CHAPTER 4

Designing and Supporting Scalable Data Analytics

This chapter presents innovative systems existing on clouds for designing and implementing scalable data analytics applications. Methods for application development based on some of the discussed systems are reported and explained on real application cases. Section 4.1 introduces a series of software frameworks designed and used for implementing data analysis application on clouds systems. Section 4.2 discusses a workflow-based scalable framework and how it can be used for designing scalable data analytics applications on clouds. Section 4.3 presents a workflow-based paradigm and two associated languages for programming data analysis applications on clouds. Finally, Section 4.4 describes a set of data analysis case studies that have been implemented with the programming languages discussed in the previous sections.

4.1 DATA ANALYSIS SYSTEMS FOR CLOUDS

Recently several software tools and frameworks have been developed and used for implementing data analysis applications on clouds systems. Here we introduce a selection of those frameworks that have been compiled taking into account features such as impact on the user community, novelty, and performance. Among them we must include the Data Mining Cloud Framework (DMCF), a programming and runtime system for enabling the scalable execution of complex data analysis workflows on clouds, we recently developed, that is presented in more detail in the next sections of this chapter where the DMCF architecture and its programming interfaces are described. In fact, DMCF provides visual and script programming models that allows users to model complex data analysis workflows without worrying about low-level aspects that can be addressed by the runtime system or the cloud platform.

4.1.1 Pegasus

As mentioned in the previous chapter, Pegasus is a workflow management system developed at the University of Southern California for supporting the implementation of scientific applications also in the area of data analysis. Pegasus (Deelman et al., 2009) includes a set of software modules to execute workflow-based applications in a number of different environments, including desktops, clouds, clusters, and grids. It has been used in several scientific areas including bioinformatics, astronomy, earthquake science, gravitational wave physics, and ocean science. The Pegasus workflow management system can manage the execution of an application expressed as a visual workflow by mapping it onto available resources and executing the workflow tasks in the order of their dependencies. In particular, significant activities have been recently performed on Pegasus to support the system implementation on cloud platforms and manage computational workflows in the cloud for developing data-intensive scientific applications (Juve et al., 2010) (Nagavaram et al., 2011). The Pegasus system has been used with IaaS clouds for workflow applications and the most recent versions of Pegasus can be used to map and execute workflows on commercial and academic IaaS clouds such as Amazon EC2, Nimbus, OpenNebula, and Eucalyptus (Deelman et al., 2015).

The Pegasus system includes four main components:

- *The Mapper*: which builds an executable workflow based on an abstract workflow provided by a user or generated by the workflow composition system. To this end, this component finds the appropriate software, data, and computational resources required for workflow execution. The Mapper can also restructure the workflow in order to optimize performance, and add transformations for data management or to generate provenance information.
- *The Execution Engine* (*DAGMan*): which executes in appropriate order the tasks defined in the workflow. This component relies on the compute, storage, and network resources defined in the executable workflow to perform the necessary activities. It includes a local component and some remote ones.
- *The Task Manager*: which is in charge of managing single workflow tasks by supervising their execution on local and/or remote resources.

- *The Monitoring Component*: which monitors the workflow execution, analyzes the workflow and job logs and stores them into a workflow database used to collect runtime provenance information. This component sends notifications back to users notifying them of events like failures, success, and completion of workflows and jobs.

The Pegasus software architecture includes also an error recovery system that attempts to recover from failures by retrying tasks or an entire workflow, remapping portions of the workflow, providing workflow-level checkpointing, and using alternative data sources, when possible. The Pegasus system records provenance information including the locations of data used and produced, and which software was used with which parameters. This feature is useful when a workflow must be reproduced.

4.1.2 Swift

Swift (Wilde et al., 2011) is a implicitly parallel scripting language that runs workflows across several distributed systems, like clusters, clouds, grids, and supercomputers. The Swift language has been designed at the University of Chicago and at the Argonne National Lab to provide users with a workflow-based language for grid computing. Recently it has been ported on clouds and exascale systems.

Swift separates the application workflow logic from the runtime configuration. This approach results in a flexible development model. As the DMCF programming interface, the Swift language allows invocation and running of external application code and allows binding with application execution environments without extra coding from the user. Swift/K is the previous version of the Swift language that runs on the Karajan grid workflow engine across wide-area resources. Swift/T is a new implementation of the Swift language for high-performance computing. In this implementation, a Swift program is translated into an MPI program that uses the Turbine and ADLB runtime libraries for scalable dataflow processing over MPI. The Swift-Turbine Compiler (STC) is an optimizing compiler for Swift/T and the Swift Turbine runtime is a distributed engine that maps the load of Swift workflow tasks across multiple computing nodes. Users can also use Galaxy (Giardine et al., 2005) to provide a visual interface for Swift.

The Swift language provides a functional programming paradigm where workflows are designed as a set of code invocations with their associated command-line arguments and input and output files. Swift is based on a C-like syntax and uses an implicit data-driven task parallelism (Wozniak et al., 2014). In fact, it looks like a sequential language, but being a dataflow language, all variables are *futures*, thus execution is based on data availability. When input data is ready, functions are executed in parallel. Moreover, parallelism can be exploited through the use of the *foreach* statement. The Turbine runtime comprises a set of services that implement the parallel execution of Swift scripts exploiting the maximal concurrency permitted by data dependencies within a script and by external resource availability. Swift has been used for developing several scientific data analysis applications, such as prediction of protein structures, modeling the molecular structure of new materials, and decision making in climate and energy policy. A programming example written in Swift is discussed in Section 4.4.4.

4.1.3 Hunk

Hunk is a commercial data analysis platform developed by Splunk for rapidly exploring, analyzing, and visualizing data in Hadoop and NoSQL data stores. Hunk uses a set of high-level user and programming interfaces to offer speed and simplicity of getting insights from large unstructured and structured datasets. One of the key components of the Hunk architecture is the Splunk Virtual Index. This system decouples the storage tier from the data access and analytics tiers, so enabling Hunk to route requests to different data stores. The analytics tier is based on Splunk's Search Processing Language (SPL) designed for data exploration across large, different datasets.

The Hunk web framework allows building applications on top of the Hadoop Distributed File System (HDFS) and/or the NoSQL data store. Developers can use Hunk to build their big data applications on top of data in Hadoop using a set of well-known languages and frameworks. Indeed, the framework enables developers to integrate data and functionality from Hunk into enterprise big data applications using a web framework, documented REST API and software development kits for C#, Java, JavaScript, PHP, and Ruby. Developers can use common development languages such as HTML5 and Python.

The Hunk framework can be deployed on on-premises Hadoop clusters or private clouds and is available as a preconfigured instance on the Amazon public cloud using the Amazon Web Services (AWS). This public cloud solution allows Hunk users to utilize the Hunk facilities and tools from AWS, also exploiting commodity storage on Amazon S3, according to a pay-per-use model. Finally, the framework implements and makes available a set of applications that enable the Hunk analytics platform to explore and visualize data in NoSQL and other data stores, including Apache Accumulo, Apache Cassandra, MongoDB and Neo4j. Hunk is also provided in combination with the Cloudera's enterprise data hub to develop large-scale applications that can access and analyze big datasets.

4.1.4 Sector/Sphere

Sector/Sphere is a cloud framework designed at the University of Illinois-Chicago to implement data analysis applications involving large, geographically distributed datasets in which the data can be naturally processed in parallel (Gu and Grossman, 2009). The framework includes two components: a storage service called *Sector*, which manages the large distributed datasets with high reliability, high-performance IO, and a uniform access, and a compute service called *Sphere*, which makes use of the Sector service to simplify data access, increase data IO bandwidth, and exploit wide-area high-performance networks. Both of them are available as open source software.[1]

Sector is a distributed storage system that can be deployed over a wide area and allows users to ingest and download large datasets from any location with a high-speed network connection to the system. The system can be deployed over a large number of commodity computers (called nodes), located either within a data center or across data centers, which are connected by high-speed networks. In an example scenario, nodes in the same rack are connected by 1 Gbps networks, two racks in the same data center are connected by 10 Gbps networks, and two different data centers are connected by 10 Gbps networks. Sector assumes that the datasets it stores are divided into one or more separate files, called *slices*, which are replicated and distributed over the various nodes managed by Sector.

[1] http://sector.sourceforge.net

The Sector architecture includes a *Security server*, a *Master server*, and a number of *Slave nodes*. The Security server maintains user accounts, file access information, and the list of authorized slave nodes. The Master server maintains the metadata of the files stored in the system, controls the running of the slave nodes, and responds to users' requests. The Slaves nodes store the files managed by the system and process the data upon the request of a Sector client.

Sphere is a compute service built on top of Sector and provides a set of programming interfaces to write distributed data analysis applications. Sphere takes streams as inputs and produces streams as outputs. A stream consists of multiple data segments that are processed by Sphere Processing Engines (SPEs) using slave nodes. Usually there are many more segments than SPEs. Each SPE takes a segment from a stream as an input and produces a segment of a stream as output. These output segments can in turn be the input segments of another Sphere process. Developers can use the Sphere client APIs to initialize input streams, upload processing function libraries, start Sphere processes, and read the processing results.

4.1.5 BigML

BigML[2] is provided as a Software-as-a-Service (SaaS) for discovering predictive models from data sources and using data classification and regression algorithms. The distinctive feature of BigML is that predictive models are presented to users as interactive decision trees. The decision trees can be dynamically visualized and explored within the BigML interface, downloaded for local usage and/or integration with applications, services, and other data analysis tools. Extracting and using predictive models in BigML consists of multiple steps, as detailed in the following:

- *Data source setting and dataset creation*: A data source is the raw data from which a user wants to extract a predictive model. Each data source instance is described by a set of columns, each one representing an instance feature, or field. One of the fields is considered as the feature to be predicted. A dataset is created as a structured version of a data source in which each field has been processed and serialized according to its type (numeric, categorical, etc.).

