

Spark does not allocate itself the required cluster resources for instantiating the application executors and the driver. Instead, it assumes that a pluggable cluster manager is responsible for managing cluster resources and as such, it issues requests to the cluster manager for instantiating the required application execution environment (the executors and the driver if in cluster mode).

Spark can use a number of cluster managers, Spark Standalone, Hadoop Yarn and Apache Mesos, each having its own philosophy in distributed resource allocation and management. The Spark machinery is truly agnostic to the underlying cluster manager. Different cluster managers may be specified for different applications at application invocation times.

In Yarn, the exact number of executors per application needs to be specified together with the required number of cores per executor—all executors have the same computing profile. In Standalone and MESOS, the total number of cores for all executors needs to be specified and the cluster manager determines the number of executors to instantiate; Standalone tries to maximize the number of executors for the specified number of cores, whereas MESOS tries to minimize this number. In either cluster manager, the RAM per executor needs to be specified and, if in cluster mode, the cores and RAM of the driver need also be specified.

Generally speaking, cluster managers rely on master-slave architectures with the terms master and worker denoting the centralized and distributed portions of their architecture. Without loss of generality, it can be assumed that Spark executors and drivers, if in cluster mode, run in worker nodes. Multiple executors may run in the same worker node on behalf of the same or different Spark applications. As such, the resources (cores, memory) of executors and drivers requested by Spark applications should fit the dimensioning of worker nodes in terms of allowable cluster resources.

Figure 8 presents an overview of the components involved for executing a Spark application in a cluster, summarizing the discussion in sections 7.2.5 - 7.2.7.
7.2.8 Spark on Yarn

In Yarn, the master part, called Resource Manager, overlooks cluster resource allocation by enforcing resource limits; obviously, these limits should be in accordance to the available cluster resources. It spawns an Application Master to supervise resource allocation per Spark application. The Application Master in turn requests from the Node Managers running in worker nodes the appropriate resources according to the Spark application driver, if in cluster mode, and executors’ requirements. Note that in cluster mode the Spark application driver runs inside the Application Master.

The resources per worker node allocated by Yarn are generically referred to as containers - bucket of cores and memory- and they correspond to JVMs. So, when running Spark on Yarn, in addition to the system-specific Resource and Node Manager processes, a number of containers (JVMs) need to be instantiated per application: one container for running the Application Master and the driver, if in cluster mode, and as many containers as the Spark executors requested by the application.

7.3 Spark vs. Hadoop Map-Reduce

Spark adopts the Map-Reduce (split-and-combine) parallel processing paradigm. However, it differs from the classical Hadoop implementation of Map-Reduce in the following key aspects:

- Spark allocates JVMs (through the cluster manager) on the basis of application executors not on the basis of individual application tasks as in classical Hadoop.
- Spark allows caching even in-memory of intermediate datasets for speeding up the execution of iterative operations applying on the same dataset; in classical Hadoop all tasks start from the input dataset i.e. from the disk, requiring a lot of disk i/o.

Evidently, both aspects - avoiding run-time resource allocation and enabling in-memory processing- impact positively on application performance. Benchmark tests carried out by the project [30] verify the improved performance of Spark over Hadoop.

In addition, the lazy evaluation feature of Spark reduces the number of Map-Reduce passes over data and removes associated complexity from the application developers. Developers may no longer be concerned with casting application logic to a bunch of carefully sized map and reduce tasks for grouping together operations so that to minimize the number of Map-Reduce passes as it is typical in Hadoop. In Spark, these concerns are taken over by the engine; operations, small or large, are chained together and executed through internal performance-optimized mechanisms e.g. shuffle cashing. As a result, Spark offers a richer and more easy-to-use API, leaving developers free to organize their applications around smaller, more manageable operations.

7.4 Working experience

In this section, we share our experience from working with Apache Spark and Apache Hadoop in research and experimentation.

When running a Spark application, even when only using the standard MLlib (the Spark Machine Learning Library) algorithms, it was necessary to manually configure certain technical parameters such as the number of executors, the number of cores and the memory assigned to each executor. The default configuration settings, in fact, do not allow exploiting all the resources available in the cluster. Furthermore, even if configuration heuristics exist, certain parameters may still need to be tuned up taking into account the specific processing needs of each application. The determination of the ‘best’ configuration settings lies in multiple dimensions - CPU, RAM, data shuffling, serialization, etc.- and it is far from being a trivial problem. The inherent difficulty is amplified when considering complex data mining and
machine-learning algorithms such as item-set mining and clustering. Because of its crucial importance, the issue of Spark configuration is further discussed in section 7.5.

From the point of view of programming language, Spark allows to use Scala, Java and Python (and very recently, R). Scala is the most optimized language, allowing containing very complex operations in a few commands. Python is also powerful in this aspect while it has a rich set of built-in libraries; however, Python APIs usually follow Scala and Java releases. As a result, new Spark APIs can only be used within Scala applications for the first months; as an example, we mention the Association Rules algorithms in Spark 1.5.2. Furthermore, Scala is the language that is officially used to develop Spark. As such, in cases requiring the modification or the use as baseline of one of the Spark APIs or MLlib algorithms, it is mandatory to use Scala. Alternatively, one could develop from scratch the needed implementation in the selected programming language.

The documentation and examples of the Spark API usage are often provided as commands in the Spark interactive shell, rather than as complete sample Spark applications. As a consequence users need to find out themselves the dependencies and details of putting together their Spark code. In any case, the Spark interactive shell can connect to the cluster making it very helpful for fast development, testing and exploratory data analysis.

Regarding ML support in Spark and Hadoop, we find that the algorithms in Mahout (Hadoop ML library) are easy-to-use off-the-shelf. Mahout implementations are already compiled and available to be used from command line. MLlib implementations are offered only as APIs; as such, users who are not confident with programming or the Spark interactive shell are not able to use them. This advantage is mitigated by the requirement of some Mahout applications (e.g., K-means clustering algorithm) to convert input datasets into the Hadoop Sequence-File format before processing.

Last on the performance side, Spark is found to outperform Hadoop. In addition to the initial benchmark tests [30] carried out at the beginning of the architectural activities with the k-means algorithm, recent experimentation involved Hadoop- and Spark-based implementations of the Parallel FP-growth algorithm. The Spark implementation proved to have the best performance, during the core mining step, while it achieved a much better load balancing across cluster resources. However, we have noticed that a non-negligible amount of time was spent in moving the results from a temporary folder to the final output folder. This indicates that disk in/out operations are expensive and applications should avoid the transfer of data within the cluster.

7.5 Performance Tuning

7.5.1 Overall

This section focuses on the critical aspect of Spark configuration for optimizing application performance. The configuration process is first put into a perspective for understanding the scope and effect of the various configuration parameters. Then, key parameters are outlined and their settings are discussed.

For optimizing the performance of Spark applications given the resources of a cluster, there are three layers to consider:

- The Spark application
- The Spark engine
- The cluster manager (Standalone, Yarn, Mesos)

The Spark engine -based on the operations included in an application- determines the required computational tasks over stages per application job and requests from the cluster manager to
launch an appropriate application execution environment - the executors and the driver if in cluster mode - based on relevant configuration information. Following, the cluster manager instantiates the requested resources into appropriate computational blocks, JVMs, in the cluster, taking into account available resources and related configured limits. Once the required computational blocks are instantiated, the engine sends to them the application tasks for execution. This way parallel execution is achieved, however it is important to note that this involves three players: the application, the engine and the cluster manager.

