FP7 Project ASAP
Adaptable Scalable Analytics Platform

ASAP D7.3

ASAP System Prototype

WP 7 – Integration of the ASAP System

Nature: Report

Dissemination: Public

Version History

Version	Date	Author	Comments
0.1	03 Feb 2017	P. Pratikakis, S. Papagiannaki, M. Chalkiadaki	Initial Version
0.2	23 Feb 2017	S. Papagiannaki, M. Chalkiadaki	First Revision
0.3	28 Feb 2017	P. Pratikakis	Second Revision, review by K. Doka

Acknowledgement This project has received funding from the European Union's 7th Framework Programme for research, technological development and demonstration under grant agreement number 619706.

Executive Summary

This deliverable describes work done in WP7 during the third year of the ASAP project. Work involves integrating features added to all modules during the second and third years of the project, development of test cases, debugging, performance debugging, development and integration of documentation for all modules and the system as a whole, and evaluation on the ASAP system installation and operation on the actual applications, in three different set-up (both virtual and silicon) cluster environments.

Contents

1 **Introduction** 4
- 1.1 Task Description . . . 4
- 1.2 Overview of Integrated System . . . 4

2 **ASAP System Components** 6
- 2.1 Intelligent, Multi-Engine Resource Scheduler (IReS) . . . 6
- 2.2 Workflow Management Tool (WMT) . . . 7
- 2.3 Asap Operators . . . 7
- 2.4 External Engines . . . 7
- 2.5 ASAP Engines . . . 7
- 2.6 Visualization Component . . . 8

3 **ASAP Integration Summary** 9

4 **Prototype Setup** 11
- 4.1 ASAP source code . . . 11
- 4.2 Unified Setup using Fabric . . . 11

5 **Integration of System Components** 13
- 5.1 Integration of WMT and IReS . . . 13
  - 5.1.1 Modifications in IReS . . . 13
- 5.2 Integration of IReS and Yarn . . . 19
- 5.3 Integration of IReS and Analytics Engines . . . 20
- 5.4 Integration of IReS with WIND Application . . . 23
- 5.5 Integration of IReS and Swan . . . 27
- 5.6 Integration of Visualization . . . 27

6 **Integrated Prototype** 30
- 6.1 Cluster Deployment . . . 30
- 6.2 VM Deployment . . . 30

7 **Evaluation** 33

1 Introduction

1.1 Task Description

This deliverable describes work performed within task *T7.2 Integration Prototype "ASAP System Prototype"*, and task *T7.3: Testing of the ASAP platform*. The tasks integrate modules developed in the other work packages, develop testing and evaluation for the modules, assist with deployment in the two application use cases, and include evaluation of the integrated system and individual modules, on usability, performance, throughput, and scalability.

Task T7.2 integrates the technologies of the components from WP2, WP3, WP4, WP5 and WP6, based on the overall architecture defined in D1.2, in order to execute a subset of the use cases proposed in WP8 and WP9. The task produced a Virtual Machine image set that integrates all ASAP modules as well as off-the-shelf analytics engines and components, and facilitates deployment and use.

Task T7.3 tests the integrated platform and the individual components. It gives continuous feedback to the application development and drives modifications to the integrated tool, as well as to the individual components and services. During task T7.2 we tested the functionality of the simple and more complex use cases developed in WP1 to verify the correctness of the integrated system. We also tested the queries developed on the two applications using simple or reduced, heterogeneous data stores. All of these tests are automated. We test and verify the correct function of the integrated platform and evaluate it by deploying it on three different environments, provided by WIND, IMR and FORTH.

1.2 Overview of Integrated System

The ASAP project focuses on (i) innovative methods and technologies and (ii) tools and applications. Regarding methods and technology, we develop novel methods in order to model cost and performance of multiple data stores and analytics execution engines. Building on these, we perform automated job scheduling to multiple runtime and data store technologies together with real-time tracking of intermediate results. To deliver this technology to the end user, we couple it with state-of-the-art visualization tools enabling both qualitative and quantitative monitoring of a job's performance and cost. The integrated technology enables fast, easy development and submission of both simple and highly complex analytics tasks that take full advantage of the existing resources according to user requirements. Overall, ASAP delivers open source tools that can be used both separately and as an integrated system in order to provide efficient execution and management of complex analytics tasks. This deliverable reports on the work for the integration of all components into an integrated system.

Specifically, the main objectives of work package WP7 are:

- Ensure the integration of the contributions in WP2, WP3, WP4, WP5 and WP6.

- Coordinate development and delivery of the integrated modules based on the research and development results in the different Research Areas.
- Ensure that these prototypes are used to integrate and coordinate the coherent delivery of the ASAP system for its application in WP8 and WP9.
- Evaluate the end system.

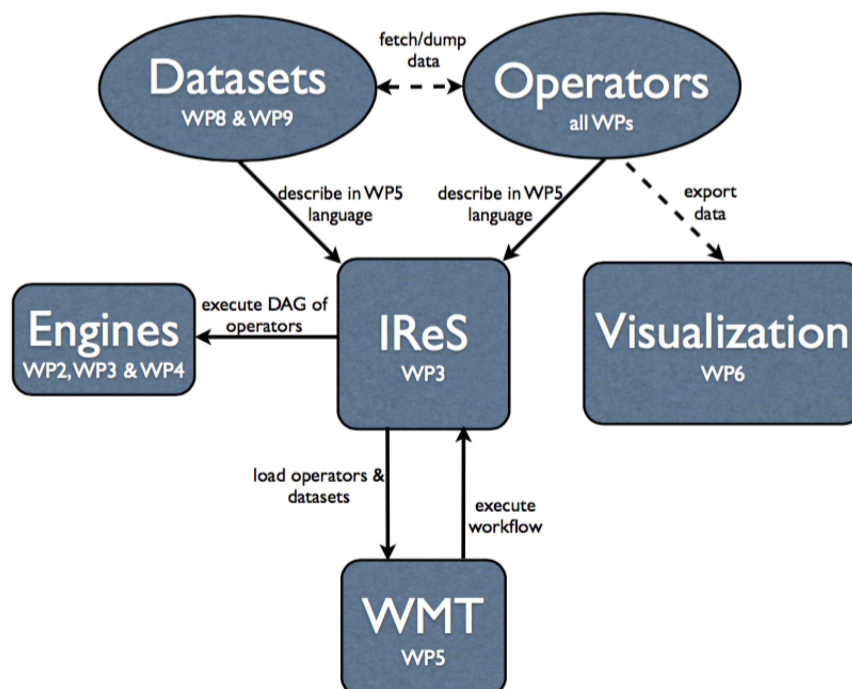


Figure 1: ASAP components

2 ASAP System Components

This section briefly describes the separate components of the ASAP system, depicted along with their interactions, in Figure 1. Readers already familiar with every module, as reported in detail in the corresponding deliverables, can skip this section.

2.1 Intelligent, Multi-Engine Resource Scheduler (IReS)

The IReS platform, thoroughly described and implemented in the scope of WP3, targets the workflow optimization, examining alternative execution paths using various underlying engine and operator implementations. Using its web interface the user can define operators and datasets along with their properties and restrictions and store them in a language understandable by the other ASAP components and specified in WP5. Furthermore, it provides functionality for validating and executing workflows by extending the Apache Kitten7 framework [2] in order to execute over YARN [1], apart from separate operators, also workflows as a DAG of operators.