[2] https://bigml.com

- *Model extraction and visualization*: Given a dataset, the system generates the number of predictive models specified by the user, who can also choose the level of parallelism level for the task. The interface provides a visual tree representation of each predictive model, allowing users to adjust the support and confidence values and to observe in real time how these values influence the model.
- *Prediction making*: A model can be used individually, or in a group (the so-called ensemble, composed of multiple models extracted from different parts of a dataset), to make predictions on new data. The system provides interactive forms to submit a predictive query for a new data using the input fields from a model or ensemble. The system provides APIs to automate the generation of predictions, which is particularly useful when the number of input fields is high.
- *Models evaluation*: BigML provides functionalities to evaluate the goodness of the predictive models extracted. This is done by generating performance measures that can be applied to the kind of extracted model (classification or regression).

4.1.6 Kognitio Analytical Platform

Kognitio Analytical Platform, available as cloud-based service or supplied as a preintegrated appliance, allows users to pull very large amounts of data from existing data storage systems into high-speed computer memory, allowing complex analytical questions to be answered interactively.[3]

Although Kognitio has its own internal disk subsystem, it is primarily used as an analytical layer on top of existing storage/data processing systems, for example, Hadoop clusters and/or existing traditional disk-based data warehouse products, cloud storage, etc. A feature called External Tables allows persistent data to reside on external systems. Using this feature the system administrator, or a privileged user, can easily setup access to data that resides in another environment, typically a disk store such as the above-mentioned Hadoop clusters and data warehouse systems.

To a final user, the Kognitio Analytical Platform looks like a relational database management system (RDBMS) similar to many commercial

[3] www.kognitio.com

databases. However, unlike these databases, Kognitio has been designed specifically to handle analytical query workload, as opposed to the more traditional on-line transaction processing (OLTP) workload. Key reasons of Kognitio's high performance in managing analytical query workload are:

- Data is held in high-speed RAM using structures optimized for in-memory analysis, which is different from a simple copy of disk-based data, like a traditional cache.
- Massively Parallel Processing (MPP) allows scaling out across large arrays of low-cost industry standard servers, up to thousand nodes.
- Query parallelization allows every processor core on every server to be equally involved in every query.
- Machine code generation and advanced query plan optimization techniques ensure every processor cycle is effectively used to its maximum capacity.

Parallelism in Kognitio Analytical Platform fully exploits the so-called "shared nothing" distributed computing approach, in which none of the nodes share memory or disk storage, and there is no single point of contention across the system.

4.1.7 Mahout

Apache Mahout is an open-source framework that provides scalable implementations of machine learning algorithms. The goal of this project is to provide implementations of common machine learning algorithms applicable on big input in a scalable manner. The algorithms and techniques provided by Mahout can be divided in three main categories[4]: collaborative filtering, classification and clustering. In the following, some examples for each algorithm's category are listed: analyzing user history and preferences to suggest accurate recommendations (collaborative filtering), selecting whether a new input matches a previously observed pattern or not (classification), and grouping large number of things together into clusters that share some similarity (clustering) (Anil et al., 2012). Moreover, Mahout provides common algorithms for manipulating collections of data and for math operations (e.g., linear algebra and statistics).

[4] http://mahout.apache.org/users/basics/algorithms.html

Originally, the Mahout project provided implementations of machine learning algorithms executable on the top of Apache Hadoop framework. But the comparison of the performance of Mahout algorithms on Hadoop with other machine learning libraries, showed that Hadoop spends the majority of the processing time to load the state from file at every intermediate step (i.e., intermediate data are always stored in distributed file systems) (Shahrivari, 2014). For this reason, the Mahout project has recently supported algorithm implementations on other frameworks that are more suitable for machine learning such as Apache Spark and H20.[5] Both Apache Spark and H20 process data in memory so they can achieve a significant performance gain when compared to Hadoop framework for specific classes of applications (e.g., interactive jobs, real-time queries, and stream data) (Shahrivari, 2014).

4.1.8 Spark

Spark[6] is an open-source framework for in-memory data analysis and machine learning developed at UC Berkeley in 2009. It can process distributed data from several sources, such as HDFS, HBase, Cassandra, and Hive. It has been designed to efficiently perform both batch processing applications (similar to MapReduce) and dynamic applications like streaming, interactive queries, and graph analysis. Spark is compatible with Hadoop data and can run in Hadoop clusters through the YARN module. However, in contrast to Hadoop's two-stage MapReduce paradigm in which intermediate data are always stored in distributed file systems, Spark stores data in a cluster's memory and queries it repeatedly so as to obtain better performance for applications involving interactive jobs, real-time queries, and other specific types of computation (Xin et al., 2013). The Spark project has different components:

- Spark Core contains the basic functionalities of the library such as for manipulating collections of data, memory management, interaction with distributed file systems, task scheduling, and fault recovery.

[5] http://0xdata.com/
[6] http://spark.apache.org

- Spark SQL provides API to query and manipulate structured data using standard SQL or Apache Hive variant of SQL.[7]
- Spark Streaming provides an API for manipulating streams of data.
- GraphX is a library for manipulating and analyzing big graphs.
- MLlib is a scalable machine learning library on top of Spark that implements many common machine learning and statistical algorithms.

Several big companies and organizations use Spark for big data analysis purpose[8]: for example, Ebay[9] uses Spark for log transaction aggregation and analytics, Kelkoo[10] for product recommendations, SK Telecom[11] analyses mobile usage patterns of customers.

4.1.9 Microsoft Azure Machine Learning

Microsoft Azure Machine Learning (Azure ML) is a SaaS that provides a Web-based machine learning IDE (i.e., integrated development environment) for creation and automation of machine learning workflows. Through its user-friendly interface, data scientists and developers can perform several common data analysis/mining tasks on their data and automate their workflows.

Using its drag-and-drop interface, users can import their data in the environment or use special readers to retrieve data form several sources, such as Web URL (HTTP), OData Web service, Azure Blob Storage, Azure SQL Database, Azure Table. After that, users can compose their data analysis workflows where each data processing task is represented as a block that can be connected with each other through direct edges, establishing specific dependency relationships among them. Azure ML includes a rich catalog of processing tools that can be easily included in a workflow to prepare/transform data or to mine data through supervised learning (regression and classification) or unsupervised learning (clustering) algorithms. Optionally, users can include their own custom scripts (e.g., in R or Python) to extend the

[7] https://spark.apache.org/docs/1.1.0/sql-programming-guide.html
[8] http://cwiki.apache.org/confluence/display/SPARK/Powered+By+Spark
[9] http://www.ebay.com/
[10] http://www.kelkoo.com/
[11] http://www.sktelecom.com/

tools catalog. When workflows are correctly defined, users can evaluate them using some testing dataset. Users can easily visualize the results of the tests and find useful information about models accuracy, precision, and recall. Finally, in order to use their models to predict new data or perform real time predictions, users can expose them as Web services. Through a Web-based interface, users can monitor the Web services load and use over time.

Azure Machine Learning is a fully managed service provided by Microsoft on its cloud platform; users do not need to buy any hardware/software nor to manage virtual machines manually. One of the main advantages of working with a cloud platform like Azure is its auto-scaling feature: models are deployed as elastic Web services so that users do not have to worry about scaling them if models usage increases.

4.1.10 ClowdFlows

ClowdFlows (Kranjc et al., 2012) is an open source cloud-based platform for the composition, execution, and sharing of data analysis workflows. It is provided as a software as a service that allows users to design and execute visual workflows through a simple Web browser and so it can be run from most devices (e.g., desktop PCs, laptops, and tablets).

ClowdFlows is based on two software components: the workflow editor (provided by a Web browser) and the server side application that manages the execution of the application workflows and hosts a set of stored workflows. The server side consists of methods for supporting the client-side workflow editor in the composition and for executing workflows, and a relational database of workflows and data.

The workflow editor includes a workflow canvas and a widget repository. The widget repository is a list of all the available workflow components that can be added to the workflow canvas. The repository includes a set of default widgets. According to this approach, the CloudFlows service-oriented architecture allows users to include in their workflow the implementations of various algorithms, tools and Web services as workflow elements. For example, the Weka's algorithms have been included and exposed as Web services and so they can be added in a workflow application.

ClowdFlows is also easily extensible by importing third-party Web services that wrap open-source or custom data mining algorithms. To this end, a user has only to insert the WSDL URL of a Web service to create a new workflow element that represents the Web service in a workflow application.

4.2 HOW TO DESIGN A SCALABLE DATA ANALYSIS FRAMEWORK IN CLOUDS

As we discussed, designing and executing data analysis workflows over cloud platforms requires the availability of effective programming environments and efficient runtime systems. According to this observation, we designed and prototyped the Data Mining Cloud Framework (DMCF), a programming and runtime system for enabling the scalable execution of complex data analysis workflows on clouds. DMCF provides visual and script-based programming interfaces that allow developers to model complex data analysis workflows without worrying about low-level aspects that can be addressed by the runtime system or the cloud platform. In addition, the DMCF's runtime fully exploits the underlying cloud infrastructure enabling workflow execution on multiple virtual machines, this way reducing the turnaround times of complex data analysis workflows.

DMCF functionality is provided according with the SaaS model. This means that no installation is required on the user's machine: the DMCF visual user interface works in any modern Web browser, and so it can be invoked from most devices, including desktop PCs, laptops, and tablets. This is a key feature for users who need ubiquitous and seamless access to scalable data analysis services, without having to cope with installation and system management issues. A distinctive feature of DMCF is that it has been designed to run on top of a Platform-as-a-Service (PaaS) cloud. An important advantage of this approach is the independence from the infrastructure layer. In fact, the DMCF's components are mapped into PaaS services, which in turn are implemented on infrastructure components. Possible changes to the cloud infrastructure affect only the infrastructure/platform interface, which is managed by the cloud provider, and therefore DMCF's implementation and functionality are not influenced. In addition, the

PaaS approach facilitates the implementation of the system on a public cloud, which free final users and organizations from any hardware and OS management duties.