In addition to being multi-leveled, the tuning of application performance is multi-faceted. In addition to properly sizing-up execution resources (cores, memory) other essential aspects of operations influencing application performance in a distributed environment need to be appropriately set-up, such as data shuffling, serialization and compression, encryption, logging and security. For instance, the data shuffling aspect is uplifted in applications, like ML applications, requiring operations on massive key-value datasets. A full account of the available configuration parameters, at Spark engine level, can be found in the official documentation pages\(^7\).

Obviously, application performance tuning should a) exploit cluster resources, b) be consistent both vertically across the above layers and horizontally across different aspects of operations, and c) take explicitly into account intimate knowledge about the processing needs of the application. A ‘one-fits-all’ configuration can hardly prove efficient for the diversity of Spark applications.

As the above show, the configuration of a Spark cluster computing environment is a multi-dimensional, not straightforward, application-dependent task. The project will look at this issue for optimizing the performance of the developed off/on-line algorithms through appropriate settings as well as through code refinements based on a deeper understanding of the Spark internals - the technology analysis presented in this chapter is to this end. The purpose is to be fair to the proposed algorithms by avoiding performance downfalls due to misconfigurations and inefficient development. In this context, work along the Spark configuration dimension will continue until the end of the project as part of the development and experimentation activities.

In the following, we concentrate on the configuration of the available computing resources (cores and memory) in cluster nodes for achieving fast application execution, assuming that Spark will run on a Yarn cluster manager.

### 7.5.2 Configuring Computing Resources for Spark on Yarn

To the end of optimizing application performance, the goal is threefold:

- To fully exploit the computing resources (cores, memory) available in cluster nodes by appropriately configuring Yarn.

- Given the available cluster resources, to appropriately size the capacity of the application executors and, if in cluster mode, of the application driver in terms of cores and memory, through the relevant Spark properties.

- Given executors’ sizing, to appropriately set the number of partitions in the stage-border transformations (input, shuffling-causing) and actions, which actually determines the number of tasks that need to be executed.

The above goals refer in reverse order to the three layers of configuration outlined previously. Obviously, the resources requested by Spark (for the executors and the driver, if in cluster mode) should fit into what Yarn have available which in turn should fit into what is actually available.

\(^7\) [http://spark.apache.org/docs/latest/configuration.html#spark-properties](http://spark.apache.org/docs/latest/configuration.html#spark-properties)
It is recalled that (see sections 7.2.7, 7.2.8):

- The application executors and the driver, if in cluster mode, correspond 1-1 to Yarn containers and are instantiated as JVMs in the cluster nodes; the application tasks over stages per job will run in these JVMs.
- Yarn needs to know the exact number of executors to instantiate.
- In Yarn all executors (containers) must have the same size (cores and memory).
- There may be multiple executors (containers) in a cluster node even for the same application.

The relevant configuration parameters for Yarn and Spark are listed in Table 2 and Table 3, respectively:

<table>
<thead>
<tr>
<th>Yarn Property</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>yarn.nodemanager.resource.cpu-vcores</td>
<td>8</td>
</tr>
<tr>
<td>yarn.nodemanager.resource.memory-mb</td>
<td>8g</td>
</tr>
<tr>
<td>yarn.scheduler.minimum-allocation-vcores</td>
<td>1</td>
</tr>
<tr>
<td>yarn.scheduler.maximum-allocation-vcores</td>
<td>32</td>
</tr>
<tr>
<td>yarn.scheduler.increment-allocation-vcores</td>
<td>1</td>
</tr>
<tr>
<td>yarn.scheduler.minimum-allocation-mb</td>
<td>1g</td>
</tr>
<tr>
<td>yarn.scheduler.maximum-allocation-mb</td>
<td>64g</td>
</tr>
<tr>
<td>yarn.scheduler.increment-allocation-mb</td>
<td>512m</td>
</tr>
<tr>
<td>spark.yarn.executor.memoryOverhead</td>
<td>max(384m, 0.07*spark.executor.memory)</td>
</tr>
</tbody>
</table>

The `yarn.nodemanager.resource.*` properties refer to the Yarn Node Manager and denote the total number of resources that can be allocated to all containers to be instantiated in a cluster node—they do not refer to a particular container. As such, they effectively define the ‘total capacity’ of a Yarn-controlled cluster node. Obviously, the settings should be less than the available nodal resources and take into account the requirements of running system processes (OS, HDFS, Yarn itself, etc.).

The `yarn.scheduler.*` properties refer to the Yarn Resource Manager and denote the limits on container resources that can be requested by the Node Managers. Node Managers’ requests exceeding the maximum limits are rejected, while minimum limits apply even though lesser resources are requested. Therefore, these settings define the lower and upper bounds on the size of each container that can be requested to instantiate.

Based on the above semantics, the minimum and maximum number of containers that can be instantiated in a cluster node can be determined by dividing `yarn.nodemanager.resource.*` with `yarn.scheduler.[maximum,minimum]-allocation-*`. Note that the maximum or minimum number of containers is relevant to the type of resources—cores or memory.

The `yarn.scheduler.increment-allocation.*` properties determine the ‘step’ for sizing the containers in the sense that the actual amount of resources per container should be in multiples of these properties over the minimum.

Finally, the `spark.yarn.executor.memoryOverhead` property accounts for some memory off-heap that JVMs can use, such as interned strings and direct byte buffers.
### Table 3: Spark computational properties

<table>
<thead>
<tr>
<th>Spark Property</th>
<th>Line Option</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spark.executor.instances</code></td>
<td><code>--num-executors</code></td>
<td>2</td>
</tr>
<tr>
<td><code>spark.executor.cores</code></td>
<td><code>--executor-cores</code></td>
<td>1</td>
</tr>
<tr>
<td><code>spark.executor.memory</code></td>
<td><code>--executor-memory</code></td>
<td>1g</td>
</tr>
<tr>
<td><code>spark.driver.cores</code></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td><code>spark.driver.memory</code></td>
<td><code>--driver-memory</code></td>
<td>1g</td>
</tr>
<tr>
<td><code>spark.default.parallelism</code></td>
<td></td>
<td>Parent RDD partitions; if no parent, the total cores available</td>
</tr>
<tr>
<td><code>spark.dynamicAllocation.enabled</code></td>
<td></td>
<td>False</td>
</tr>
<tr>
<td><code>spark.scheduler.mode</code></td>
<td></td>
<td>FIFO</td>
</tr>
</tbody>
</table>

The `*.cores` properties will result in requesting from Yarn the same number of cores per container. As such, they should be in line with the `yarn.scheduler.[minimum, maximum, increment]-allocation-vcores` properties; otherwise the request is rejected and the application will not run.

The `*.memory` properties control the heap memory size per executor (container/JVM) and impact on the amount of data Spark can cache, as well as on the maximum sizes of the shuffle data structures used for grouping, aggregations and joins - such relevant properties are the `spark.[shuffle, storage].memoryFraction` properties. The value of the `spark.yarn.executor.memoryOverhead` property, discussed previously, is added to the values of the previous properties to determine the final amount of memory to be requested from Yarn for each executor (container/JVM). As already said, Yarn may round the requested memory (see `yarn.scheduler.[minimum, increment]-allocation-mb` properties before).

The `spark.driver.[cores, memory]` properties make sense mainly in cluster mode; the `spark.driver.memory` property makes sense also in client mode. These properties should not be set programmatically (through the `SparkConf` object) as the driver’s JVM has already started.

The `spark.dynamicAllocation.enabled` when set to ‘True’ (starting from Spark 1.3) avoids setting the `spark.executor.instances` property and turns on the dynamic allocation capability of Spark. This enables an application to request executors when there is a backlog of pending tasks and free up executors when idle.