IReS is an open source[1] web application that exposes its functionality to the rest of the ASAP components through a RESTful API. It is implemented in Java using the Jetty [11] servlet engine and the Jersey [10] RESTful Web service framework.

[1] https://github.com/project-asap/IReS-Platform

2.2 Workflow Management Tool (WMT)

The WMT, described and implemented in the scope of WP5, provides full functionality for designing, editing, analyzing and optimizing an abstract workflow. It interacts with IReS for loading the registered operators and executing workflows via the RESTful API the latter provides. Its Analysis and Optimization functionality can be invoked by web actions that call the respective Python methods.

The WMT is an open source[2] Javascript [5] Web application rendered behind an Nginx [7] web server.

2.3 Asap Operators

In the scope of WP3 a number of popular analytics operations (TF/IDF, K-Means, Word2Vec etc) are modeled and profiled in several runtimes (Hadoop [9], Weka [15], Mahout [6]). In addition to this for the needs of the ASAP the following additional operators have been implemented, registered and profiled by IReS:

- K-Means, TF/IDF and Word2Vec implementations for Swan, developed in the scope of WP2.
- Web analytics operation implementations, developed in the scope of WP8.
- Peak detection and Sociometer implementations in Spark and Spark-Nested, developed in the scope of WP9.
- Operators for dumping data in the visualization dashboard, developed in the scope of WP9.

2.4 External Engines

For the execution of the operators listed in the previous section, ASAP uses Hadoop [9] and more precilly HDFS [13], Mahout [6], Weka [15] and Spark ML-Lib [12] running on Spark [16].

2.5 ASAP Engines

In addition, we use Swan [14], an experimental extension of Cilk [8] for operators written to use a data-flow style of execution. Finally, we use the Spark-Nested framework developed in WP4 for support of Spark applications that require nested transformations, hierarchical data representation and distributed scheduling.

[2] https://github.com/project-asap/workflow

2.6 Visualization Component

The visualization component, subject of the WP6, consists of:

- The ASAP dashboard which collects, queries and visualizes data. The dashboard is implemented using the D3 JavaScript library and it is hosted on a webLyzard server.
- Open APIs for ingesting data in an Elasticsearch [3] installation. The Document API is used for ingesting crawled Web documents (unstructured data) from WP8. The Statistical Data API is used for ingesting the telecommunications data (structured data) produced by WP9.

3 ASAP Integration Summary

Intuitively, the life-cycle execution of a data analytics job in the scope of ASAP is summarised in the following steps:

1. The Developer designs a primitive computation as an abstractor operator using the IReS web interface.
2. The operator metadata, describing its semantics in the workflow description language developed in WP5, are stored in the ASAP operator library that resides in the IReS.
3. The Developer adds a number of data sources using the IReS web interface.
4. The data source metadata, describing its location in the workflow language developed in WP5, are stored in the ASAP operator library that resides in the IReS.
5. The Developer creates one or multiple implementations of the above operator, following the programming model proposed in WP2.
6. The Developer using the IReS web interface, stores the above implementations in the ASAP operator library as materialized operators.
7. The Developer updates the metadata describing the materialized operator's semantics in the workflow description language developed in WP5 by introducing scripts for its execution automation.
8. The IReS profiler builds a cost model of the operator implementations and saves them along with the materialized operator's metadata.
9. The WMT, during the initialization process, loads the ASAP library by making the respective request to the RESTful API of the IReS platformm.
10. The Workflow Designer, using the WMT web interface, designs a workflow by combining the available data sources and operators.
11. The Workflow Designer, using the WMT web interface, can analyze and optimize the workflow.
12. The Workflow Desinger, can save the workflow in the workflow language developed in WP5.
13. The User, using the WMT web interface, can load a existing workflow and initiate the execution of the computation by making the respective request to the API of the IReS platform.
14. The IReS platform schedules the workflow using the best possible execution plan, based on the operator metadata and costs.

15. The IReS platform orchestrates the execution of the selected execution plan by employing YARN for integrating with the various computing engines that lay underneath (e.g. Swan or Spark Nested documented in WP4).
16. An operator can dump results to the Elasticsearch installation of the Visualization component using the RESTful API introduced in WP6.
17. The User can see the intermediate or final computation results using the Visualization Dashboard described in WP6.

Repository	Description	Work Package	Collaborators
asap_operators forked from hvdieren/asap_operators	Swan Analytics Operators	WP2	hvdieren (QUB), Murphky(QUB)
IReS-Platform	IReS	WP3	npapa (ICCS), gsvic (ICCS), vpapaioannou (ICCS), cmantas (ICCS), kdoka (ICCS), polyvios (FORTH)
Spark-Nested	Spark Nested	WP4	polyvios (FORTH), papagian (FORTH), mhalkiad (FORTH), p01K (FORTH)
workflow	WMT	WP5	maxfil (GENEVE)
weblyzard_api forked from weblyzard's repository	Visualization Web Services	WP6	
fabric-scripts	Fabric scripts	WP7	papagian (FORTH)
web-analytics	Web analytics application	WP8	thanh-im (IMR), rigaux (IMR)
telecom-analytics	Telecom analytics application	WP9	papagian (FORTH), mhalkiad (FORTH), mesosbrodleto (UNIPI)

Table 1: Github repositories

4 Prototype Setup

This section describes the final status of the integration. Spcecifically, we describe the organization of code in repositories and provide brief directions on how one can deploy the corresponding modules and the system as a whole. For more detailed and up-to-date user guide and installation instructions, we refer the reader to the documentation within the repositories described below. The documentation of all modules has also been integrated and is available as a whole at the project website.

4.1 ASAP source code

The majority of the ASAP components are open source. We created a separate repository for each component, unified under a common project account in Github, or it has been forked from a repository residing under another account (usually the main contributor). Table 1 shows the respective Github repositories.

4.2 Unified Setup using Fabric

To simplify the installation of all the components and their continuous and smooth integration we have employed Fabric [4]: a Python library and command-line tool for streamlining the use of SSH for application deployment or systems administration

tasks. Fabric tasks are typically methods that execute shell commands easily and handle failures nicely.

Deliverable D7.2 describes the work done during the second year of the project regarding continuous integration of installation and deployment. During the final year of the project, we have augmented the work done in the second year with additional use cases and tests, and also integrated the additional operators developed as alternative implementations within WP8 and WP9. Specifically, we developed alternative implementations of the Sociometer and TF/IDF operators in Spark-Nesting and Swan (described in detail in the corresponding Deliverables). These operators were integrated with WMT, IReS, and the execution engines (Yarn, Spark-Nesting, Python/PySpark) so that the ASAP applications partners can easily, or sometimes completely transparently, integrate them in their workflows. This mainly amounted to metadata description files that integrate each new operator with the available tools, together with the development of "formatting" operators that need to transform data formats to the formats accepted by each operator's alternative implementations.

5 Integration of System Components

During the third year of the project, a lot of effort was spent to develop, test and evaluate integration functionality between the individual ASAP modules. This section presents the work done in FORTH for each pair of modules that interoperate in ASAP, during the third year of the project.

5.1 Integration of WMT and IReS

The Workflow Management Tool (WMT) was modified in order to contact the IReS external API and perform the following operations:

- load the defined datasets,
- load the defined abstract operators,
- design an abstract workflow using the above dataset and operators
- upload an abstract workflow in IReS so as to be ready to be materialized and executed
- execute a workflow directly from the WMT

In order to accomplice the above tasks we performed a number of modifications in both components. The following sections describe the modifications in each component separately.