Another key feature of DMCF is the provision of novel data mining-specific workflow formalisms (Data and Tool arrays) that significantly ease the design of parallel and distributed data analysis applications. A Data array allows representing an ordered collection of input/output data sources in a single workflow node. Similarly, a Tool array node represents multiple instances of the same tool. Array nodes are effective to fork the concurrent execution of many parallel tasks to cloud resources, thus improving scalability.

4.2.1 Architecture and Execution Mechanisms

The architecture of DMCF includes different kinds of components that can be grouped into storage and compute components (see Figure 4.1).

The storage components include:

- *Data Folder*, which contains data sources and the results of data analysis processes, and *Tool Folder*, which contains libraries and executable files for data selection, preprocessing, transformation, data mining, and results evaluation.

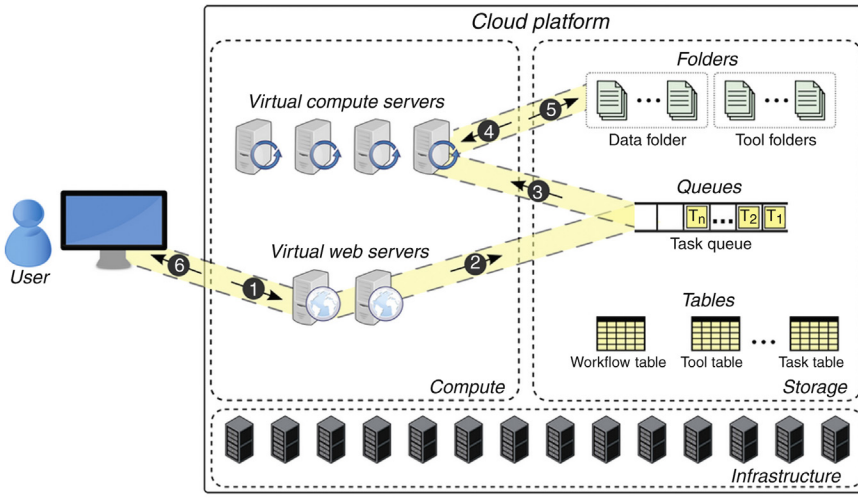


Fig. 4.1. Architecture of the Data Mining Cloud Framework.

- *Workflow Table*, *Tool Table* and *Task Table*, which contain metadata information associated with workflows, tools, and tasks.
- *Task Queue*, which contains the tasks ready for execution.

The compute components are:

- A pool of *Virtual Compute Servers*, which are in charge of executing the data analysis tasks.
- A pool of *Virtual Web Servers*, which host the Web-based user interface.

The user interface provides access to three functionalities: (i) *App submission*, which allows users to define and submit data analysis applications; (ii) *App monitoring*, which is used to monitor the status and access results of the submitted applications; (iii) *Data/Tool management*, which allows users to manage input/output data and tools.

Design and execution of a data analysis application in DMCF is a multistep process (see Figure 4.1):

1. The user accesses the Website and designs the workflow through a Web-based interface.
2. After workflow submission, the system creates a set of tasks and inserts them into the Task Queue.
3. Each idle Virtual Compute Server picks a task from the Task Queue, and concurrently executes it.
4. Each Virtual Compute Server gets the input dataset from its location. To this end, a file transfer is performed from the Data Folder where the dataset is located, to the local storage of the Virtual Compute Server.
5. After task completion, each Virtual Compute Server puts the result on the Data Folder.
6. The Website notifies the user whenever each task is completed, and allows her/him to access the results.

The set of tasks created on the second step depends on how many tools are invoked within the workflow. All the potential parallelism of the workflow is exploited by using the needed Virtual Compute Servers. In addition, multithreaded tasks exploit all the cores available on the Virtual Compute Servers they are assigned to.

4.2.2 Implementation on Microsoft Azure

A version of the Data Mining Cloud Framework has been implemented using Microsoft Azure, whose components and features have been introduced in Chapter 2. The choice of Azure was guided by the need of satisfying three main requirements:

- The use of a PaaS system, because implementing the execution mechanisms of DMCF does not require the low level facilities provided by a IaaS.
- The use of a platform whose components match the needs of the components defined in DMCF's architecture.
- The use of a public cloud, to free final users and organizations from any hardware and OS management duties.

Microsoft Azure satisfies all the requirements above, since it is a public PaaS platform whose components fully fit with those defined by DMCF. In the following, we describe how the generic components of DMCF's architecture are mapped to the Azure's components.

As shown in Figure 4.1, the architecture of DMCF distinguishes its high-level components into two groups, *Storage* and *Compute*, with the same approach followed by Azure and other cloud platforms. This made possible to implement the data and computing components of DMCF by fully exploiting the Storage and Compute components and functionalities provided by Azure.

For the Storage components, the following mapping with Azure was adopted:

1. Data Folder and Tool Folder are implemented as Azure's Blob containers;
2. Workflow Table, Tool Table, and Task Table, are implemented as nonrelational Tables;
3. Task Queue is implemented as an Azure's Queue.

For the Compute components, the following mapping was adopted:

1. Virtual Compute Servers are implemented as Worker Role instances;
2. Virtual Web Servers are implemented as Web Role instances.

Each Worker Role instance executes the operations described in Figure 4.2.

```
while true do
    if TaskQueue.isNotEmpty() then
        task ← TaskQueue.getTask();
        TaskTable.update(task, 'running');
        localInput = <local input location>;
        localOutput = <local output location>;
        transfer(task.input, localInput);
        taskStatus ← execute(task.algorithm, task.paramaters, localInput, localOutput);
        if taskStatus = 'done' then
            transfer(localOutput, task.outputBlobURI);
            TaskTable.update(task, 'done');
        end
        else
            TaskTable.update(task, 'failed');
        end
        TaskQueue.remove(task);
        delete(localInput);
        delete(localOutput);
    end
end
```

Fig. 4.2. Pseudocode of the operations performed by each Worker Role instance in DMCF.

As shown by the algorithm, file transfers are performed when input/output data have to be moved between storage and servers. To reduce the impact of data transfer on the overall execution time, DMCF exploits the Azure's Affinity Group feature, which allows Data Folder and Virtual Compute Servers to be located near to each other in the same data center for optimal performance. At least one Virtual Web Server runs continuously in the cloud, as it serves as user front-end for DMCF. Moreover, the user indicates the minimum and maximum number of Virtual Compute Servers he/she wants to use. DMCF exploits the auto-scaling features of Microsoft Azure that allows spinning up or shutting down Virtual Compute Servers, based on the number of tasks ready for execution in the DMCFs Task Queue.

4.3 PROGRAMMING WORKFLOW-BASED DATA ANALYSIS

Workflows may encompass all the steps of discovery based on the execution of complex algorithms and the access and analysis of scientific data. In data-driven discovery processes, knowledge discovery workflows can produce results that can confirm real experiments or provide insights that cannot be achieved in laboratories. In particular, DMCF allows users to program workflow applications using two languages (Marozzo et al., 2015):

- *VL4Cloud* (Visual Language for Cloud), a visual programming language that lets users develop applications by programming the workflow components graphically.
- *JS4Cloud* (JavaScript for Cloud), a scripting language for programming data analysis workflows based on JavaScript.

Both languages use two key programming abstractions:

- *Data* elements denote input files or storage elements (e.g., a dataset to be analyzed) or output files or stored elements (e.g., a data mining model).
- *Tool* elements denote algorithms, software tools or complex applications performing any kind of operation that can be executed on a data element (data mining, filtering, partitioning, etc.).

A descriptor, expressed in JSON format, is associated with each Data and Tool element. A Tool descriptor includes a reference to its executable, the required libraries, and the list of input and output parameters. Each parameter is characterized by name, description, type, and can be mandatory or optional. An example of descriptor for a data classification tool is presented in Figure 4.3.

The JSON descriptor of a new tool is automatically created through a guided procedure provided by DMCF, which allows users to specify all

```
"J48": {
  "libraryList": ["java.exe","weka.jar"],
  "executable": "java.exe -cp weka.jar weka.classifiers.trees.J48",
  "parameterList":[{
      "name": "dataset", "flag": "-t",
      "mandatory": true, "parType": "IN",
      "type": "file", "array": false,
      "description": "Input dataset"
    },{
      "name": "confidence", "flag": "-C",
      "mandatory": false, "parType": "OP",
      "type": "real", "array": false,
      "description": "Confidence value",
      "value": "0.25"
    },{
      "name": "model", "flag": "-d",
      "mandatory": true, "parType": "OUT",
      "type": "file", "array": false,
      "description": "Output model"}]}
```

Fig. 4.3. Example of a tool descriptor in JSON format.

the needed information for invoking the tool (executable, input and output parameters, etc.). A DMCF module, called Tool Manager, supports the deployment of new tools in the system allowing their subsequent use in DMCF's workflows. To this end, the Tool Manager performs the following tasks:

1. uploads libraries and executable files in Tool Folder;
2. creates a tool descriptor in JSON format;
3. publishes the JSON descriptor in Tool Table.

Similarly, a Data descriptor contains information to access an input or output file, including its identifier, location, and format. Differently from Tool descriptors, Data descriptors can also be created dynamically as a result of a task operation during the execution of a workflow. For example, if a workflow W reads a dataset D_i and creates (writes) a new dataset D_j, only D_i's descriptor will be present in the environment before W's execution, whereas D_j's descriptor will be created at runtime.