The `spark.scheduler.mode` determines the discipline for scheduling jobs included in an application. With default being FIFO, it is also possible to configure fair sharing between jobs. Under fair sharing, Spark assigns tasks between parallel-running jobs (jobs submitted from separate threads) in a “round robin” fashion, so that all jobs get a roughly equal share of cluster resources. This means that short jobs submitted while a long job is running can start receiving resources right away and still get good response times, without waiting for the long job to finish. This mode is best for multi-user settings.

In the following, a procedure for configuring the above parameters so that to make use of all available cluster resources is outlined based on recommended practices:

---

1. The cores and memory per node for not starving the other system processes running in the cluster (HDFS, Hive, OS etc.) are subtracted and the relevant Yarn properties are set accordingly. It is this amount of resources that is actually characterized as available cluster resources.

2. The desired number of cores per executor is specified. This is merely an ad-hoc decision based on trials; common practices deem up to 5 cores per executor as a valid choice, for not saturating input/output throughput, however more elaborate estimates, perhaps taking into account application needs, should be provided. Evidently, this number should be less than the available number of cores per node.

3. The total number of executors and the number of executors per node are determined by dividing the available number of cores in the cluster by the number of cores per executor (determined in step 2) and the so-yielded result by the number of nodes.

4. The memory per executor is calculated by dividing the available memory per node by the previously calculated number of executors per node. Memory overheads and the step-wise nature of memory allocation (in certain multiples above a minimum), as per description of the relevant parameters, need to be taken into consideration.

5. The calculated number of executors per node (step 3) needs to be reduced by one, if in cluster deployment mode, for accounting for the application driver. The executor memory (step 4) to be allocated to the driver may need to be updated according to specific application needs.

6. The yielded settings per executor and the driver need to be checked against the Yarn settings for ensuring consistency. Steps 2-5 may need to repeat in cases the yielded numbers leave unexploited resources -note that the results of the division operations are rounded since they need to yield an integer.

7. The default level of parallelism is specified as a multiple of the number of cores per executor (step 2). With this applying to all applications, fine-grained configurations may also be done at the level of individual transformations, in the application code, by setting the number of partitions for the resulting dataset.

8. Multiple configurations may be specified (depending on settings in step 2 and 7) for selecting the one that performs best through test runs.

The above presents a bottom-up, core-centric configuration approach. It is noted that following this procedure we have verified performance gains compared to default settings.

Alternatively, a memory-centric approach (by suitably changing steps 2-4) could be considered in cases where memory is deemed essential for applications. Furthermore, additional configuration aspects should be considered e.g. regarding data shuffles as well as emerging capabilities.

As already stated the task of optimal configuration settings is not straightforward; it is application-specific and it largely requires refinements through trials (set-and-see) in the specific cluster environment. The project will investigate suitable configuration procedures for providing a fair evaluation of the performance of the developed off/on-line algorithms and the experience will be documented in the final version of the Deliverable.

Having discussed its anatomy regarding application performance, the guide to the Spark technology is concluded by presenting its business dynamics.

---

9 https://spark.apache.org/releases/spark-release-1-6-0.html
7.6 Industry Adoption

Spark was initiated and originally developed as a research project in 2009 at the AMPLab, University of California Berkeley. It aimed at creating a clustering computing framework addressing target workloads poorly served by Hadoop. The project went open source in 2010 and, in 2013, it was donated to the Apache Software Foundation, becoming a Top-Level Apache Project in February 2014. The project [2] is active, currently in release 1.6.0 as of January 2016, having gathered more than 800 developers from over 200 organizations since 2009.

Apache Spark is not only growing fast as an open source project but it has also been rapidly adopted by the industry as the prevailing distributed Big Data processing platform because of its in-memory processing capability and its highly versatile nature.

IBM recently announced their commitment to Spark. They plan to embed Spark into the core of their analytics and commerce offerings. Also, to leverage Spark in the insights platform of the IBM’s Watson Health Cloud for delivering faster results to doctors and medical researchers; furthermore, to offer Spark as a cloud service on their Bluemix platform. Finally, their plans include putting more than 3,500 researchers and developers to work on Spark-related projects at more than a dozen labs worldwide.

Related to the above, the SETI Institute, NASA and IBM have joined efforts to analyze 100 million radio events detected over several years using the ML capabilities of the Spark-as-a-service on Bluemix in a hunt of evidence of intelligent extraterrestrial life. Other Spark-based research projects, sponsored by IBM among others, can be found in [28].

Google have recently (after September 2015) started offering Spark as part of their cloud services. The Google Cloud Dataproc service allows creating a managed Spark data processing platform that can be quickly, within 90 seconds, instantiated on clusters of variable sizes, ranging from a few to hundreds of nodes. Being in beta version, we have witnessed that the service is continuously enhanced with new capabilities.

The Amazon simple storage service, S3, is seamlessly integrated with Spark. Spark sees S3 as a file-system like it sees HDFS, passing it as an argument to the native file read/write methods. As such, Spark can readily run inside of Amazon’s elastic compute cloud, EC2, with very fast data in/out performance.

Oracle announced their Big Data Appliance X5-2 at the beginning of 2015 for offering enterprise-grade solutions. With the focus being on Big Data SQL, security and data discovery, the product embeds Apache Spark and various Hadoop workloads offering “these capabilities for on premises deployment as well as a public cloud service – Oracle Big Data Cloud Service”.

Databricks, a company founded by the creators of Spark, offers large-scale Spark clusters as a cloud service, while they also provide Spark-based analytics for vertical domains such as IoT, healthcare, finance and manufacturing.

Guavus has been one of the first companies set up to exploit the power of Big Data analytics. They offer applications for planning, operations, customer experience management and IoT. Their innovative data processing platform embeds Spark and Yarn at its core. As they state

10 https://blogs.apache.org/foundation/entry/the_apache_software.Foundation_announces50
12 https://cloud.google.com/dataproc/
14 https://databricks.com
“With Spark embedded in the Guavus Reflex platform, Guavus delivers advanced analytics capabilities to customers and enables continuous real-time streaming analytics within seconds of data being collected.”

E-bay, one of the world’s largest marketplaces, have announced\(^{16}\) that they use Spark on Yarn to accelerate and simplify their data analytics for improving customer experience and creating enhanced business value. The China’s largest on-line shopping and consumer-to-consumer platform, Alibaba Taobao, have put into production\(^{17}\), since May 2014, a Spark on Yarn cluster for running their heavy duty analytics, recommendation and prediction models using GraphX and MLlib. Baidu, a leading Chinese search engine and content provider, have introduced\(^{18}\) Spark in a large-scale cluster, operating on petabytes of data, for enhancing the performance of their interactive query engine.

The above cases are indicative of the fast adoption of Spark by the industry. An expanded list of research and commercial organizations committed to Spark can be found in [29]. Clearly, these cases justify the choice made by the project regarding the technology of its architecture.


\(^{18}\) [http://www.slideshare.net/SparkSummit/how-spark-fits-into-baidus-scale-james-peng](http://www.slideshare.net/SparkSummit/how-spark-fits-into-baidus-scale-james-peng)
8. Cloud-based ONTIC Platform

8.1 Scope

ONTIC has opted, from its initial work-plan, porting its Big Data architecture in enterprise clouds for enabling large-scale experimentation beyond lab capabilities and for trying out the developed off/on-line algorithms in actual production environments. The first aspect is important for validating and comparatively evaluating the research activities of the project through credible, extensive experimentation. The second aspect contributes to the exploitation of project results since it proves the feasibility of their deployment in enterprise-grade clusters like the ones available in NSP domains.