5.1.1 Modifications in IReS

Support for Cross-Origin requests User agents commonly apply same-origin restrictions to network requests. These restrictions prevent a client-side Web application running from one origin from obtaining data retrieved from another origin, and also limit unsafe HTTP requests that can be automatically launched toward destinations that differ from the running application's origin.

WMT and IReS are both web applications. So initially any request to IReS originated from WMT used to fail with Access Denied. Therefore IReS had to be equipped it with CORS support according to Cross Origin Resource Sharing specification.[3] IReS has incorporated Jersey[4] for its Web Services, therefore we selected jersey-cors-filter[5] library for doing that. Briefly, this library provides a filter that wraps an HTTP request and adds the some necessary headers to the HTTP response according to the above specification. So, we modified IReS adding a dependency on that library and registering the filter.

[3] https://www.w3.org/TR/cors/
[4] https://jersey.java.net/
[5] https://github.com/palominolabs/jersey-cors-filter

HTTP Method	Path	Java Method	Description
GET	/abstractOperators/{id}/	AbstractOperators.getApplicationInfo()	Get abstract operator description in JSON
GET	/abstractWorkflows/id/	AbstractWorkflows.getDescription()	Get abstract workflow description in JSON
POST	/abstractWorkflows/add/id/	AbstractWorkflows.addWorkflow()	Upload an abstract workflow
GET	/datasets/json/id/	Datasets.getOperatorInfoJson()	Get dataset description in JSON

Table 2: IReS and WMT API modifications

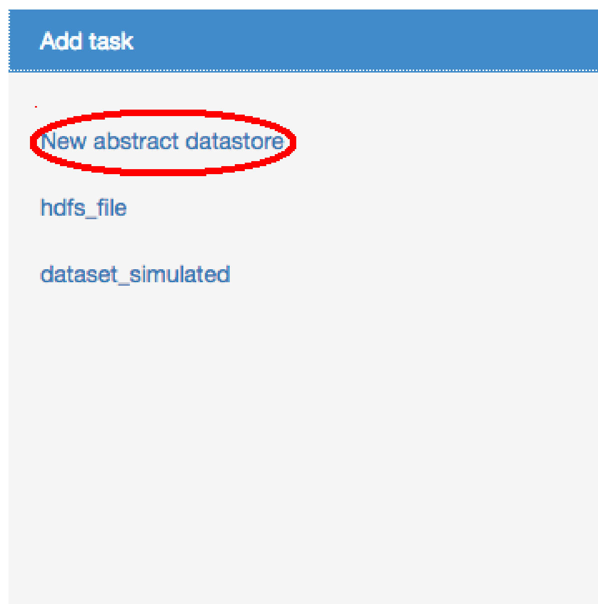
API annotation for supporting Cross-Origin requests Once the filter was registered, the necessary resource methods (@GET, @POST, etc.) had to be annotated with @Cors in order to send basic resource response headers and the respective @OPTIONS methods with @CorsPreflight to send preflight request response headers. Table 2 summarizes the parts of the API that were modified.

REST API extensions We extended the external API of IReS in order to provide some extra functionality. Specifically, the initial API calls for listing of the existing datasets and abstract operators return HTML code in text format that is not convenient to be parsed. Therefore, we introduced two additional API calls that return the list of the datasets and abstract operators respectively in JSON format. These methods are also annotated with @Cors in order to send the proper resource response headers. Moreover, integration lead to several issues being found and corrected in IReS, on the workflow materialization and execution via the REST API. These modifications are communicated to WP3 by being uploaded in the branch `project-asap-patch-3`[6] of the IReS-Platform repository.

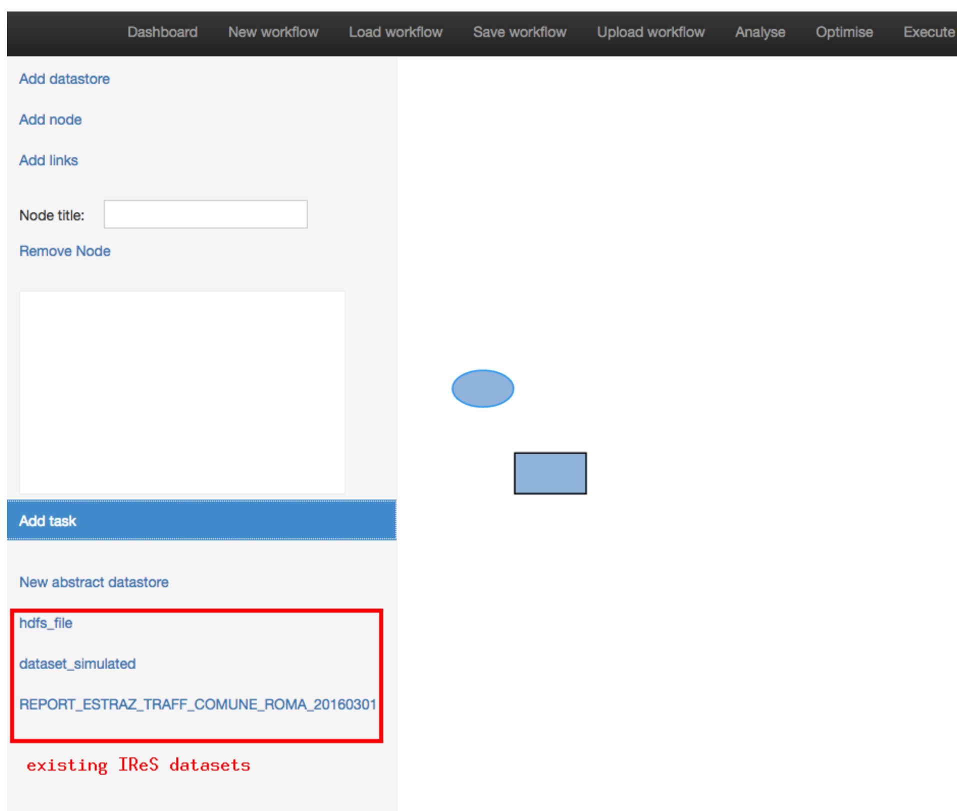
Modifications in WMT We extended WMT by introducing the following functionality:

- We added a new link "New abstract datastore" that is visible in the task board when a dataset node is select in the graph and triggers an action for adding a description in the selected dataset similar as the existing "Create new" link that used to do the same for an operator:

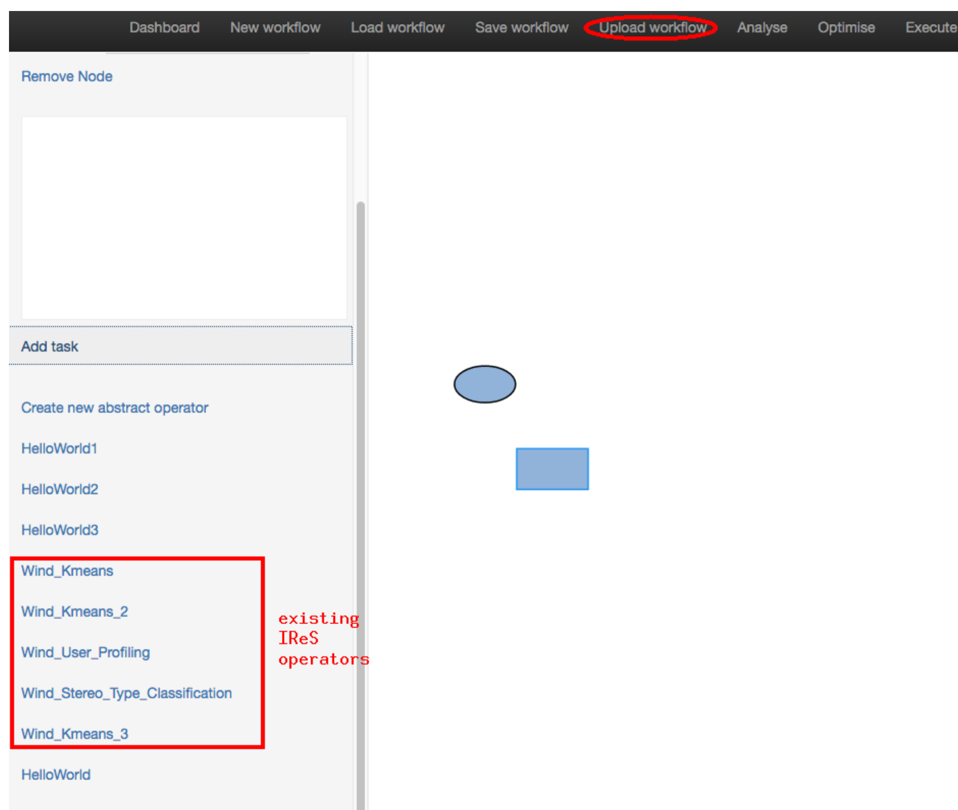
[6] `https://github.com/project-asap/IReS-Platform/tree/project-asap-patch-3`



- We added functionality to request the list of the existing datasets in IReS and place the results in the task board under the above "New abstract datastore" link that permits the user to select among the existing datasets instead of creating a new one:



- We added functionality for WMT to request the list of the existing abstract operators in IReS and place the results in the task board under the above "Create new" link that permits the user to select among the existing abstract operators instead of creating a new description:



- We added a new "Upload workflow" link in the navigation bar that uploads the graph in IReS that is visible also in the above image.

These modifications were communicated to WP5 by being uploaded in the branch `integration`[7] of the WMT repository.

Workflow creation in WMT and uploading in IReS To demonstrate the integration of WMT with IReS, we designed the WIND sociometer use case in the WMT and uploaded it in IReS. We extended integration tests to include the following scenario.

Assuming that the description of the three abstract operators assembling the use case (`Wind_User_Profiling`, `Wind_Kmeans`, `Wind_Stereo_Type_Classification`) and the input dataset (`dataset_simulated`) are defined in IReS as described in the ASAP documentation[8], the steps are the following:

[7] `https://github.com/project-asap/workflow/tree/integration`
[8] `https://project-asap.github.io/ASAP-documentation/ires_docs/install.html#creating-abstract-operators`

1. Using a browser, navigate to the WMT Web UI.
2. From the navigation bar, click “New workflow” and provide a name, *e.g.,* sociometer, and press OK; the workflow board will be emptied.
3. In the left sidebar click “Add datastore”; an ellipse node will appear in the workflow board.
4. Click in the ellipse node; the taskboard will become visible in the sidebar.
5. Click “Add Task” in the sidebar; the link “New abstract datastore” as well as a list with links for the existing datastores will appear below.
6. Select `dataset_simulated` among the existing datastores; an ellipse node will appear in the taskboard above. In addition to this the ellipse node in the workflow board will get the datastore name (`dataset_simulated`).
7. Click in the ellipse in the taskboard: the datastore metadata (e.g. engine, path) will appear in the text area below.
8. In the left sidebar click “Add node”; a rectangular node will appear in the workflow board.
9. Click in the rectangular node.
10. . Click “Add Task” in the sidebar; the link “Create new abstract operator” as well as a list with links for the existing abstract operators will appear below.
11. . Select the `Wind_User_Profiling` among the existing datastores; a rectangular node will appear in the taskboard above. In addition to this the rectangular node in the workflow board will get the operator name (`Wind_User_Profiling`).
12. . Click in the rectangular in the taskboard: the operator metadata (*e.g.,* number of inputs, number of outputs, algorithm) will appear in the text area below.
13. In the left sidebar click “Add links”; select first the dataset_simulated datastore in the workflow board and then the `Wind_User_Profiling` operator in the workflow board; an arrow from the datastore_simulated to the `Wind_User_Profiling` will appear in the workflow board.
14. In the left sidebar click “Add datastore”; an ellipse node will appear in the workflow board.
15. Click in the ellipse node.
16. Click “Add Task” in the sidebar and click “New abstract datastore” this time. Provide a name *e.g.,* `d1` in the prompt and press OK; If you click in the ellipse in the taskboard: empty metadata will appear in the text area below because it is an abstract datastore.

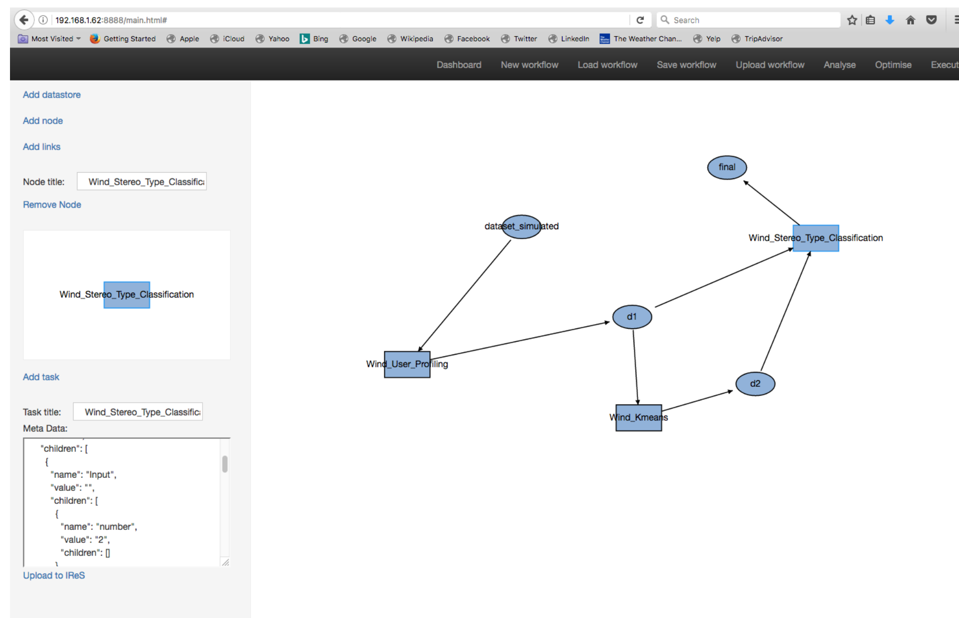


Figure 2: Example Workflow test case

17. In the left sidebar click "Add links"; select first the `Wind_User_Profing` operator in the workflow board and then the `d1` dataset in the workflow board; an arrow from the `Wind_User_Profiling` to the `d1` datastore will appear in the workflow board.

18. Repeat steps 8-17 for the two successive operators (`Wind_Kmeans`, `Wind_Stereo_Type_Classifi` in order to complete the workflow. The resulting graph appears in Figure 2. Notice also that the `Wind_Stereo_Type_Classification` has 2 inputs, the output of the `Wind_User_Profiling` operator saved temporarily in d1 datastore and the output or the `Wind_Kmeans` operator saved temporarily in `d2` datastore, therefore there are two arrows from `d1` and `d2` destinated to this operator.