Another common element is the *task* concept, which represents the unit of parallelism in the DMCF model. A task is a Tool invoked in the workflow, which is intended to run in parallel with other tasks on a set of cloud resources. According to this approach, VL4Cloud and JS4Cloud implement a *data-driven task parallelism*. This means that, as soon as a task does not depend on any other task in the same workflow, the runtime asynchronously spawns it to the first available virtual machine. A task T_j does not depend on a task T_i belonging to the same workflow (with $i \neq j$), if T_j during its execution does not read any data element created by T_i.

4.3.1 VL4Cloud

In *VL4Cloud*, workflows are directed acyclic graphs whose nodes represent data and tools elements. The nodes can be connected with each other through direct edges, establishing specific dependency relationships among them. When an edge is being created between two nodes, a label is automatically attached to it representing the type of relationship between the two nodes.

For example, Figure 4.4 shows a `J48` Tool (an implementation of the C4.5 algorithm (Quinlan, 1993) provided by the Weka toolkit (Hall et al., 2009)) that takes in input a `TrainSet` and generates a `Model`.

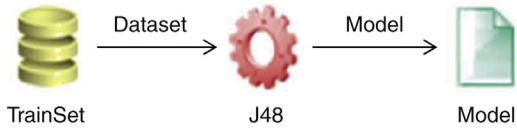


Fig. 4.4. Example of a tool connected to input dataset and output model.

For each Tool node, input/output connections are allowed on the basis of the Tool descriptor present in Tool Table.

Data and Tool nodes can be added to the workflow as single instances or in an array form. A data array is an ordered collection of input/output data elements, while a tool array represents multiple instances of the same tool.

In order to present the main features of VL4Cloud, we use as an example a data analysis application composed of several sequential and parallel steps. The example application analyzes a dataset by using *n* instances of the J48 classification algorithm that work on *n* partitions of the training set and generate *n* classification models. By using the *n* generated models and the test set, *n* predictors produce in parallel *n* classified datasets. In the final step of the workflow, a voter generates the final classification by assigning a class to each data item. This is done by choosing the class predicted by the majority of the models (Zhou and Li, 2010).

Figure 4.5a shows a snapshot of the visual interface with the first step of the workflow, where the original dataset is split in training and test set by a partitioning tool. Since a set of parameters is associated with each workflow node, the interface allows users to configure them through a pop-up panel. For example, the central part of Figure 4.5a shows the configuration panel for the partitioning tool. In this case, only one parameter can be specified, namely which percentage of the input dataset must be used as training set. In a second step, the training set is partitioned into 16 parts using another partitioning tool (see Figure 4.5b). The 16 training sets resulting from the partitioning are represented in the workflow as a single data array node, labeled as `TrainSetPart[16]`. Figure 4.5c shows the third step of the workflow, in which the 16 training sets are analyzed in parallel by 16 instances of the J48 classification algorithm, to produce the same number of classification

Fig. 4.5. Example of workflow composition using VL4Cloud. (a) Partitioning the input dataset. (b) Partitioning the train set. (c) Analyzing each train set part. (d) Classifying the test set. (e) Voting.

models. A tool array node, labeled as `J48[16]`, is used to represent the 16 instances of the J48 algorithm, while another data array node, labeled as `Model[16]`, represents the models generated by the classification algorithms. In practice, this part of the workflow specifies that `J48[i]` takes in input `TrainSetPart[i]` to produce `Model[i]`, for $1 \leq i \leq 16$. The fourth step classifies the test set using the 16 models generated on the previous step (see Figure 4.5d). The classification is performed by 16 predictors that run in parallel to produce 16 classified test sets. In

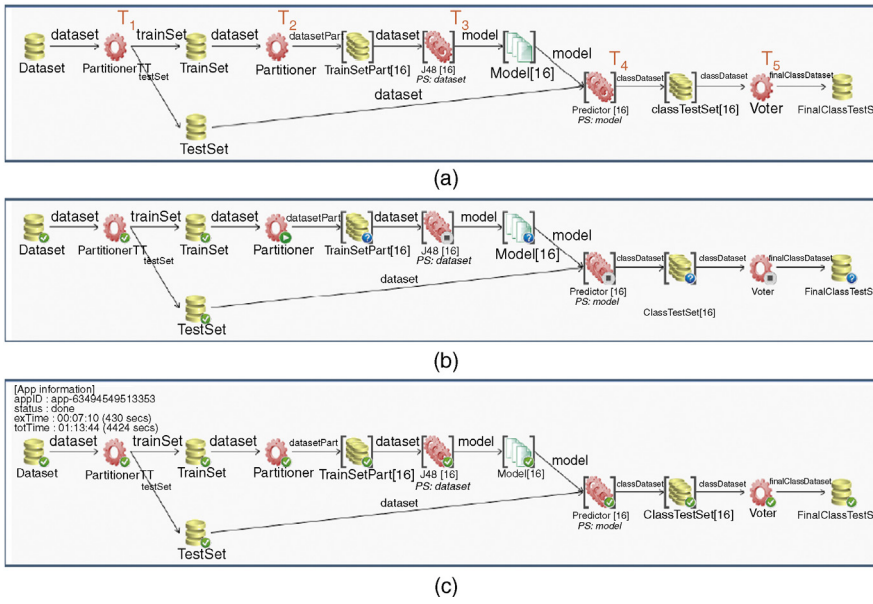


Fig. 4.6. Execution of the VL4Cloud workflow shown in Figure 4.5. (a) Workflow tasks. (b) Workflow running. (c) Workflow completed.

more detail, `Predictor[i]` takes in input `TestSet` and `Model[i]` to produce `ClassTestSet[i]`, for $1 \leq i \leq 16$. Finally, the 16 classified test sets are passed to a voter that produces the final dataset, labeled as `FinalClassTestSet` (see Figure 4.5e).

When the workflow design is complete, execution proceeds as detailed in the following. The five tools (`PartitionerTT`, `Partitioner`, `J48`, `Predictor`, and `Voter`) are translated into five groups of tasks, indicated as `T1`...`T5` in Figure 4.6a.

The execution order of the workflow tasks depends on the dependencies specified by the workflow edges. To ensure the correct execution order, to each task is associated a list of tasks that must be completed before starting its execution. Figure 4.7 shows a possible order in which

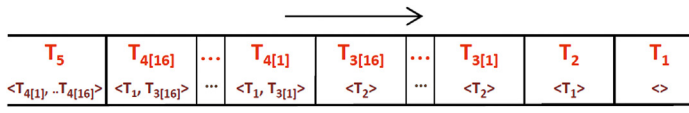


Fig. 4.7. A possible order in which the tasks are inserted into TaskQueue.

the tasks are generated and inserted into TaskQueue. For each task, the list of tasks to be completed before its execution is included. Note that task group T3, which represents the execution of 16 instances of `J48`, is translated into 16 tasks `T3[1]...T3[16]`. Similarly, `T4` is translated into tasks `T4[1]...T4[16]`.

According with the tasks dependencies specified by the workflow, the execution of `T2` will start after completion of `T1`. As soon as `T2` completes, the 16 tasks that compose `T3` can be run concurrently. Each task `T4[i]` can be executed only after completion of both `T2` and `T3[i]`, for $1 \leq i \leq 16$. Finally `T5` will start after completion of all tasks that compose `T4`. Figure 4.6b shows a snapshot of the workflow taken during its execution. The figure shows that `PartitionerTT` has completed the execution, `Partitioner` is running, while the other tools are still submitted. Figure 4.6c shows the workflow after completion of its execution. Some statistics about the overall application are shown on the upper left part of the window. In this example, it is shown that, using 16 virtual machines, the workflow completed 430 s after its submission, whereas the total execution time (i.e., the sum of the execution times of all the tasks) is 4424 s.

4.3.2 JS4Cloud

In *JS4Cloud*, workflows are defined through a JavaScript code that interacts with Data and Tool elements by three sets of functions, listed in Table 4.1:

- *Data Access*: for accessing a Data element stored in the cloud;
- *Data Definition*: to define a new Data element that will be created at runtime as a result of a Tool execution;
- *Tool Execution*: to invoke the execution of a Tool available in the cloud.

Data Access is implemented by the `Data.get` function, which is available in two versions: the first one receives the name of a data element, and returns a reference to it. The second one returns an array of references to the data elements whose name matches the provided regular expression. For example, the following statement:

```
var ref = Data.get("Census");
```

Table 4.1 JS4Cloud Functions

Functionality	Function	Description
Data Access	Data.get(<*dataName*>);	Returns a reference to the data element with the provided name.
	Data.get(new RegExp(<*regular expression*>));	Returns an array of references to the data elements whose name match the regular expression.
Data Definition	Data.define(<*dataName*>);	Defines a new data element that will be created at runtime.
	Data.define(<*arrayName*>,<*dim*>);	Define an array of data elements.
	Data.define(<*arrayName*>,[<dim_1>,...,<dim_n>]);	Define a multi-dimensional array of data elements.
Tool Execution	<*toolName*>({<par_1>:<val_1>,...,<par_n>:<val_n>});	Invokes an existing tool with associated parameter values.

assigns to variable `ref` a reference to the dataset named `Census`, while the following statement:

```
var ref = Data.get(new RegExp("^CensusPart"));
```

assigns to `ref` an array of references (`ref[0]...ref[n-1]`) to all the datasets whose name begins with `CensusPart`.

Data Definition is done through the `Data.define` function, available in three versions: the first one defines a single data element; the second one defines a one-dimensional array of data elements; the third one defines a multidimensional array of data elements. For instance, the following piece of code:

```
var ref = Data.define("CensusModel");
```

defines a new data element named `CensusModel` and assigns its reference to variable `ref`, while the following statement:

```
var ref = Data.define("CensusModel", 16);
```

defines an array of data elements of size 16 `(ref[0]...ref[15])`.

The following is an example statement defining a bidimensional array of data elements of size 4 times 16:

```
var ref = Data.define("ClassDataset", [4,16]);
```

In all cases, the data elements defined using `Data.define` will be created at runtime as result of a tool execution.