Cloud-based experimentation allows assessing the performance of the developed algorithms in various configurations of cluster resources and parameter settings, which would be inefficient, if not practical at all, to be carried out in a lab. Specifically, the performance of the algorithms can be tested under:

- Increasing cluster sizes ranging from tens to hundreds nodes of the same type, for assessing the ‘quality of parallelism’ of the implementation.
- Various types of nodes regarding computing resources (cores, memory) such as processing- or memory-intense nodes, as a function of data sizes, for assessing the critical point(s) in the efficiency of the algorithm.
- Different configuration settings across cluster sizes and node types, for providing guidelines on the optimal performance tuning of the implementation.

8.2 Requirements

The type, capacity and amount of cloud computing (processing, storage) resources should be configurable for facilitating a rich set of experimentation scenarios e.g. scaling the number of cluster nodes and/or the cores, memory per node (see previous).

The cloud services to use should allow running standard Apache Spark distributions - the chosen Big Data technology. Furthermore, they should allow full-fledged configuration of the Spark platform according to framework specifications.

Appropriate tools for launching and deploying cloud resources in the desired configurations as required per experiment and for monitoring application performance and resource consumption should be provided.

The ONTIC platform set-up and the experiment execution process should be carefully planned and automated, to the extent possible, for gaining on cost-effectiveness and efficiency.

8.3 Cloud Platform Choice

In its work-plan, the project had opted for the AWS cluster provided through one of its partners, EMC2. However, the cluster became unavailable during the project’s second year and an alternative had to be devised. Following this development, the project selected the Google Cloud Platform19 for carrying out its experimentation work, predominantly on the following grounds:

- **Operational and management adequacy**: Google Cloud Platform services can fulfill all requirements stated in the previous section - flexible and automated cluster instantiation in different flavors of processing and storage resources, including ready-to-use Spark

19 https://cloud.google.com/
clusters, comprehensive Web-based management interfaces for monitoring and resource consumption, and integrated ‘devops’ tools. The Google Cloud Platform services used are outlined in the next section.

- **Familiarity**: project partners have prior working experience with using Google Cloud Platform technologies for development and testing; in fact, one of the project partners has been using the Google Cloud Platform for building its local instance of the ONTIC Big Data system.

Although there might be alternative clouds to use such as of Amazon, IBM, Oracle and others, which have recently started offering Spark-as-a-service (as discussed in section 7.6), the project could not afford undertaking a thorough comparative analysis between them since this would take time and resources that have not been planned. Note that there are not “killer” functionalities in the currently available clouds; all of them provide similar services and features. Thus, the project has chosen a cloud platform that the partners are more familiar with while fulfilling project requirements.

The selected Google Cloud Platform is a valid and widely accepted choice. A recent benchmark study\(^{20}\) has highly ranked the Google cloud among the clouds of Amazon, IBM, Microsoft and Rackspace, on grounds of pricing, performance and level of service.

### 8.4 Building the Google Cloud-based ONTIC Platform

Based on the general considerations laid down in section 8.2, among the various services offered by the Google Cloud Platform we opt for the following ones\(^{21}\):

- Compute Engine,
- Cloud Storage, and
- Dataproc

The above services are described in the following sections.

#### 8.4.1 The Compute Engine Service

The ‘Compute Engine’ service provides for **virtual machines (VMs)** and associated **persistent disks**. Various types of VMs are provided differing in the:

- number of virtual CPUs, or cores for simple, implemented as hardware hyper-threads on 2.3-2.6 GHz Intel platforms, with the number of cores per VM being up to 32 in powers of two, that is 2, 4, 8, 16 and 32 (beta);
- amount of RAM per each core, distinguishing between: **standard**, with 3.75 GB of RAM per core; **highmem**, with 6.5 GB of RAM per core; and, **highcpu**, with 0.9 GB of RAM per core.

As it can be noticed, VMs do not differentiate in processing capabilities -all cores have the same power- but in the number of cores and the amount of RAM per core, essentially in the amount of RAM which is shared among all cores. Therefore, considering 4 cores per VM, a ‘standard’ VM has 15 (=4x3.75) GB of RAM, a ‘highmem’ VM has 26 (=4x6.5) GB of RAM and a ‘highcpu’ VM has 3.6 (=4x0.9) GB of RAM. Pricing depends on the type of VM and charges are accounted per minute with a minimum charge of 10 minutes.

---


Persistent disk block-level storage is provided per VM in two technologies: standard and SSD. Furthermore, a local, physically attached to the server hosting the VMs, SSD disk service is provided. Pricing depends on total disk size (in GBs), technology and on disk locality, while charges are made per month on a provisioned basis, despite actual use.

8.4.2 The Cloud Storage Service

The ‘Cloud Storage’ service provides for data storage and retrieval and associated simple programming interfaces. Pricing mainly depends on the amount of stored data (in GBs) per storage option (standard, durable reduced availability, near-line) available.

Regarding data transfers, uploading (ingress traffic) is free of charge whereas data transfers from the Google Cloud to other destinations (egress traffic), including transfers between cloud regions, are chargeable. The project will make use of cloud services in the same region.

8.4.3 The Dataproc Service

The ‘Dataproc’ service is a platform service allowing the automated instantiation of a Spark platform on clusters of arbitrarily defined sizes and types of nodes. The instantiation is indeed fast—less than 90 seconds. Currently, the service has been augmented with deployment, logging, monitoring tools and initialization scripts. It makes use of the previously-mentioned services underneath, so the same charges apply plus a fixed surplus per core per hour.

8.5 Deploying ONTIC in the Google Cloud Platform

Based on the ‘Compute Engine’ service, various clusters of computing and storage resources can be instantiated. The Spark platform is then installed in the clusters, thus instantiating the ‘Off/On-line Analytics Engine’ of the ONTIC Big Data architecture (Figure 1). The other components of the architecture including the developed algorithms are ported as Spark applications.

Spark installation in smaller clusters is done manually using the Cloudera’s latest open source distribution (CDH)\(^2\). This distribution offers a comprehensive suite of Big Data platforms, data ingestion, query, scripting and workflow tools available in the Hadoop and Apache ecosystem (HDFS, Hive, Pig, Flume, Oozie, etc.), including Web-based management interfaces. The ‘Dataproc’ service may be used for instantiating ‘at once’ larger Spark clusters required for experimentation, for avoiding the effort required to install ourselves the Spark platform.

A moderately-sized Spark cluster will be maintained for preparatory tests and for determining optimal configurations. Once the applications are tested and tuned they are moved for experimentation in larger-scale clusters.

The ‘Cloud Storage’ service is used for providing additional storage space beyond the space available in the disks of the cluster, if this is required (see Figure 10, next section).

8.6 Overview of the ONTIC Google Cloud-based Platform

Based on the cloud services outlined in the previous section, Figure 9 provides a high level view of the main systems required for the operation of the ONTIC cloud-based platform.

\(2\) https://www.cloudera.com/content/www/en-us/products/apache-hadoop.html
The ‘ONTIC Spark Computing Cluster’ instantiates the ONTIC Big Data architecture (Figure 1), depicted as an overlay eclipse, in a variety of cluster configurations based on the ‘Compute Engine’ service. In these clusters, the developed off/on-line algorithms will be tested for their performance. The ‘ONTS Ingestion Manager’ ingests the captured ONTS (ONTIC Network Traffic Summary) dataset \[31\] into the platform in the form of files. The ingestion frequency could be regular ranging from higher to smaller time-scales or ad-hoc depending on experimentation needs. Synthetic data, if required, will be produced by the ‘ONTIC Data Creator’. Note that in addition to ONTS, other reference datasets required for comparative testing of the algorithms may need to be ingested.