19. Click "Upload workflow" in the navigation bar; the workflow will be send to the IReS and will appear in the tab of the Abstract Workflows of the IReS Web UI.

20. Follow the specific link and you will be navigated in the abstract workflow view, depicted in the Figure 3. If the respective materialized operators exist the workflow can be materialized and executed as any other abstract workflow designed in IReS as it is described in the ASAP documentation[9].

[9] `https://project-asap.github.io/ASAP-documentation/ires_docs/install.html#workflow-materialization`

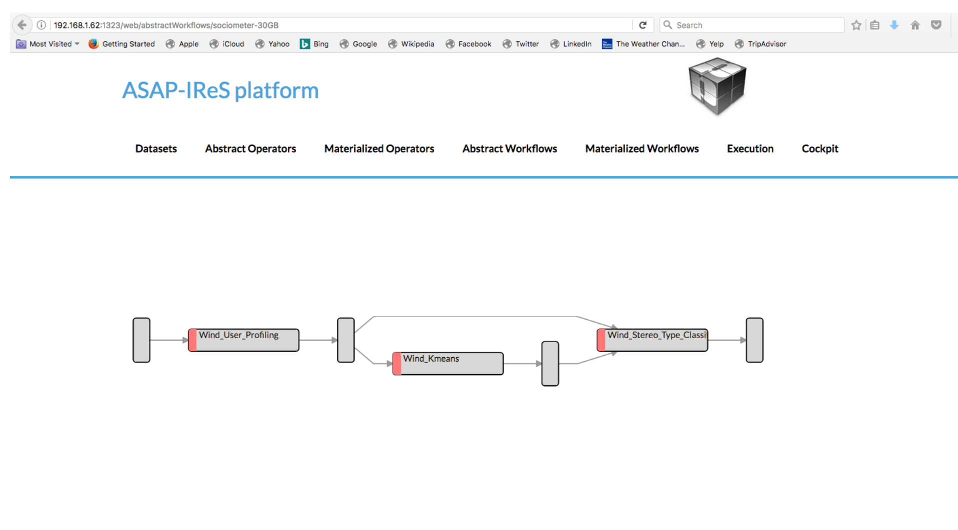


Figure 3: Abstract Workflow View

5.2 Integration of IReS and Yarn

The IReS enforcer module is responsible for the execution of the generated optimal workflow execution plan. It enforces the execution of each operator of the workflow over the physical infrastructure. The enforcer module is built on top of Yarn resource scheduler.

Integration with Yarn In order to be able to use Yarn as the underlying workflow executor and scheduler, the enforcer module extends Cloudera Kitten, a framework for the easy deployment of distributed applications running over Yarn. In our case, we use Kitten to define the execution parameters of the operators, which can be either distributed or centralized.

Container Allocation After the optimal plan generation by the multi-engine planner the output workflow plan is sent to the enforcer module. Then, one Yarn container is allocated as a master/resource manager node and one container for the execution of each operator in the workflow. The execution of each operator takes place inside the allocated container. After the successful operator execution the results are written at the operator's directory in HDFS and the container is being destroyed.

Configuration File Along with each operator definition, the corresponding `.lua` file that describes how the operator will be executed is required. Specifically, the `.lua` file contains information about the container resources (cores, memory), the execution command, as well as the several possible files required by the operator to run. Figure 4

```
1  operator = yarn {
2    name = "LineCount",
3    timeout = 10000,
4    memory = 1024,
5    cores = 1,
6    container = {
7      instances = 1,
8      --env = base_env,
9      resources = {
10       ["count_lines.sh"] = {
11         file = "asapLibrary/operators/LineCount/count_lines.sh",
12                type = "file",              -- other value:  'archive'
13                visibility = "application"  -- other values: 'private', 'public'
14       }
15     },
16     command = {
17       base = "./count_lines.sh"
18     }
19   }
20 }
```

Figure 4: Example operator description file in LUA

shows an example description for an operator that counts the lines of an input text file using a bash script.

This file defines that this operator with name “LineCount” will be run inside a container with 1024 MB of memory (line 4) and 1 core (line 5). The required file(resource, line 5) is the “count_lines.sh” script while the command that will be executed inside the Yarn operator is “./count_lines.sh” (line 17). All files included in the resources key will be copied to the container before execution.

For the integration between IReS and Yarn, the “BasicLuaConf.lua” file inside the `asapLibrary` directory needs to be properly set. Figure 5 shows a sample file that describes the location of the Kitten `.jar` file (line 1) that the master node requires in order to schedule the execution, the Yarn Classpath (line 3) as long as allocation of the master node (operator key, line 18).

5.3 Integration of IReS and Analytics Engines

IReS follows an engine-agnostic approach for the integration with the underlying systems, allowing the easy addition of new engines and operators by the users.

Definition of Operator Engine To achieve easy integration with new engines, IReS adopts an extensible metadata-framework which describes in a common way an operator using a set of mutual description parameters. These parameters are represented with the Tree-Metadata Framework. Concretely, an operator can be characterized as

```
1  MASTER_JAR_LOCATION = "kitten-master-0.2.0-jar-with-dependencies.jar"
2
3  CP = "/opt/hadoop-2.7.0/etc/hadoop:.../opt/hadoop-2.7.0/share/hadoop/yarn/lib/*"
4
5  -- Resource and environment setup.
6  base_resources = {
7    ["master.jar"] = { file = MASTER_JAR_LOCATION }
8  }
9  base_env = {
10   CLASSPATH = table.concat({"${CLASSPATH}", CP, "./master.jar"}, ":"),
11 }
12
13 operator = yarn {
14   name = "Asap master",
15   timeout = 1000000000,
16   memory = 1024,
17   cores = 1,
18   nodes = "slave-1",
19   master = {
20     name = "Asap master",
21     env = base_env,
22     resources = base_resources,
23     command = {
24       base = "${JAVA_HOME}/bin/java -Xms64m -Xmx128m " ..
25         "com.cloudera.kitten.appmaster.ApplicationMaster",
26       args = { "-conf job.xml" },
27     }
28   }
29 }
```

Figure 5: Sample LUA file for setting operator properties

```
1  Constraints.Engine=Spark
2  Constraints.Output.number=1
3  Constraints.Input.number=1
4  Constraints.OpSpecification.Algorithm.name=LineCount
5  Optimization.model.execTime=gr.ntua.ece.cslab.panic.core.models.UserFunction
6  Optimization.model.cost=gr.ntua.ece.cslab.panic.core.models.UserFunction
7  Optimization.outputSpace.execTime=Double
8  Optimization.outputSpace.cost=Double
9  Optimization.cost=1.0
10 Optimization.execTime=1.0
11 Execution.Arguments.number=2
12 Execution.Argument0=In0.path.local
13 Execution.Argument1=lines.out
14 Execution.Output0.path=$HDFS_OP_DIR/line.out
15 Execution.copyFromLocal=lines.out
16 Execution.copyToLocal=In0.path
```

Figure 6: Definition of Engine for an Operator

abstract in high level or materialized. In the second case, a materialized operator is described by the aforementioned description tree which also includes the execution engine in which the operator will run. Figure 6 presents a sample operator definition. Line 1 (Constraints.Engine) defines the operator engine.