Differently from Data Access and Data Definition, there is not a named function for Tool Execution. In fact, the invocation of a tool `T` is made by calling a function with the same name of `T`. The DMCF makes the tools available to the users by loading their descriptions into the environment. For example, the J48 tool defined by the descriptor in Figure 4.3 can be invoked as in the following statement:

```
J48({dataset:DRef, confidence:0.05, model:MRef});
```

where `DRef` is a reference to the dataset to be analyzed, previously introduced using the `Data.get` function, and `MRef` is a reference to the model to be generated, previously introduced using `Data.define`.

From an implementation perspective, the `Data.get` primitive returns a reference to a data element stored in Data Folder, which is a persistent storage independent from the local storage of each Virtual Compute Server. Whenever a data element referenced by `Data.get` must be processed, it is transparently copied to the local storage of the virtual server onto which processing will take place. Similarly, the `Data.define` primitive defines a new data element that will be created at runtime in the local storage of a virtual server, as a consequence of a tool execution. The data elements created in such a way are then transparently copied to the Data Folder.

Figure 4.8 shows the JS4Cloud workflow corresponding to VL-4Cloud workflow shown in Figure 4.5. Parallelism is exploited in the for loop at line 7, where up to 16 instances of the J48 classifier are executed in parallel on 16 different partitions of the training set, and in the for loop at line 10, where up to 16 instances of the Predictor are executed in parallel to classify the test set using 16 different classification models.

The DMCF interface allows users to monitor the execution of JS-4Cloud scripts. To this end, beside each code line number, a colored circle indicates the status of execution. A green circle indicates that the tasks on a given line have completed their execution; blue circles

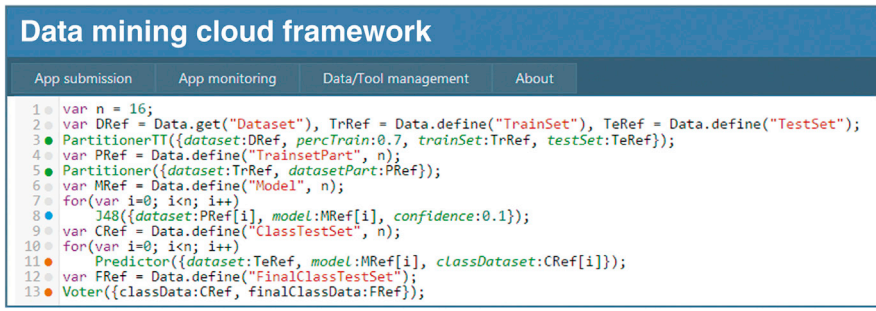


Fig. 4.8. JS4Cloud workflow corresponding to the VL4Cloud workflow shown in Figure 4.5.

denote tasks that are still running; orange circles denote tasks that are waiting to be executed.

4.3.3 Workflow Patterns in DMCF

In the following we describe how the basic control flow patterns can be programmed with VL4Cloud and JS4Cloud. We focus on basic patterns (Bharathi et al., 2008) such as *single task*, *pipeline*, *data partitioning* and *data aggregation*, and on three additional patterns supported by DMCF, namely *parameter sweeping*, *input sweeping*, and *tool sweeping*. For each pattern, we first introduce an example as a VL4Cloud workflow, and then we show how the same example can be coded using JS4Cloud.

4.3.3.1 Single Task

An example of single-task pattern is shown in the following figure:

This example represents a `K-Means` tool that produces a clustering model from the `Customers` dataset. In this example, a configuration parameter that has been set by the user, for example, the number of clusters for the `K-Means` tool, is not visible. The following JS4Cloud script is equivalent to the visual workflow shown above:

```
var DRef = Data.get("Customers");
```

```
var nc = 5;

var MRef = Data.define("ClustModel");

K-Means({dataset:DRef, numClusters:nc, model:MRef});
```

The script accesses the dataset to be analyzed (`Customers`), sets to five the number of clusters, and defines the name of data element that will contain the clustering model (`ClustModel`). Then, the `K-Means` tool is invoked along with the parameters indicated in its JSON descriptor (input dataset, number of clusters, output model).

4.3.3.2 Pipeline

In the pipeline pattern, the output of a task is the input for the subsequent task, as in the following example:

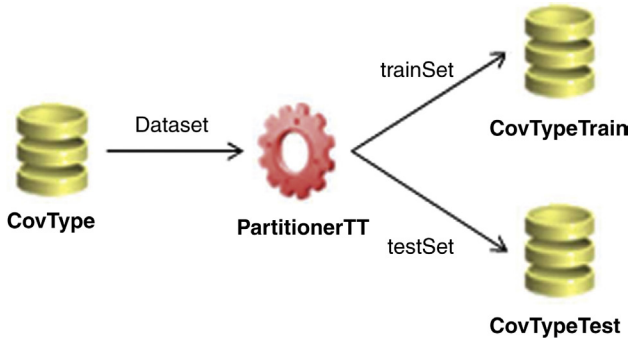
The first part of the shown example extracts a sample from an input dataset using a tool named `Sampler`. The second part creates a classification model from the sample using the `J48` tool. This pattern example is implemented in JS4Cloud as follows:

```
var DRef = Data.get("Census");

var SDRef = Data.define("SCensus");

Sampler({input:DRef, percent:0.25, output:SDRef});

var MRef = Data.define("CensusTree");

J48({dataset:SDRef, confidence:0.1, model:MRef});
```

In this case, since `J48` receives as input the output of `Sampler`, the former will be executed only after completion of the latter.

4.3.3.3 Data Partitioning

The data partitioning pattern produces two or more output data from an input data element, as in the following example:



In this example a training set and a test set are extracted from a dataset, using a tool named `PartitionerTT`. With JS4Cloud, this can be written as follows:

```
var DRef = Data.get("CovType");

var TrRef = Data.define("CovTypeTrain");

var TeRef = Data.define("CovTypeTest");

PartitionerTT({dataset:DRef, percTrain:0.70, trainSet:TrRef,
   testSet:TeRef});
```

If data partitioning is used to divide a dataset into a number of splits, the DMCF's data array formalism can be conveniently used as in the following example:

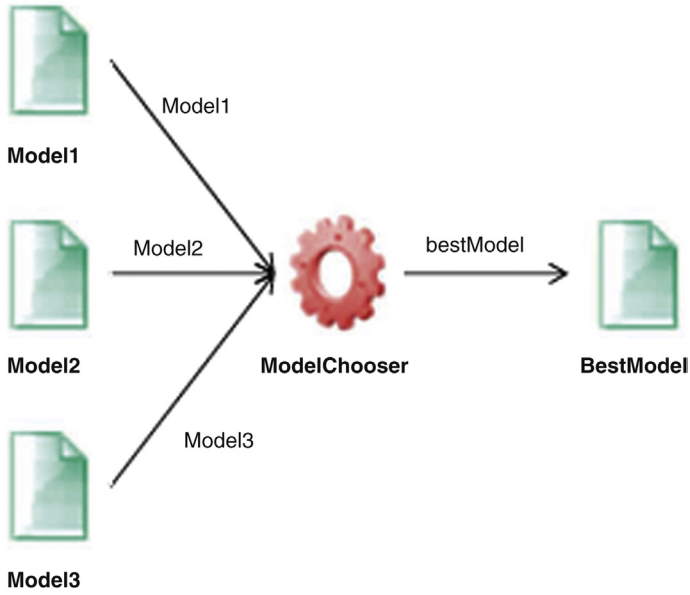
In this case, a `Partitioner` tool splits a dataset into 16 parts. The corresponding JS4Cloud code is:

```
var DRef = Data.get("NetLog");

var PRef = Data.define("NetLogParts", 16);

Partitioner({dataset:DRef, datasetParts:PRef});
```

Note that an array of 16 data elements is first defined and then created by the `Partitioner` tool.

4.3.3.4 Data Aggregation

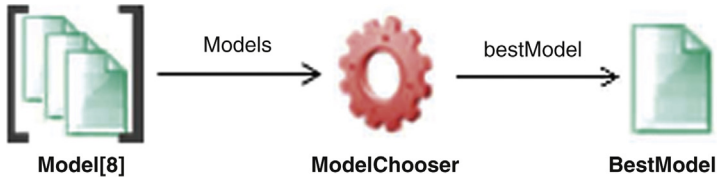
The data aggregation pattern generates one output data from multiple input data, as in the following example:



In this example, a `ModelChooser` tool takes as input three data mining models and chooses the best one based on some evaluation criteria. The corresponding JS4Cloud script is:

```
var M1Ref = Data.get("Model1");

var M2Ref = Data.get("Model2");

var M3Ref = Data.get("Model3");

var BMRef = Data.define("BestModel");

ModelChooser({model1:M1Ref, model2:M2Ref, model3:M3Ref,
   bestModel:BMRef});
```

DMCF's data arrays may be used for a more compact visual representation. For example, the following pattern example chooses the best one among 8 models:



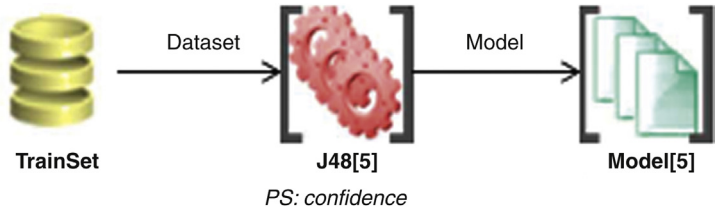
The same task can be coded as follows using JS4Cloud:

```
var BMRef = Data.define("BestModel");

ModelChooser({models:MsRef, bestModel:BMRef});
```

In this script, it is assumed that `MsRef` is a reference to an array of models created on a previous step.