The ‘ONTIC Data Store’, provided by the ‘Cloud Storage’ service, stores the ingested or created datasets for later processing by the off/on-line algorithms. It may also store produced results from the algorithms, if so is required. The ‘ONTIC Cluster Manager’ undertakes the launching, configuration and monitoring of the ‘ONTIC Spark Computing Cluster’. This is provided by existing packages and/or services - by Cloudera’s CDH management tools or by specialized ‘devops’ tools (like fabric8) or by the ‘Dataproc’ service tools.

In the reference model presented above, the ‘ONTIC Spark Computing Cluster’ would store only the dataset that is currently being processed, while the ‘ONTIC Data Store’ would store datasets for future use (for later processing, experimentation results etc.). However, all data could be stored in the ‘ONTIC Spark Computing Cluster’ giving rise to two options (see Figure 10). The place of storing data depends on the data volumes of the experimentation activities and of course on cost. In any case, either option can be introduced dynamically through the offered cloud services.
8.7 Running Experiments

Cloud-based experimentation requires a well-defined, manageable, not ad-hoc, and carefully-planned procedure in order to facilitate the necessary deployment activities and maximize the number of tests within the budget available. To this end, a staged-experimentation procedure is adopted, as follows.

1. The off/on-line algorithms under test are first profiled in terms of data, software and cluster requirements (Table 4).
2. The profiles are consolidated so that to obtain a clear view of the dimensioning and expected usage of cloud resources. The goal is to use a common set of cluster set-ups in terms of number and types of nodes and storage resources.
3. Based on their profiles, appropriate initialization scripts per algorithm are prepared.
4. The data required per algorithm are uploaded and staged in the permanently installed Spark cluster where initial validity and performance tuning tests are carried out.
5. Larger-scale Spark clusters (as specified in step 2) are instantiated and the algorithm applications are shipped to run for assessing their performance. In a specific cluster set-up, algorithms may run multiple times for different configuration settings.

<table>
<thead>
<tr>
<th>Table 4: Algorithm test profile</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithm</strong></td>
</tr>
<tr>
<td><strong>Datasets</strong></td>
</tr>
<tr>
<td>- Input data file names, estimated sizes</td>
</tr>
<tr>
<td>- Synthetic data file names, estimated sizes</td>
</tr>
<tr>
<td>- Provisioning mode: retrieve or on-line feed and feed rate</td>
</tr>
<tr>
<td>- Output file names, estimated sizes</td>
</tr>
<tr>
<td>- Data persistency duration</td>
</tr>
<tr>
<td><strong>Software</strong></td>
</tr>
<tr>
<td>- Programming language</td>
</tr>
<tr>
<td>- Main application file name</td>
</tr>
<tr>
<td>- Other application file names</td>
</tr>
<tr>
<td>- Required packages/libraries</td>
</tr>
<tr>
<td><strong>Configuration parameters</strong></td>
</tr>
<tr>
<td>- Command line arguments</td>
</tr>
<tr>
<td>- Cluster manager</td>
</tr>
<tr>
<td>- List of Spark properties</td>
</tr>
<tr>
<td><strong>Cluster</strong></td>
</tr>
<tr>
<td>- Min/max/increment of number of nodes</td>
</tr>
<tr>
<td>- Types of nodes (cores, RAM)</td>
</tr>
</tbody>
</table>
9. Feature Engineering

9.1 Scope and Requirements

Feature engineering is often a crucial step for the successful application of machine-learning and data mining techniques, especially when Big Data comes into play. Choosing a smaller subset of relevant features (attributes) helps to avoid the curse of dimensionality and over-fitting, reduce training times and improve model interpretability.

An expert choice is mostly helpful in defining the essential features for performing effective analytics, but this does not make feature engineering obsolete. Numerous features can be extracted from raw data, in particular in the case of network traffic analysis. Distilling the most important ones for the application case at hand is a crucial task. A combination of algorithmic feature engineering and expert knowledge is often an optimal strategy.

Algorithms for dimensionality reduction generally belong to one of the following two families:

- **Feature transformation**: These algorithms transform the data into a new space, which either captures most of the information of the original dataset according to some criterion or has certain desirable properties. This is the case, for instance, of Principal Component Analysis (PCA), Singular Value Decomposition (SVD), kernels, auto-encoders, etc. These techniques often perform very well in certain scenarios, but the models they are part of are usually hard to interpret. PCA and its applicability to network traffic analysis have been addressed during the initial architectural activities and a description can be found in [30]. An overview of the SVD from a numerical algebra standpoint is available in [32] while kernels and auto-encoders are discussed in [33] and [34], respectively.

- **Feature selection**: Feature selection algorithms try to choose that subset of the features that best represents the whole dataset in some regard. This goal is often modeled as a hard combinatorial problem and therefore the solution cannot be found efficiently. As a consequence, there exists a wealth of approximation algorithms for optimizing various feature selection criteria. This makes feature selection an even more stimulating research focus in the field of Big Data analytics.

9.1.1 Feature Engineering for Network Traffic Analysis

In the field of network traffic analysis, model interpretability is highly valued because it can lead to decisions affecting business integrity and revenues -services and customers. There are two main challenges that must be overcome to the end of achieving effective dimensionality reduction. First, the size of today’s datasets makes it infeasible to run many of the existing feature selection algorithms. Second, it is almost impossible to label Big Data-sets, effectively rendering supervised approaches useless. For being able to handle massive datasets, we have decided to focus our efforts on unsupervised feature selection algorithms.

To address the two challenges mentioned above, several parallelized unsupervised techniques for feature selection have recently surfaced, based on a variety of concepts such as information theory [35], Minimum Redundancy Maximum Relevance [36], variance preservation [37] and rough set theory [38]. Of particular interest is the approach known as the Column Subset Selection Problem (CSSP), which can be formulated as follows:
Definition 1 Column Subset Selection Problem. Given a matrix $A \in \mathbb{R}^{m \times n}$ and a positive integer $k$, let $A_k$ denote the set of matrices comprised of $k$ columns of $A$. Find $C$ such that

$$C = \underset{X \in A_k}{\text{argmin}} \| A - XX^+ A \|_F$$

where $X^+$ denotes the Moore-Penrose pseudoinverse of $X$.

The reason this problem constitutes an interesting framework for unsupervised feature selection is that its objective function is a measure of the difference between the original data matrix and the low rank approximation obtained by using a chosen set of features as basis. Therefore, using SVD as a baseline, we can directly assess the quality of the chosen feature subset. The CSSP is believed to be NP-Hard and therefore efficient algorithms for providing an exact solution are unlikely to exist.

There exist different approaches to approximately solve the CSSP problem [39], [40], but there is a lack of work on the issue of adapting these methods to datasets comprised a huge number of samples/traces. In the following, we describe key approaches for tackling this issue. We also show results from applying these algorithms on the ONTS dataset. The results suggest that the considered algorithms present a good approach for interpretability-preserving dimensionality reduction for network traffic analysis.

9.2 Processes and Algorithms for Unsupervised Feature Selection

9.2.1 Parallelized Random Algorithms

The approach of randomly sampling candidate feature subsets following a judicious probability distribution helps to achieve good theoretical results while it tends to provide good results in practice. The algorithm described in [39] is a good representative of this approach. In basic terms, in order to choose a subset of $k$ features the algorithm draws various samples of $c-k$ columns from the data matrix, performs a rank-revealing QR factorization on the top-$k$ right singular vector loadings that correspond to each of the sampled candidates and then chooses the one that minimizes the CSSP objective function.