Engine Monitoring and Fault Tolerance IReS monitors the underlying engines to check when they are up and running in order to prevent possible execution errors. In order for a materialized operator to be included in the planning and execution stages, the engine defined in "Constraints.Engine" key needs to be registered to the system and also be up and running. Otherwise, IReS will not consider the specific engine in the planning stage nor execution. The list of registered execution engines as well as the status of each one can be seen by sending a GET request at `/clusterStatus/services` path of the IReS web server, or by accessing the Cockpit through the Web UI. Figure 7 shows a Cockpit with the running services list.

Integration with Engines The instructions for the execution of the operator in a specific engine are defined inside the operator's LUA script, described above. This script includes the commands that will be executed inside the operator's container, as long as the required files such as execution scripts, binaries etc. Thus, for the integration of IReS and any execution engine, the following conditions needs to be satisfied:

1. The engine in which the operator will run needs to be accessible from all the slave nodes of the underlying Yarn cluster. In case of centralized operators, the required dependencies such as libraries, files etc needs to be available in every

ASAP-IReS platform

Datasets Abstract Operators Materialized Operators Abstract Workflows Mate

Service	Status	Action
Python	true	stop
Spark	true	stop
MLLib	true	stop
MapReduce	true	stop
WEKA	true	stop
PostgreSQL	false	start
HIVE	false	start

Capacity Scheduler Resource	Min	Max
VCores	1	8
Memory	512	8192

Figure 7: Cockpit View

node of the cluster either by installing the required packages or including them as resources in the LUA file.

2. The operator's LUA file needs to be set properly by containing all the required files and the proper arguments that the container will execute.

3. The engine needs to be registered to IReS and also be up and running.

5.4 Integration of IReS with WIND Application

Running Pyspark applications The WIND operators are Pyspark applications that have dependencies on NumPy[10] and scikit-learn[11] Python libraries. Instead of Scala/-

[10] http://www.numpy.org/

[11] http://scikit-learn.org/stable/index.html

Java Spark applications that the code and its dependencies are packaged into JAR[12] files and run in the same JVM in the case of PySpark applications an extra effort should be put in order to manage the dependencies and making them available for the Python jobs on the cluster[13].

A choice would be to compile and distribute an EGG[14] file for each application but this often fails in the case of complex, compiled packages since a Python egg built on a client host is specific to the client CPU architecture because of the required C compilation.

The alternative chosen was to set up a virtual environment and install the required Python packages on each host of the cluster and specify the path to the Python binaries for the worker hosts to use by setting the `PYSPARK_PYTHON` variable in `spark-env.sh`, to point on the specific path.

In addition to this any python files needed to be distributed with the applications are passed in the `spark-submit` script by setting appropriately the `py-files` argument.

Integration of WIND Operators The sociometer workflow classifies the users using the presence of cellphone users and it identifies residents, commuters and visitors. It consists of three operators, implemented in PySpark. For each operator, we created a bash script for integration with IReS. The script set the necessary environmental variables and submits the PySpark application in Spark with the necessary arguments as dictated by a description file we generated for each operator. The script and description files, as well as any other necessary resource for running the operator, is sent in a tarball to IReS, using the respective API call[15]. When IReS receives them, it creates the particular LUA script in order to run them.

Deliverables D1.3, D9.2, D9.3, and D9.4 describe the WIND application in detail. The three operators we targeted in integrating the application are:

1. A User Profiling Operator that receives a given CDR dataset and a set of geographical regions and it returns the user profiles for each spatial region. The results are tuples that contains the following information: `<region>`, `<user_id>`,`<profile>`. The description file of the User profiling operator is shown in Figure 8. According to that description file, the operator has one input and one output, and the engine is Spark. In addition to this the algorithm implemented by this operator is “user_profiling” and it has to be the same as it is defined in the respective abstract operator. The seven input arguments are the path, IP and port where Spark runs, the spatial region (argument3) which is a CSV file with the format of the GSM tower id, and the starting and ending date for the analysis. The results are stored in HDFS into several files in the directory profiles.

[12] https://docs.oracle.com/javase/tutorial/deployment/jar/basicsindex.html
[13] https://www.cloudera.com/documentation/enterprise/5-5-x/topics/spark_python.html#concept_qzp_p3s_b5__section_yqd_bjt_25
[14] http://peak.telecommunity.com/DevCenter/PythonEggs
[15] https://project-asap.github.io/ASAP-documentation/ires_docs/install.html#creating-materialized-operators-client-side-via-the-rest-api

```
1  Constraints.Input.number=1
2  Constraints.Output.number=1
3  Constraints.Engine=Spark
4  Constraints.OpSpecification.Algorithm.name=user_profiling
5  Execution.Output0.path=$AS_IShdfs\:///profiles
6  Execution.Arguments.number=7
7  Execution.Argument0=spark\://192.168.1.62\:7077
8  Execution.Argument1=/home/asap/asap/spark-final/
9  Execution.Argument2=Input0.path
10 Execution.Argument3=aree_roma.csv
11 Execution.Argument4=forth
12 Execution.Argument5=2016-03-01
13 Execution.Argument6=2016-03-31
14 Execution.command=./user_profiling.sh
15 Execution.memory=4096
16 Execution.cores=96
17 Optimization.execTime=1.0
```

Figure 8: Description of the User Profiling Operator

```
1  Constraints.Input.number=1
2  Constraints.Output.number=1
3  Constraints.Engine=Spark
4  Constraints.OpSpecification.Algorithm.name=kmeans
5  Execution.Arguments.number=7
6  Execution.Argument0=spark\://192.168.1.62\:7077
7  Execution.Argument1=/home/asap/asap/spark-final
8  Execution.Argument2=4g
9  Execution.Argument3=Input0.path
10 Execution.Argument4=forth
11 Execution.Argument5=2016-03-01
12 Execution.Argument6=2016-03-31
13 Execution.command=./clustering.sh
14 Execution.Output0.path=$AS_IShdfs\:///centroids
15 Execution.memory=4096
16 Execution.cores=96
17 Optimization.execTime=5.0
18 Optimization.model.execTime=gr.ntua.ece.cslab.panic.core.models.UserFunction
19 Optimization.outputSpace.execTime=Double
```

Figure 9: Description of the MLlib k-means Operator

```
1  Constraints.Input.number=2
2  Constraints.Output.number=1
3  Constraints.Engine=Spark
4  Constraints.OpSpecification.Algorithm.name=stereo_type_classification
5  Execution.Arguments.number=8
6  Execution.Argument0=spark\://192.168.1.62\:7077
7  Execution.Argument1=/home/asap/asap/spark-final/
8  Execution.Argument2=2g
9  Execution.Argument3=Input0.path
10 Execution.Argument4=Input1.path
11 Execution.Argument5=forth
12 Execution.Argument6=2016-03-01
13 Execution.Argument7=2016-03-31
14 Execution.command=./classification.sh
15 Execution.memory=4096
16 Execution.cores=96
17 Execution.copyFromLocal=results
18 Optimization.execTime=1.0
19 Optimization.model.execTime=gr.ntua.ece.cslab.panic.core.models.UserFunction
20 Optimization.outputSpace.execTime=Double
```

Figure 10: Description of the Classification Operator

2. A Clustering Operator that uses K-means from Spark MLlib to identify the representative profiles and returns typical calling behaviors with a label for each behavior. Figure 9 shows the description file for the k-means operator. This description file states that this operator has one input and one output, and its engine is Spark. The seven input arguments are the path, IP and port where Spark runs, the profile dataset prefix (which can be any Hadoop-supported file system), the region name featuring in the stored results, and the starting and ending date for the analysis. The results of this operator are also stored into several HDFS files.