4.3.3.5 Parameter Sweeping

Parameter sweeping is a data analysis pattern in which a dataset is analyzed in parallel by multiple instances of the same tool with different parameters, as in the following example:



In this example, a training set is processed in parallel by five instances of J48 to produce the same number of data mining models. The DMCF's tool array formalism is used to represent the five tools in a compact form. The J48 instances differ from each other by the value of a single parameter, the confidence factor, which has been configured (through the visual interface) to range from 0.1 to 0.5 with a step of 0.1. The equivalent JS4Cloud script is:

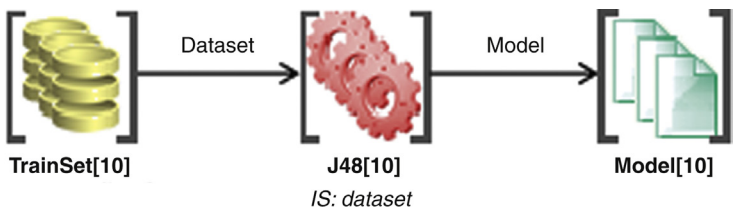
```
var TRef = Data.get("TrainSet");

var nMod = 5;

var MRef = Data.define("Model", nMod);

var min = 0.1;
```

```
var max = 0.5;

for(var i = 0; i<nMod; i++)

  J48({dataset:TRef, model:MRef[i], confidence:
    (min + i*(max-min)/(nMod-1))});
```

In this case, the for loop is used to create five instances of J48, where the *i*th instance takes as input the same training set (`TRef`), and produces a different model (`MRef[i]`), using a specific value for the confidence parameter (0.1 for `J48[0]`, 0.2 for `J48[1]`, and so on). It is worth noticing that the tools are independent of each other, and so the runtime can execute them in parallel.

4.3.3.6 Input Sweeping

Input sweeping that exploits data parallelism is a pattern in which a set of input data is analyzed independently to produce the same number of output data. It is similar to the parameter-sweeping pattern, with the difference that in this case the sweeping is done on the input data rather than on a tool parameter. An example of input sweeping pattern is represented in the following figure:

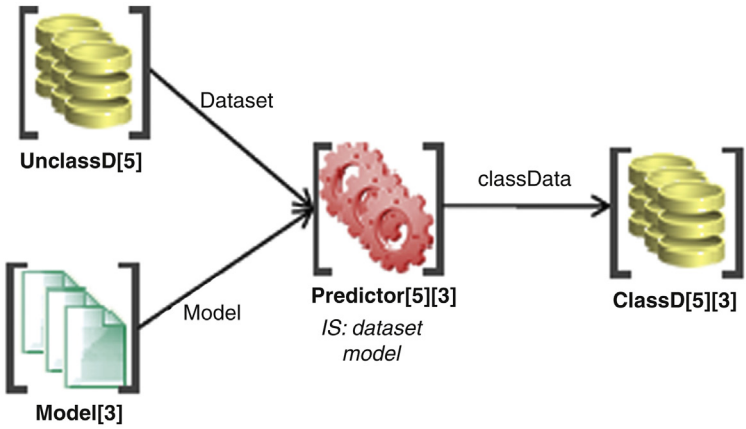


In this example, 10 training sets are processed in parallel by 10 instances of `J48`, to produce the same number of data mining models. Data arrays are used to represent both input data and output models, while a tool array is used to represent the `J48` tools. The following JS4Cloud script corresponds to the example shown above:

```
var nMod = 10;

var MRef = Data.define("Model", nMod);

for(var i = 0; i<nMod; i++)

  J48({dataset:TsRef[i], model:MRef[i], confidence:0.1});
```

It is assumed that `TsRef` is a reference to an array of training sets created on a previous step. The for loop creates 10 instances of `J48`, where the *i*th instance takes as input `TsRef[i]` to produce `MRef[i]`.

Another example of input sweeping pattern is represented in the following figure:



In this case there are 15 instances of a `Predictor`. Each `Predictor` takes as input one unclassified dataset and one model, and generates concurrently one classified dataset. The following JS4Cloud script corresponds to this example:

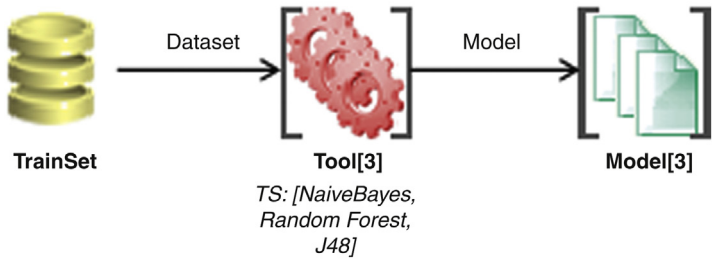
```
var nData = 5, nMod = 3;

var CRef = Data.define("ClassD", [nData, nMod]);

for(var i = 0; i<nData; i++)

  for(var j = 0; j<nMod; j++)

    Predictor({dataset:DRef[i], model:MRef[j],
      classDataset:CRef[i][j]});
```

Here is assumed that `DRef` is a reference to an array of unlabeled datasets, and `MRef` is a reference to an array of models, created on previous steps. The double for loop creates a bidimensional array of classified datasets, denoted `CRef`, where `CRef[i][j]` is the classified dataset generated by a `Predictor` instance on `DRef[i]` using `MRef[j]`. Also in this

case, since the tools are independent of each other, they can be executed in parallel by the runtime.

4.3.3.7 Tool Sweeping

Tool sweeping is a pattern in which a dataset is analyzed in parallel by different tools, as in the following example:



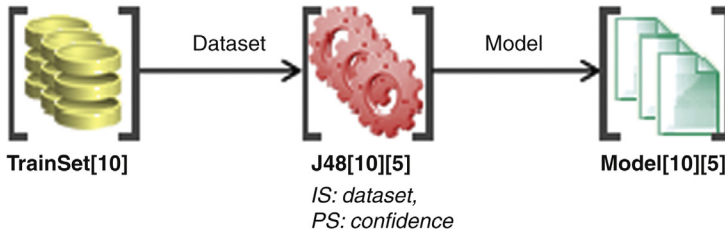
In this example, each one of the three classification tools (`NaiveBayes, RandomForest, J48`) analyzes the same training set to produce a classification model. The result of this pattern example are three different models. This corresponds to the following JS4Cloud script:

```
var TRef = Data.get("TrainSet");

var MRef = Data.define("Model", 3);

NaiveBayes({dataset:TRef, model:MRef[0],
   kernelDensity:true});

RandomForest({dataset:TRef, model:MRef[1],
   numberOfTrees:500});

J48({dataset:TRef, model:MRef[2], confidence:0.1});
```

4.3.3.8 Combination of Sweeping Patterns

In JS4Cloud, it is possible to combine parameter, input and tool sweeping patterns. In the following we show two examples of sweeping pattern combinations.

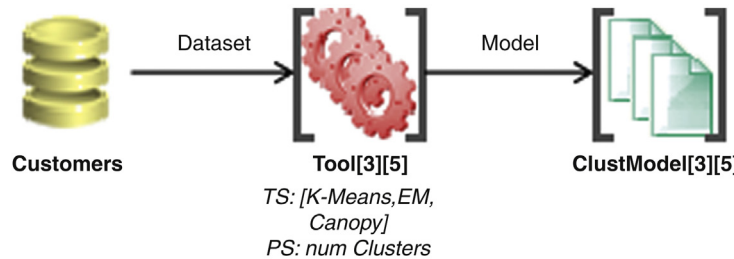
As a first example, we show an input/parameter sweeping, that is, the combination of input and parameter sweeping. With this pattern, each input data is analyzed in parallel by multiple instances of the same tool with different parameters, as in the following figure:



In this example, each of the 10 training sets is processed by five instances of `J48` to produce five data mining models. Thus, there are in total 50 instances of `J48`, represented by a bidimensional array of size 10 times 5, that generate the same number of models. The following JS-4Cloud script corresponds to this example:

```
var nTr = 10;

var conf = [0.1, 0.2, 0.3, 0.4, 0.5];

var MRef = Data.define("Model", [nTr, conf.length]);

for(var i = 0; i<nTr; i++)

  for(var j = 0; j<conf.length; j++)

    J48({dataset:TsRef[i], model:MRef[i][j],
       confidence:conf[j]});
```

The second example is a tool/parameter sweeping, that is, the combination of tool and parameter sweeping. With this pattern, a dataset is analyzed in parallel by a set of tools, each of them configured with different parameters, as in the following figure:



In this workflow, three clustering tools, `K-Means`, `EM`, and `Canopy`, analyze in parallel the same dataset. Each clustering algorithm is executed five times varying an algorithm parameter (the number of clusters). Thus, for each of the three clustering tools five instances are created. They are rerepresented by a bidimensional array of size 3 times 5. This is the equivalent JS4Cloud script:

```
var DRef = Data.get("Customers");

var nt = 3;

var nc = [3,4,5,6];

var MRef = Data.define("ClustModel", [nt, nc.length]);

for(var i = 0; i<nc.length; i++){

  K-Means({dataset:DRef, numClusters:nc[i],
    model:MRef[0,i]});

  EM({dataset:DRef, numClusters:nc[i], model:MRef[1,i]});

  Canopy({dataset:DRef, numClusters:nc[i],
    model:MRef[2,i]});

}
```

4.4 DATA ANALYSIS CASE STUDIES

In this section we describe four examples of data analysis workflows designed using some of the programming languages discussed in the previous sections, namely:

- a trajectory mining workflow designed using VL4Cloud;
- an ensemble learning workflow programmed with JS4Cloud;
- a data analysis workflow with MapReduce computations executed in DMCF;
- a parallel classification workflow programmed using Swift.