The main challenge in applying the above method to Big Data-sets lies in the second phase. If the data matrix $A$ is big, then the column subset $C$ is big too, making it costly to compute the corresponding pseudo-inverse. Members of the ONTIC consortium recently proposed a parallelized version of this algorithm, PPCSS, [41], focusing mainly on finding the minimizing column subset. In this implementation, projection matrices are generated and then broadcast at once to compute the residual matrix norm for all of them via one map-reduce operation. Documentation and source code for our implementation can be found at https://gitlab.com/ontic-wp3/PPCSS.
9.2.2 Increasing the Efficiency and Applicability of Randomized Algorithms

The algorithm described previously presents certain drawbacks in practice. First, the approach employed for random sampling requires a target rank, which might be hard to determine in scenarios where the exact number of features to be kept is not known in advance. Second, the explicit computation of the pseudo-inverse and the matrix corresponding to each column subset is inefficient with respect to both time and memory requirements.

To overcome the above issues we propose **Parallelized Independent Column Selection (PICS)**. First, we introduce a random sampling strategy that is not dependent on the target rank.

### Column sampling phase

**Input:** data matrix $A \in \mathbb{R}^{m \times n}$, number of columns $k$

1. $\text{rows} := \text{RDD} \left( \{ A_i : i \in \{1, m\} \} \right)$
2. Compute the top $k$ right singular vectors of $A$, $V_k$.
3. $(\text{colNorms}, \text{prodNorms}) := \text{rows}.\text{mapSamplingPartitions}(\text{broadcast}(V_k V_k^T)).\text{reduceSamplingPartitions}()$
4. Compute the sampling probabilities $\{p_i\}$ for $1 \leq i \leq n$
5. Obtain $40$ random samples of $c = O(k \log k)$ columns, forming matrices $S_i$ and $D_i$ for all $i$ in $\{1, \ldots, 40\}$.
6. for $1 \leq i \leq 40$
   - Run LAPACK’s DGEQP3 on $V_k^T S_i D_i$ and keep the $k$ first columns, obtaining the column indices to form matrix $C_i$.
7. output $\{C_i \mid i \in [1, 40]\}$

### Norm minimization phase

**Input:** $\{C_i \mid 1 \leq i \leq 40\}$

1. $P := \emptyset$
2. for $1 \leq i \leq 40$
   - $P := P \cup \{C_i^+ A\}$
3. $\text{allMats} := \text{RDD} \left( \{ F_j : (C_i)_j : i \in [1, 40] \} \cup \{ A_j : j \in [1, m] \} \right)$
4. $d := \text{allMats}.\text{mapNormAddends}(\text{broadcast}(P)).\text{reduceNormAddends}()$
5. output $\arg \min_i \|A - P \cdot C_i^+ A\|_F$, i.e. $C_i$ with $i = \arg \min_i d_i$

---

<table>
<thead>
<tr>
<th><strong>mapSamplingPartitions</strong> $(A_i, \text{broadcast}(V_k V_k^T))$</th>
<th><strong>mapNormAddends</strong> $(F_j, \text{broadcast}(P))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $a := A_i \odot A_i$ &lt;br&gt; 2. $b := (A_i V_k V_k^T) \odot (A_i V_k V_k^T)$</td>
<td>1. $D_i := (C_i)_j : C_i^+ A - A_j : \forall (C_i)<em>j : \in F_j \setminus {A_j}$ &lt;br&gt; 2. output vector $d = (D_1 u, \ldots, D</em>{40} u)$, where $u$ is a vector of ones</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>reduceSamplingPartitions</strong> $((a_i, b_i), (a_j, b_j))$</th>
<th><strong>reduceNormAddends</strong> $(d_i, d_j)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $a_{out} := a_i + a_j$ &lt;br&gt; 2. $b_{out} := b_i + b_j$ &lt;br&gt; 3. output $(a_{out}, b_{out})$</td>
<td>1. output $d_i + d_j$</td>
</tr>
</tbody>
</table>
Second, we derive an efficient approach for finding the best candidate subset. A brief description of the algorithm is provided below.

First, we draw a number for the candidate column subsets, where each column \( i \) is sampled with probability

\[
p_i \propto \| (\Sigma V^T)_{(:,i)} \|_2^2
\]

where \( V \) is a matrix whose columns are the right singular vectors of the original data matrix and \( \Sigma \) is a diagonal matrix containing its singular values. Intuitively, high values of this distribution correspond to columns that are well aligned with a singular vector which, because of the orthogonality of \( V \), are almost orthogonal to the space spanned by the rest of the columns. By multiplying by the singular values, we favor the leading singular vectors. Note that it is not necessary to compute all vectors, since the entries of \( V \) that are multiplied by a small singular value vanish and do not contribute noticeably to the final score. In practice, the top-\( r \) values and vectors of a numerically rank-\( r \) matrix will suffice.

Afterwards, we exploit two key aspects pertinent to the nature of the problem of computing residual matrix norms efficiently. First, we leverage the fact that low-rank approximations can be obtained by performing \( n \) linear regressions and therefore they can be obtained via the normal equations, i.e. for candidate subset \( i \)

\[
C_i^+ A = (C_i^T C_i)^{-1} C_i^T A
\]

Second, we take advantage of the fact that different candidate subset matrices tend to share columns in common and so we build a matrix \( Q \) that contains all sampled columns. These two key realizations allow us to compute efficiently all the \( C_i^+ A \) approximating matrices resorting to just one large matrix product that can be done in parallel. More specifically, the distributed computation of the residual matrix norms can be performed as follows:

1. We broadcast the column indices corresponding to each candidate.
2. We broadcast the \( k \times n \) approximating matrices.
3. We do a map operation on each row of the data matrix for obtaining the corresponding row from each candidate subset, multiply it by the corresponding approximating matrix and subtract the low-rank approximation row from the data matrix row.
4. We add up the partially calculated norms via a reduce operation and choose the minimum.

The above outlined procedure is detailed in Figure 11 below. The input parameters are:

- \((A, k, \gamma)\) the data matrix, the number of columns to keep and the number of candidates to sample.
- \(\mathcal{W}(k \log k, 1_n, \pi)\) denotes a Wallenius distribution and \(\text{RRQR}_k((\Sigma V^T)_{(:,i)})\) is a function that performs a rank-revealing QR factorization on the input matrix and returns the first \( k \) indices from the resulting permutation.
Algorithm 1

1: procedure CHOOSECOLUMNS($A, k, \gamma$)  
2: \hspace{1em} rows ← RDD(\{A_{(i,:)}|1 \leq i \leq m\})  
3: \hspace{1em} \textbf{for} $i = 1, \ldots, \gamma$ \textbf{do}  
4: \hspace{2em} $\bar{\Omega}_i \sim \mathcal{W}(k \log k, 1_n, \pi)$  
5: \hspace{2em} $\Omega_i \leftarrow \text{RRQR}_k((\Sigma V^T)(:,:))$  
6: \hspace{1em} \textbf{end for}  
7: \hspace{1em} $Q^T A \leftarrow \text{rows.map}(x \mapsto$  
8: \hspace{2em} $\text{output } x(\Theta)x^T$  
9: \hspace{2em} $\text{reduce}(\langle x, y \rangle \mapsto x + y)$  
10: \hspace{1em} $P \leftarrow \emptyset$  
11: \hspace{1em} \textbf{for} $i = 1, \ldots, \gamma$ \textbf{do}  
12: \hspace{2em} $C_i^T A \leftarrow (Q^T A)_{(\Omega_i,:)}$  
13: \hspace{2em} $C_i^T C_i \leftarrow (Q^T A)_{(\Omega_i',\Omega_i)}$  
14: \hspace{1em} $P \leftarrow P \cup \{(C_i^T C_i)^{-1} C_i^T A\}$  
15: \hspace{1em} \textbf{end for}  
16: \hspace{1em} broadcast($P, T, \Omega$)  
17: \hspace{1em} $\delta \leftarrow \text{rows.map}(x \mapsto$  
18: \hspace{2em} $\text{for } i = 1, \ldots, \gamma \text{ do}$  
19: \hspace{3em} $d_i \leftarrow x - x_{(\Omega_i)}^T C_i^T A$  
20: \hspace{3em} \textbf{end for}$\delta \rightarrow P$  
21: \hspace{1em} \textbf{end procedure}$\delta \leftarrow P$

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure11}
\caption{The PICS algorithm}
\end{figure}

The efficiency and effectiveness of the proposed approach are shown from the experimental results included in section 9.3. Documentation and source code for our implementation can be found at https://gitlab.com/ontic-wp3/pics.