3. A Classification Operator that assigns each spatio-temporal user's profile to the closest representative profile based on a proper distance measure. It receives a set of the user profiles and a set of labeled calling behaviors and it returns the percentage of each label on each spatial region. Figure 10 shows the description file of the classification operator. As above, the description file states that this operator has two inputs and one output, and the engine is Spark. The eight input arguments are the path, IP and port that Spark runs, the profile dataset prefix (that can be any Hadoop-supported file system), the cluster dataset location, and the starting and ending date for the analysis. The results of this operator are stored into several local files. Note that this operator uses two inputs, both from the output of the previous and the previous-to-previous operator in the workflow.

5.5 Integration of IReS and Swan

An implementation of k-means in Swan was added as an alternative for the MLlib k-means operator, in the sociometer conceptual workflow. The Swan K-means operator is a standalone executable compiled from C++ source code with support for Cilk extensions. This operator is always preferable to the distributed implementation when the data can fit in a single node, as it is much faster than Spark. A prerequisite for running Swan k-means is that `LD_LIBRARY_PATH` should include the location of the runtime libraries of gcc5 (which is not the default compiler in all setups) and these runtime libraries must physically exist at the same path location on all worker nodes within the cluster. We solve this dependency issue by extending installation scripts, and set `LD_LIBRARY_PATH` through a wrapper python script by calling the operating system's environment API. We generated the Swan k-means clustering operator in the IReS format using the executable, a description file similar to the ones above, a wrapper script that solves any library dependencies, an invocation script that handles data formatting and representation issues, so that the Swan k-means operator (as the Spark MLlib k-means) satisfy the input and output data interface of the abstract k-means operator, a tar file with all libraries required, and a LUA script (similar to the ones described above) that registers each of these resouces with IReS. Specific to Swan, as there is no other way to view error handling and logs, we generated a wrapper Web interface that allows the user to monitor outputs and logs of the operator within the browser, as with other execution engines.

5.6 Integration of Visualization

Visualization components were integrated in coordination with webLyzard, using the WIND application to drive, test, and verify all functionality. Specifically, The Wind Telecommunication use case is realized in modules, described in details Deliverables D9.3 and D9.4, and includes a step of publication of the results to the WebLyzard portal. In particular, the data flows are:

1. Geographic coordinates of the Areas.
2. Geographic coordinates of the POIs.
3. Distribution in time of the number of SMS and CALLS. Distributions in time of the number of users having the Home for each area of the city.
4. Distribution in time of the number of users starting from an area of the city and move to another area (O/D Matrix Area-Area).
5. Distribution in time of the number of users starting from an area of the city and move to a specific POI (O/D Matrix Areas-POIs).
6. Distributions in time of the types of users for each area.

7. Distributions in time of the types of users for each POI Each distributions is transmitted as absolute numbers and as deviation from typical values.

The publisher modules use the API provided by webLyzard. Testing of the modules was done jointly with webLyzard and WIND. In order to use the API an account was set up and a token was requested for each 8-hours session. Data was sent directly to the webLyzard repositories in order to be visualized through the ASAP dashboard, available through the ASAP webpage. The uploader scripts are based on the examples provided together with the API, but the structure of the observations has been adapted according to each case:

- No changes have been made to the Observation data structure from the webLyzard API.
- The classes of the classification (Resident, Dynamic Resident, Visitor, *etc.*) are introduced in description fields.
- For some of the datasets, results have only been available for specific areas; therefore,instead of providing full geolocation coordinates only the area number is added to the dataset.
- Depending on the dataset and use cases, several weeks or months of data have been loaded at a time.

The datasets have been ingested in the webLyzard Platform through the Statistical Data API. Each indicator (Sociometer, Call Data, *etc.*) has represented observations collected during 2016. All of the datasets have been automatically converted in the native webLyzard Platform's Statistical Data format through a set of statistical publishers (*e.g.,* Sociometer Publisher). A minimal observation that will be sent to the API includes an ID, a date, a description, a value and a location, whereas an indicator will contain multiple observations. The early versions of the API are described in deliverable D6.3, the specification also being included in appendix C. The most up to date specification and documentation can be always be found at a Swagger interface and public point for the APIs[16]. In order to use the API, the users need to request an account for the API and make sure they request new keys for each day of work (*e.g.,* every 8 hours).

Several examples of uploading data to the Statistical Data API can be found on a public GitHub repository[17]:

The examples include:

1. A script for uploading data to the API.
2. Several bash scripts that demonstrate how to use each API endpoint (*e.g.,* adding/removing observations or inidcators, modifying observations, adding/deleting entire datasets).

[16] `https://api.weblyzard.com/doc/ui/`
[17] `https://github.com/weblyzard/statistical-tests`

3. A description of the conventions that need to be followed when preparing the datasets that need to be visualized (*e.g.,* naming conventions, content conventions)

4. Details about troubleshooting (*e.g.,* what to do if the token expires or if you use old versions of Ubuntu).

This repository has been used by WIND in order to create the statistical publishers for all datasets from this project. Each publisher has been successfully tested and the data was uploaded to the the webLyzard repositories.

The visualization components have also been published on a GitHub repository[18]. The visualizations presented in this repository are part of the webLyzard Visualization API that is used for fast integration of visualizations into the webLyzard dashboards.

The repository contains:

1. Bower components for several modules:
   - Basic charts
   - Line charts
   - An advanced geomap.

2. Examples for integrating a visualization into a dashboard.

3. An API reference documentation.

4. Documentation and how-to guides for troubleshooting.

Both the Statistical Data API and the Visualization API are part of the webLyzard API, which also includes a Document API (for uploading text documents and annotations) and a Search API (for searching through documents and statistical observations).

In addition to the open source visualization modules, a complete **ASAP dashboard**[19] that leveraged the webLyzard ecosystem was developed.

The creation of the dashboard was strongly influenced by our two use cases (WIND and IMR) and therefore included components to visualize both statistical data and news/social media. The dashboard also uses all the visualization components that are already available on GitHub. Data from the two use cases that was already ingested in the webLyzard Platform via the APIs can also be visualized with the latest version of the dashboard.

[18] `https://github.com/weblyzard/infovyz`
[19] `https://asap.weblyzard.com`

6 Integrated Prototype

6.1 Cluster Deployment

The ASAP prototype has been installed in the IMR and WIND clusters, and has been maintained and upgraded for the duration of WP7. FORTH has also installed the ASAP integrated prototype in a new cluster used in the evaluation of WP4, as described in Deliverable D4.3. The IMR cluster was reset in the third year of the project, so that IMR engineers that were not previously involved in WP7 could start with a clean cluster and use the documentation to install all modules of the platform. This process generated a lot of feedback and lead to major improvements in the installation scripts and the documentation of individual modules as well as the ASAP system as a whole.

6.2 VM Deployment

FORTH has created a number of virtual machines to assist with deployment, where a second phase of "fresh installation" was applied, with more cases added and issues solved in the documentation and installation scripts. FORTH has used the VM images produced to quickly deploy the ASAP platform in two internal clusters of machines, one a low-level cluster used for teaching courses at the University of Crete, and another high-end cluster of machines used for heavy analytics workloads. We have created two flavor images: one for the master and one for the worker nodes in the cluster. Using these flavors, FORTH instantiated a number of VMs running over its physical private cluster. Below we describe in more detail the process of integration in the VM.