4.4.1 Trajectory Mining Workflow Using VL4Cloud

The increasing pervasiveness of mobile devices along with the use of technologies like GPS, Wi-fi networks, RFID, and sensors, allows for

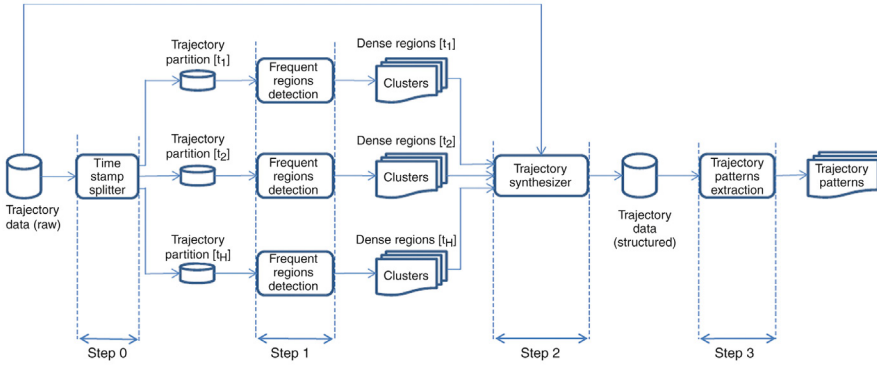


Fig. 4.9. Trajectory pattern mining abstract workflow.

the collections of large amounts of movement data. This amount of information can be analyzed to extract descriptive and predictive models that can be properly exploited to improve urban life. Here we describe a workflow for discovering patterns and rules from trajectory data, designed using VL4Cloud and executed on DMCF (Altomare et al., 2014). In particular, the trajectory mining workflow described here analyzes the trajectories followed by mobile devices to catch users' mobility behaviors.

Figure 4.9 shows the trajectory pattern mining process, described through an abstract workflow formalism. The original dataset D is a raw trajectory data, populated by the trajectories (represented in the previously described format) of some users collected somehow. In particular, we assume that D is composed of N trajectories, each one represented as a sequence of $H(x, y, t)$ triples. The workflow is composed of four steps, as described in the following:

- *Step 0 – Vertical Data Splitting*: D is partitioned vertically by a Time Stamp Splitter, based on the timestamp value. In other words, the points of the trajectories visited at time stamp $t_i \in T$ will be gathered in the *i*th output dataset. At the end of this step, |T| different datasets are available.
- *Step 1 – Frequent Regions Detection*: This step is aimed at detecting, for each timestamp, the regions that are more densely visited compared to others (thus, of interest for the subsequent analysis). This is done by running H clustering algorithm instances, each one

taking in input a dataset built at the previous step. The final result consists of H clustering models, where the clusters of the ith model represent the dense regions of the ith time stamp (each cluster corresponds to a dense region). The number of detected regions (i.e., number of clusters) may be different for each timestamp.

- *Step 2 – Trajectory Data Synthetization*: This step is aimed at synthesizing the trajectories to build a structured trajectory dataset. This task is performed by running the Trajectory Synthesizer tool, whose goal is to create a dataset where each point of the original trajectories is substituted by the dense region it belongs to (discovered on Step 1). The final dataset, named Trajectory Data in figure, results populated by trajectories between dense regions.
- *Step 3 – Trajectory Pattern Extraction*: A Trajectory Pattern Extraction algorithm on the dense regions trajectory data is executed, to discover trajectory patterns from them. The final mining model is a set of associative rules describing spatio-temporal relations between the movement of the users under investigation.

Figure 4.10 shows a snapshot of the VL4Cloud workflow corresponding to the abstract workflow in Figure 4.9, executed on DMCF. The initial dataset, Trajectory Data, is partitioned into H subsets using the Time Stamp Splitter tool, where H = 128 is equal to the number of timestamps (Step 0). Then, each partition TrajPartition[i], is analyzed by an instance of DBScan and produces a ClusteringModel (Step 1). Each clustering model is a set of clusters/dense regions, for a given timestamp. The TrajectorySynthetizer tool analyzes all models and the initial

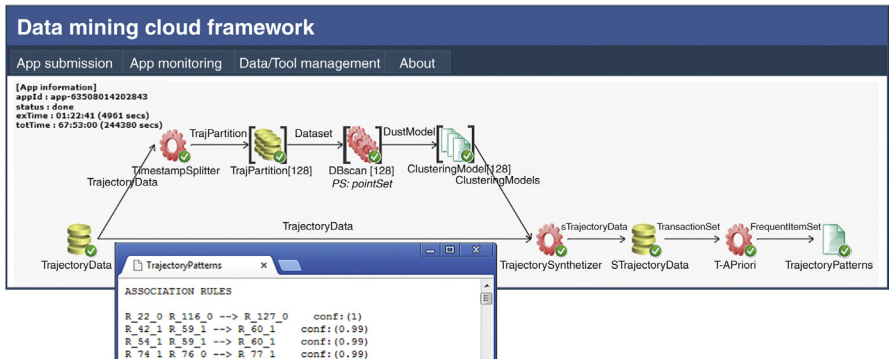


Fig. 4.10. Execution of the VL4Cloud workflow corresponding to the abstract workflow shown in Figure 4.9.

dataset, to generate the Structured Trajectory Data, where each point of the original trajectories is substituted by the dense region it belongs to (Step 2). Finally, T-APriori gets in input this dataset to extract trajectory patterns and, thus, produces the final results (Step 3).

The execution of the workflow using 64 cloud servers showed a significant reduction of turnaround time compared to that achieved by the sequential execution. In particular, the turnaround time passed from about 68 h using a single server, to about 1.4 h using 64 servers.

4.4.2 Ensemble Learning Workflow Using JS4Cloud

This workflow is the implementation of a multiclass cancer classifier based on the analysis of genes, using an ensemble learning approach (Kuncheva, 2004). The input dataset is the Global Cancer Map (GCM)[12], which contains the gene expression profiles of 280 samples representing 14 common human cancer classes. For each sample is reported the status of 16,063 genes and the type of tumor (class label). The GCM dataset is available as a training set containing 144 instances and as a test set containing 46 instances. The goal is to classify an unlabeled dataset (`UnclassGCM`) composed by 20,000 samples, divided in four parts.

The workflow begins by analyzing the training set using `n` instances of the `J48` classification tool and `m` instances of the `JRip` classification tool (Weka's implementation of the Ripper [3] algorithm). The `n J48` instances are obtained by sweeping the `confidence` and the `minNumObj` (minimum number of instances per leaf) parameters, while the `m JRip` instances are obtained by sweeping the `numFolds` (number of folders) and `seed` parameters. The resulting `n + m` classification models (classifiers) are passes as input to `n + m` evaluators, which produce an evaluation of each model against the test set. Then, `k` unclassified datasets are classified using the `n + m` models by `k*(n + m)` predictors. Finally, `k` voters take in input `n + m` model evaluations and the `k*(n + m)` classified datasets, producing `k` classified datasets through weighted voting. Figure 4.11 shows the JS4Cloud code of the workflow.

At the beginning, the training set is specified (line 1). Then, arrays `conf` and `mno` specify, respectively, the `confidence` and `minNumObj` values

[12] http://eps.upo.es/bigs/datasets.html

```
 1: var TrRef = Data.get("GCM-train");
 2: var conf = [0.1, 0.25, 0.5], mno = [2, 5, 10], nfol = [3, 5, 10],
       snum = [1487, 5741, 7699];
 3: var n = conf.length*mno.length, m = nfol.length*snum.length;
 4: var M1Ref = Data.define("Model1", n), M2Ref = Data.define("Model2", m);
 5: for(var i=0; i<conf.length; i++)
 6:    for(var j=0; j<mno.length; j++)
 7:       J48({dataset:TrRef, model:M1Ref[i*mno.length+j], confidence:conf[i],
             minNumObj:mno[j]});
 8: for(var i=0; i<nfol.length; i++)
 9:    for(var j=0; j<snum.length; j++)
10:       JRip({dataset:TrRef, model:M2Ref[i*snum.length+j], numFolds:nfol[i],
             seed:snum[j]});
11: var TeRef = Data.get("GCM-test"), EvM1Ref = Data.define("EvModel1", n),
       EvM2Ref = Data.define("EvModel2", m);
12: for(var i=0; i<n; i++)
13:    Evaluator({dataset:TeRef, model:M1Ref[i], evalModel:EvM1Ref[i]});
14: for(var i=0; i<m; i++)
15:    Evaluator({dataset:TeRef, model:M2Ref[i], evalModel:EvM2Ref[i]});
16: var k = 4;
17: var DRef = Data.get("UnlabGCM", k), CRef = Data.define("ClassGCM",[k,n+m]);
18: for(var i=0; i<k; i++){
19:    for(var j=0; j<n; j++)
20:       Predictor({dataset:DRef[i], model:M1Ref[j], classDataset:CRef[i][j]});
21:    for(var j=0; j<m; j++)
22:       Predictor({dataset:DRef[i], model:M2Ref[j], classDataset:CRef[i][n+j]});
23: }
24: var FRef = Data.define("FinalClassGCM", k), EvMRef = EvM1Ref.concat(EvM2Ref);
25: for(var i=0; i<k; i++)
26:    WeightedVoter({classDataset:CRef[i], evalModel:EvMRef,
          finalClassDataset:FRef[i]});
```

Fig. 4.11. Ensemble learning workflow programmed as a JS4Cloud script.

for `J48`, while arrays `nfol` and `snum` specify, respectively, the `numFolds` and `seed` values for `JRip` (line 2). Given the size of the above arrays, variables `n` and `m` as defined on line 3 are both equal to 9. Afterwards, `n` instances of `J48` and `m` instances of `JRip` are executed, where each instance uses a different combination of its parameters to analyze the training set (lines 5–10). Line 11 specifies the test set, which is used to evaluate the two arrays of models generated by the `n J48` instances and the `m JRip` instances (lines 12–15). Then, `k` unlabeled datasets are specified as input, with `k = 4` (line 17). Each of the `k` input datasets is classified by `n` predictors using the `n` models generated by `J48`, and by `m` predictors using the `m` models generated by `JRip`; therefore, for each of the `k` input datasets, `n + m` classified datasets are generated (lines 18–23). As a final step, `k` weighted voters are executed; the *i*th voter receives the `n + m` classified datasets generated from the *i*th input and the `n + m` models, and returns the final classified dataset for the *i*th input (lines 25–26). In general, the