9.2.3 Dealing with the Complexity of SVD

Despite being efficiently implemented in various large-scale machine-learning software packages, such as Apache Spark’s MLlib, SVD has an important drawback: when dealing with tall and skinny matrices i.e. with many more rows than columns, it depends quadratically on the number of features. This might not be considered much of a problem in traditional network traffic analysis, where the number of features usually employed rarely surpasses the order of a few hundred. However, given the advent of programmable networking paradigms such as software defined networking (SDN) and network function virtualization (NFV), network operators would like to analyze detailed traffic datasets comprised of thousands of features. As such, it is becoming desirable to start searching for efficient means to process the wealth of raw-packet or flow-aggregated information in cluster infrastructures for efficiently tuning the programmable network control plane.

To address the above issue, we propose to compute the leverage scores without resorting to SVD, but instead employing a probabilistic PCA (PPCA) approach. The advantage of this approach is that PPCA can be approximated via an Expectation-Maximization (EM) algorithm that iterates in $O(mnk)$ time, where $m$ is the number of rows of the data matrix, $n$ is the number of columns and $k$ is the number of principal components to extract. We provide a parallelized version of the EM algorithm to solve PPCA, PPICS, shown in Figure 12 below.
Algorithm 2 Parallel EM for PPCA

1: function COMPUTEPPCA(rows, k)
2: \[ W \leftarrow \begin{pmatrix} I_k \\ 0 \end{pmatrix}; \sigma^2 \leftarrow 1 \]
3: while not converged do
4: \[ M \leftarrow W^2W + \sigma^2I; \text{broadcast}(M) \]
5: \[ \text{preFactors} \leftarrow \text{rows.map}(x => \langle x_n, x_n^T \rangle) \]
6: \[ \langle x_n \rangle \leftarrow M^{-1}W^T(t_n - \mu) \]
7: \[ \langle x_n x_n^T \rangle \leftarrow \sigma^2M^{-1} + \langle x_n \rangle \langle x_n \rangle^T \]
8: \[ \text{output} \{(\langle x_n \rangle, \langle x_n x_n^T \rangle, x)\} \]
9: \[ (X, Y) \leftarrow \text{preFactors.map}((\langle x_n \rangle, \langle x_n x_n^T \rangle, x) => \langle x_n x_n^T \rangle^T \langle x_n x_n^T \rangle) \]
10: \[ \tilde{W} \leftarrow XY^{-1} \]
11: \[ \tilde{\sigma}^2 \leftarrow \text{preFactors.map}((\langle x_n \rangle, \langle x_n x_n^T \rangle, x) => \frac{1}{N} \sum_{n=1}^{N} \left( ||t_n - \mu||^2 \right) \]
12: \[ + \text{tr}(\langle x_n x_n^T \rangle \tilde{W}^T \tilde{W}) \]
13: \[ W \leftarrow \tilde{W}; \sigma^2 \leftarrow \tilde{\sigma}^2 \]
14: end while
15: \[ U, \Sigma, V \leftarrow \text{svd}(W^T W) \]
16: \[ Q, R \leftarrow \text{qr}(WV^T) \]
17: \[ \lambda \leftarrow (VW^T VW^T + \sigma^2)^{1/2} \]
18: \[ \text{output} \{(Q, \lambda)\} \]
19: end function

Figure 12: Parallelized EM algorithm for PPCA

Initial experimental results (see section 9.3) show the potential of this approach. Documentation and source code for our implementation can be found at https://gitlab.com/ontic-wp3/ppics.

9.2.4 Fast, Deterministic Feature Selection

Despite performing well in practice, the algorithms described so far are randomized and they depend on the number of drawn candidates. As a consequence, the chance of failure can vary significantly. To overcome this drawback, we propose Parallel Ranking and Selection (PRANKS), an efficient, deterministic algorithm for providing feature ranking in Big Data-sets.

The motivation behind the proposed algorithm lies predominantly in its robustness. In practical settings it is desirable to have an algorithm that performs deterministically, that is insensitive to free parameters and that is easy to implement and use. In addition, the very nature of the algorithm allows for the choice of the number of features to keep, to be made a posteriori in a straightforward manner, even more so than with PICS.

The algorithm is based on the notion of Rank-revealing QR (RRQR) factorization. This factorization tries to reveal numerically independent sets of columns by permuting them to the front in order to produce a small lower right block in the matrix R of the QR factorization. Some existing RRQR factorizations are known to provide good theoretical bounds for the CSSP [42].
However, and to the best of our knowledge, there exist no Big Data implementations in distributed environments.

PRAKS enables us to exploit the benefits of RRQR factorizations when dealing with matrices where the number of rows is much greater than the number of columns. The algorithm can be summarized as shown in Figure 13.

\[ A \rightarrow U \Sigma V^T \]

The rationale behind the algorithm is as follows: since \( \Sigma V^T = U^T A \) holds, the matrix on which RRQR is performed preserves the pair-wise distances between the columns of \( A \), note that the columns of \( U \) are orthonormal and span the column space of \( A \). The parallel implementation of the algorithm relies on DGEQP3, one of the most popular implementations of an RRQR, which functions in a way such that the column permutation produced on \( \Sigma V^T \) is equivalent (up to numerical precision) to the one obtained by running it on the original data matrix. Therefore, the PRANKS algorithm can yield a feature ranking for any tall and skinny matrix by means of an efficient, local operation, regardless of its number of rows. Documentation and source code for our implementation can be found at \( \text{https://gitlab.com/ontic-wp3/pranks} \).

9.3 Experiments and Results

This section presents experimental results with the algorithms described in the previous section.

The experiments were run on a cluster of 10 worker nodes equipped with an Intel quad-core processor and 4 GB of RAM each, connected through an ethernet switch with a capacity of 100 Mbps on each link. The HDFS distributed file system implemented on Hadoop 2.6 was employed for data storage and access. All algorithms were implemented using Scala 2.10.4 on Spark 1.4.1, except for PPCSS, which was implemented on Java 7.

All experiments were run with 2 GB of RAM allocated to each worker (the remaining amount from what the OS, Spark and Hadoop consume). The memory fraction devoted to RDD caching was left to its default value unless otherwise stated. The value of minPartitions (i.e. the requested amount of chunks to divide the file into) was set to the number of available cores, except for PICS, where we observed an improvement in performance by setting this parameter to twice that value.

All algorithms were tested on samples of the ONTS dataset, which were aggregated into 5-tuple flows using TSTAT. The resulting matrices range from 50,000 to 27 million rows, in files of up to 9.2 GB. The number of features is in some cases reached 105 (for the data aggregated with TSTAT 2.0) and 95 in others (for the data aggregated with TSTAT 3.0).