The physical private cluster consists of 5 machines each equipped with 40 cores and 256 GB of memory, 500GB SSD drive, running CentOS 7. We have installed and configured Hadoop HDFS 2.6.4 on the cluster.

VM deployment: From Cluster to ASAP System We have deployed an analytics cluster with the integreated ASAP VM over this physical cluster, using QEMU KVM, one VM for hosting the ASAP master (asap-master) and 5 VMs for hosting the ASAP workers (asap-worker-N). The asap-master allocates 12 cores and 50GB of physical memory while the workers allocate 20 cores and 50GB of physical memory. However, these numbers can easily be adjusted in order to allocate more or less resources as it is described in VM Management. The VMs are connected to the FORTH private network using a public bridge. Therefore we had to create a network bridge in each physical machine in order to share its ethernet device with its VMs.

Networking We had also to set up all three nodes so they can smoothly ssh to each other by updating the `/etc/hosts` file in all the hosts so as to contain the machine IPs and enabling public/private ssh authorization for passwordless ssh access.

ASAP-master flavor image The asap-master image was created using `libvirt virt-install`[20] tool by an Ubuntu 16.04.1 LTS image passed as a virtual CD-ROM device. It was also setup to connect to the network via the bridge mentioned in the previous paragraph and to run a VNC server that listens on a specific port and having a specific password (12345).[21]

After the Ubuntu installation we installed and configured the following software:

- Java OpenJDK 8.
- Apache Hadoop YARN 2.7.1 configured for running a namenode, a resource manager and a historyserver.
- The Spark-Nesting[22] master, developed in WP4.
- IReS-Platform, developed in WP3 (project-asap-patch-3[23] branch).
- WMT (integration[24] branch).
- Swan runtime[25], gcc5 runtime libraries, and Swan K-Means executable wrapped into an ASAP operator, as described in this deliverable.
- Python 2.7, pip[26] and virtual environment[27] with all the required python packages used by the use cases (numpy, sklearn etc).
- Squid[28] proxy server for accessing remotely the Web UI of the running services.
- IReS monitoring tools.

ASAP-worker flavor image The ASAP-worker image was also created using libvirt virt-install tool by an Ubuntu 16.04.1 LTS image passed as a virtual CD-ROM device. It was also setup to connect to the network via the bridge mentioned in the previous paragraph and to run a VNC server that listens on a specific port and having a specific password (12345).

After the Ubuntu installation was completed the following software was installed and configured:

- Java OpenJDK 8.

[20] `http://wiki.libvirt.org/page/VM_lifecycle`

[21] This assists with quickly deploying a virtual cluster which, however should not be exposed to the internet.

[22] `https://github.com/project-asap/spark01`

[23] `https://github.com/project-asap/IReS-Platform/tree/project-asap-patch-3`

[24] `https://github.com/project-asap/workflow/tree/integration`

[25] `https://github.com/project-asap/swan_runtime`

[26] `https://pypi.python.org/pypi/pip/`

[27] `https://pypi.python.org/pypi/virtualenv`

[28] `https://help.ubuntu.com/lts/serverguide/squid.html`

- Apache Hadoop YARN 2.7.1 configured for running a datanode and a namenode.
- Spark-Nesting worker, developed in WP4.
- Swan runtime, gcc5 runtime libraries, and Swan K-Means executable wrapped into an ASAP operator, as described in this deliverable.
- Python 2.7, pip and virtual environment with all the required python packages used by the use cases (numpy, sklearn etc).
- IReS monitoring tools.

VM cloning In order to instantiate more than one VMs from the same ASAP-worker image we converted the existing image in qcow2 format using qemu-img, we changed it to read-only and we created a new image for each VM by invoking the qemu-img command with the backing file option in order to to base its copy on the read-only image. That way each VM has 50GB of disk space but its actual image size is considerably smaller.

Resizing cores and memory During the evaluation step we tried several configurations for the VMs. Therefore we changed:

- the number of virtual cores using the libvirt virsh setvcpus[29] command
- the maximum number of virtual cores using the libvirt virsh setvcpus command and by passing the –maximum argument
- the memory allocated using the libvirt virsh setmem[30] command
- the maximum memory allocated using the libvirt virsh setmaxmem[31] command

Resizing Disk In addition to this we varied the size of disk image per VM. For doing so, we created a new image with the desirable size using truncate. Then we resized the image using the libguestfs virt-resize[32] with the –expand argument in order to copy the old image to the new one and extend one of its partitions to fill the extra space. Then we edited the VM description xml to point to the new image and started the VM. Finally, from inside the VM we extended the logical volume using lvextend[33] and enlarged the unmounted file system located on the device using resize2fs[34].

[29]https://libvirt.org/sources/virshcmdref/html/sect-setvcpus.html
[30]https://libvirt.org/sources/virshcmdref/html/sect-setmem.html
[31]https://libvirt.org/sources/virshcmdref/html/sect-setmaxmem.html
[32]http://libguestfs.org/virt-resize.1.html
[33]https://linux.die.net/man/8/lvextend
[34]https://linux.die.net/man/8/resize2fs

	1 worker	3 workers	4 workers	5 workers
Spark MLlib k-means	1:33:24	35:17	33:10	28:43
Swan k-means	1:41:36	1:34:29	1:10:48	1:03:53

Table 3: Experimental evaluation for WIND workflow

Dataset Loading We use the simulated dataset provided by WIND to configure and test the VM deployment. Initially, the WIND simulated dataset (30 GB) was stored in an NFS disk that was also mounted in each VM and from there they were copied into HDFS running in the VMs. However, this consumes too much space that would be better used by the computations; Since the physical and the virtual machines are in the same private network, the solution finally adopted was to store the data in the HDFS running in the physical machines and the applications running in the VM access them via the network. As the physical machines running HDFS were also hosting the VMs, locality "hides" the addition virtual network traffic to a great extent.

7 Evaluation

As an integration test, we evaluated 2 scenarios of the WIND sociometer workflow, using two different implementations of k-means, the second operator in the workflow (Spark MLlib k-means[35] and Swan k-means, described in Deliverable D2.3). We quickly deployed different virtual clusters on the physical FORTH cluster, with one to five Spark worker nodes on corresponding worker flavor VMs. Each worker flavor VM has 20 cores and 48 GB of total memory. The VMs are not overprovisioned, so that virtual resources amount to physical resources as closely as possible.

Table 3 presents the results for the whole workflow.

[35] https://spark.apache.org/docs/latest/api/python/pyspark.mllib.html#pyspark.mllib.clustering.KMeans

References

[1] Apache hadoop yarn. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[2] Apache kitten. https://github.com/cloudera/kitten.

[3] Elasticsearch. http://www.elasticsearch.org/.

[4] Fabric. http://www.fabfile.org.

[5] Javascript. http://javascript.com.

[6] Mahout. http://mahout.apache.org.

[7] Nginx. http://nginx.org.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *PPoPP*, 1995.

[9] Apache hadoop. http://hadoop.apache.org/.

[10] Jersey. https://jersey.java.net/.

[11] Jetty. http://eclipse.org/jetty/.

[12] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. MLlib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[13] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[14] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. A unified scheduler for recursive and task-based parallelism. In *PACT*, 2011.

[15] Weka. http://weka.pentaho.com/.

[16] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

FP7 Project ASAP
Adaptable Scalable Analytics Platform

End of ASAP D7.3

ASAP System Prototype

WP 7 – Integration of the ASAP System

Nature: Report

Dissemination: Public