9.3.1 PPCSS

In the case of PPCSS, since its memory requirements are high, we had to reduce the memory fraction to be used for Spark cache. We found that a value of 0.4, (i.e. 40% of the Java heap size) provided the best stability. We ran two experiments. The purpose of the first one was to test the scalability of the algorithm with respect to the number of workers. We launched the algorithm on the full 2,500,000 x 105 matrix using 1, 2, 5, and all 10 nodes. As shown in Figure 14, PPCSS benefits from the increase in the number of workers. As expected, the speedup
decreases as the number of workers grows while, as it can be observed, a point of asymptotic lack of improvement will be eventually reached. However, this point is expected to shift to the right as the size of the input data-set grows.

In the second set of experiments, we executed the algorithm on datasets of different sizes. We trimmed the original data to obtain sets of 50,000, 100,000, 200,000, 400,000, 800,000 and 1.6 million instances, and tested the algorithm on them. The results, shown in Figure 15, confirm that the algorithm scales linearly with respect to the number of rows in the input matrix. When the size of the dataset is small, however, cluster management time (which is more or less constant for a fixed-size cluster) and the overhead of I/O operations are significant with respect to the total time.

![Figure 14: PPCSS with different number of workers](image1)

![Figure 15: PPCSS with different dataset sizes](image2)

While showing good behavior in terms of Big Data processing - linear scalability and good speedup - the running times and the memory requirements of PPCSS are still high. To address these issues, we developed PICS.

### 9.3.2 PICS

In order to validate the performance of PICS we ran it on 30 different datasets corresponding to the TSTAT-aggregated traffic ranging from April 7th to May 6th 2015. Each day of the month consists of a 4-10GB file containing between 12 and 27 million 5-tuple TCP flows, totaling 534 million data samples expressed in 95 features. The total volume of all files was of 183GB.

We ran the algorithm for each day separately (obtaining 30 feature subsets) and evaluated the results. We set k=10 to obtain a manageable number of features for a posteriori domain expert analysis.

To evaluate the quality of the chosen feature subsets, we measure the ratio of the residual obtained with our algorithm to that of the best rank-k approximation. Figure 16 shows the described ratio for each day for three subsets of the full feature set: the one chosen by the algorithm for that day (algorithm_ratio), the 10 features that appear the most often among the daily chosen subset throughout the month (best_app) and the top-10 features according to their accumulated leverage scores (best_scores). The leverage score of a feature is the probability of choosing it according to the probability distribution described above, and its accumulated value is the sum of the scores over the whole month. The value of algorithm_ratio remains consistently close to 1.05, revealing first, that the analyzed data contains feature subsets that can approximate almost the full dataset as well as the top-k singular vectors, and second, that PICS can indeed find such a feature subset. The ratio for best_app is slightly higher, but
moderate throughout the month, suggesting that there exists a single subset of features that can consistently provide good approximation errors.

In order to assess the usefulness of our approach in the domain of network traffic analysis, we examine the chosen features and their leverage scores. If our proposed distribution is a good sampling strategy, then features with high scores should tend to provide low approximation errors. We collect the top-10 features that appear the most often among the chosen subset throughout the month (Table 5), where the prefixes c and s indicate client-to-server and server-to-client, respectively. The leverage scores of these features are shown in Figure 17. The fact that these features have consistently high leverage scores (the highest observed leverage scores overall are below 0.18) suggests that the proposed sampling strategy is a good heuristic for randomly sampling candidate features. It is interesting to point out the relationship between the obtained features and those most frequently present in the literature. Some of them (durat, c_cwin_ini) are consistent with choices often made when applying machine-learning techniques to network traffic analysis, usually relying on expert criteria or supervised methods. Others, however (and to the best of our knowledge) had not been previously identified as being especially relevant or informative. This is the case of s_ttl_max, c_ttl_max, c_f1323_opt, c_pkts_retx, s_ack_cnt_p and s_bytes_retx. Finally, we did not detect the explicit presence of c_rtt_max or s_rtt_avg as reported in the literature, although related metrics such as inter-packet arrival times do appear. Therefore, it would be interesting to study the linear dependencies between these sets of features and determine which one of them tends to approximate data best.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Days present</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_rtt_max</td>
<td>Maximum RTT</td>
<td>19</td>
</tr>
<tr>
<td>s_ttl_max</td>
<td>Maximum Time To Live</td>
<td>18</td>
</tr>
<tr>
<td>s_rtt_avg</td>
<td>Average RTT</td>
<td>16</td>
</tr>
<tr>
<td>durat</td>
<td>Flow duration</td>
<td>15</td>
</tr>
<tr>
<td>c_f1323_opt</td>
<td>Window scale option sent (boolean)</td>
<td>14</td>
</tr>
<tr>
<td>c_pkts_retx</td>
<td>Number of retransmitted segments</td>
<td>12</td>
</tr>
<tr>
<td>c_cwin_ini</td>
<td>First in-flight size</td>
<td>11</td>
</tr>
<tr>
<td>s_ack_cnt_p</td>
<td>Segments with ACK=1 and no data</td>
<td>11</td>
</tr>
<tr>
<td>c_ttl_max</td>
<td>Maximum Time To Live</td>
<td>11</td>
</tr>
<tr>
<td>s_bytes_retx</td>
<td>Number of retransmitted bytes</td>
<td>10</td>
</tr>
</tbody>
</table>

Finally, Figure 18 and Figure 19 show the running times of PICS with respect to the number of workers in the cluster (with 20 million rows) and the size of the dataset (in millions of rows). As
it can be seen, it exhibits significant gains from parallelization and roughly linear scalability with respect to the number of data samples.

![Figure 18: PICS with different numbers of workers](image)

![Figure 19: PICS with different data set sizes](image)

The above results confirm that PICS significantly outperforms PPCSS in terms of scalability.

### 9.4 Conclusions and Future Work

We have presented four different approaches for unsupervised feature selection on large-scale datasets. The experimental results presented for two of them show linear scalability with respect to the size of the data and significant speedup gained from parallelization. PICS clearly outperforms PPCSS with respect to running time and horizontal scalability while it has proved able to ingest large datasets in short times in a moderate cluster. Therefore, it constitutes a valid approach for unsupervised feature selection on Big Data.

With respect to comparing PPICS and PRANKS, we have only performed a small number of preliminary experiments. However, the residual matrix norms obtained from both seem to be of a very good quality. The efficiency of PRANKS, as expected, is much superior to any of the other algorithms presented here.

We plan experimentation in the future for covering the following aspects. First of all, we will perform thorough experiments with PPICS and PRANKS to evaluate their benefits. In the case of PPICS, experiments on a more powerful cluster will allow us to confirm the extent to which the impact of high dimensionality is an obstacle for using the SVD. In the case of PRANKS, its efficiency is well-proved in cases of tall and skinny matrices with a relatively moderate number of columns (several hundred). In order to confer PRANKS with the ability to handle extremely high-dimensional datasets, it would be interesting to explore the possibility of developing scalable versions of rank-revealing QR factorizations.
ANNEX A - TM Forum Big Data Analysis Reference Architecture

Figure 20: TM Forum Big Data Analytics Reference Architecture
(Source: Copyright © TM Forum 2014. All Rights Reserved.)
10. REFERENCES


[27] Sandy Ryza, Uri Laserson, Sean Owen, Josh Wills, Advanced Analytics with SPARK, O'Reilly, first edition, April 2015.


[35] Sun, Zhanquan, and Zhao Li, Data intensive parallel feature selection method study, Neural Networks (IJCNN), 2014 International Joint Conference on. IEEE, 2014.


[38] He, Qing, et al, Parallel feature selection using positive approximation based on MapReduce, Fuzzy Systems and Knowledge Discovery (FSKD), 2014 11th International Conference on. IEEE, 2014.