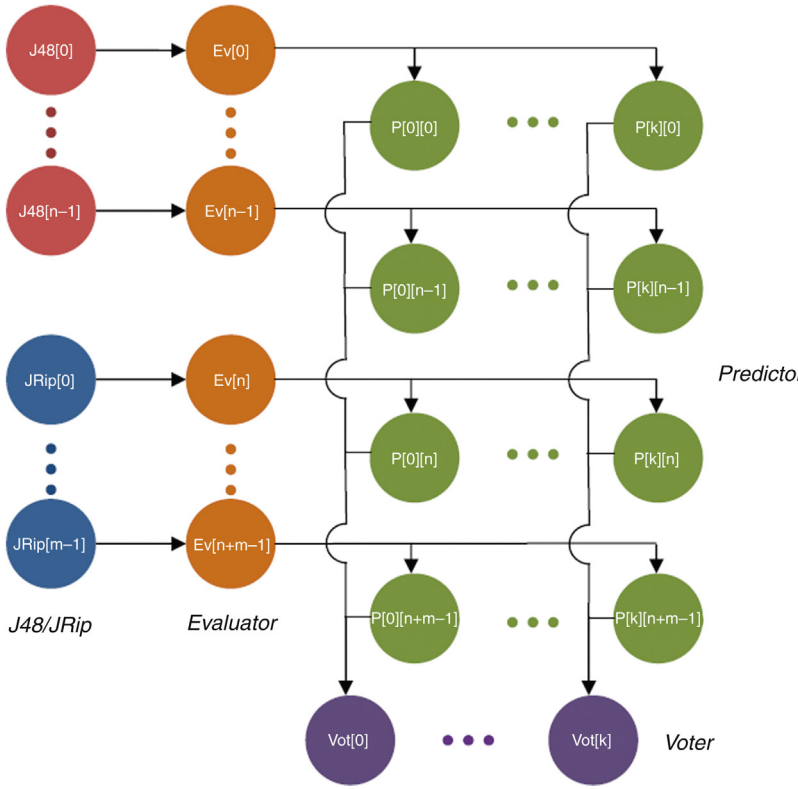


Fig. 4.12. Task dependency graph associated with the JS4Cloud script in Figure 4.11.

workflow is composed of `k + (k + 2)(n + m)` tasks, which are related to each other as specified by the dependency graph shown in Figure 4.12.

In the specific example (with `n` = 9, `m` = 9, `k` = 4) the number of tasks is 112. As for the trajectory mining workflow introduced earlier, the cloud allowed us to speed up significantly the workflow execution. In fact, the turnaround time passed from about 162 min using a single server, to about 11 min using 19 servers, which results in a speedup of about 14.7.

4.4.3 Parallel Classification Using MapReduce in DMCF

In this section we describe a DMCF data classification workflow including MapReduce computations. Through this example, we show how the MapReduce paradigm can be integrated into VL4Cloud workflows, and how it can be used to exploit the inherent parallelism of the

application. The application deals with a significant economic problem coupled with the flight delay prediction. Every year approximately 20% of airline flights are delayed or canceled mainly due to bad weather, carrier equipment, or technical airport problems. These delays result in significant cost to both airlines and passengers (Ball et al., 2010).

The goal of this application is to implement a predictor of the arrival delay of scheduled flights due to weather conditions. The predicted arrival delay takes into consideration both flight information (origin airport, destination airport, scheduled departure time, scheduled arrival time) and weather forecast at origin and destination airports. In particular, we consider the closest weather observation at origin and destination airports based on scheduled flight departure and arrival time. If the predicted arrival delay of a scheduled flight is less than a given threshold, it is classified as an on-time flight; otherwise, it is classified as a delayed flight.

Two open datasets of airline flights and weather observations have been collected, and exploratory data analysis has been performed to discover initial insights, evaluate the quality of data, and identify potentially interesting subsets. The first dataset is the *Airline On-Time Performance* (*AOTP*) provided by RITA - Bureau of Transportation Statistics.[13] The AOTP dataset contains data for domestic US flights by major air carriers, providing for each flight detailed information such as origin and destination airports, scheduled and actual departure and arrival times, air time, and nonstop distance. The second is the *Quality Controlled Local Climatological Data* (*QCLCD*) dataset available from the National Climatic Data Center.[14] The dataset contains hourly weather observations from about 1,600 US stations. Each weather observation includes data about temperature, humidity, wind direction and speed, barometric pressure, sky condition, visibility and weather phenomena descriptor.

For data classification, a MapReduce version of the Random Forest (RF) algorithm has been used. RF is an ensemble learning method for classification [13]. It creates a collection of different decision trees called *forest*. Once forest trees are created, the classification of an unlabeled

[13]http://www.transtats.bts.gov
[14]http://cdo.ncdc.noaa.gov/qclcd/QCLCD

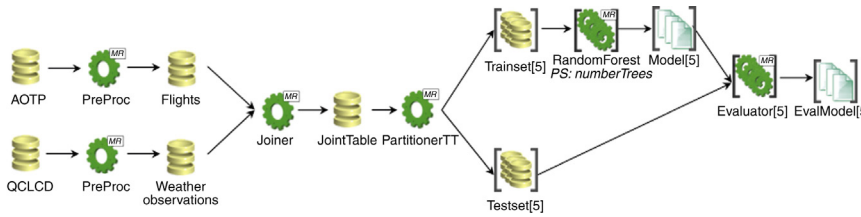


Fig. 4.13. Flight delay analysis workflow with MapReduce computations in DMCF.

tuple is performed by aggregating the predictions of the different trees through majority voting.

Using DMCF, we created a VL4Cloud workflow for the whole data analysis process (see Figure 4.13). The workflow begins by preprocessing the AOTP and the QCLCD datasets using two instances of a `PreProc` Tool. These steps allow looking for possible wrong data, treating missing values, and filtering out diverted and cancelled flights and weather observations not related to airport locations. Then, a `Joiner` Tool executes a relational join between `Flights` and `Weather Observations` data in parallel using a MapReduce algorithm. The result is a `JointTable`. Then, a `PartionerTT` Tool creates five pairs <`Trainset`, `Testset`> using different delay threshold values. The five instances of training set and test set are represented in the workflow as two data array nodes, labelled as `Trainset[5]` and `Testset[5]`.

Then, five instances of the `RandomForest` Tool analyze in parallel the five instances of `Trainset` to generate five models (`Model[5]`). For each model, an instance of the `Evaluator` Tool generates the confusion matrix (`EvalModel`), which is a commonly used method to measure the quality of classification. Starting from the set of confusion matrices obtained, these tools calculate some metrics, e.g. accuracy, precision, recall, which can be used to select the best model.

To run the workflow, we deployed a Hadoop cluster composed of 1 head node and 8 worker nodes, over the cloud servers used by the DMCF environment. With this setting, the turnaround time decreased from about 7 h using 2 workers, to about 1.7 h using 8 workers, with a speedup that is very close to linear values. In this example, scalability is obtained exploiting the parallelism offered both by MapReduce Tools

and by the VL4Cloud workflow language. In the first case, each MapReduce Tool is executed in parallel exploiting the cluster resources. The level of parallelism depends on the number of map and reduce tasks and on the resources available in the cluster. In the second case, VL4Cloud allows creating parallel paths and array of tools that can be executed concurrently. In this case, the parallelism level depends on the dependencies among tasks and on the resources available in the cluster.

4.4.4 Parallel Classification Using Swift

We describe here how the JS4Cloud workflow shown in Figure 4.8 can be coded using the Swift parallel scripting language. This application classifies a dataset by using *n* instances of the J48 classification algorithm that analyzes in parallel *n* partitions of the training set and generate *n* classification models. By using the *n* generated models and the test set, *n* predictor tasks generate in parallel *n* classified datasets. In the final step of the workflow, a voter generates the final classification by assigning a class to each data item. Choosing the class predicted by the majority of the models does this. For the reader's convenience, we report in Figure 4.14, the JS4Cloud code.

```
1  var n = 16;
2  var DRef = Data.get("Dataset"), TrRef = Data.define("TrainSet");
3  var TeRef = Data.define("TestSet");
4
5  PartitionerTT({dataset:DRef, percTrain:0.7, trainSet:TrRef,  testSet:TeRef});
6
7  var PRef = Data.define("TrainsetPart", n);
8
9  Partitioner({dataset:TrRef, datasetPart:PRef});
10
11 var MRef = Data.define("Model", n);
12
13 for(var i=0; i<n; i++)
14    J48({dataset:PRef[i], model:MRef[i], confidence:0.1});
15
16 var CRef = Data.define("ClassTestSet", n);
17
18 for(var i=0; i<n; i++)
19    Predictor({dataset:TeRef, model:MRef[i], classDataset:CRef[i]});
20
21 var FRef = Data.define("FinalClassTestSet");
22
23 Voter({classDataset:CRef, finalClassDataset:FRef});
```

Fig. 4.14. JS4Cloud code of a parallel classification example.

```
1  type datafile {}
2
3  app (datafile output) J48 (datafile partition, int conf)
4  {
5     java.exe -cp weka.jar weka.classifiers.trees.j48 -t @partition -C @conf -d @output;
6  }
7  app (datafile output) Predictor (datafile test, datafile model)
8  {...}
9  app (datafile output) Voter (datafile[] classes)
10 {...}
11
12  int n = 16;
13  int perc = 70;
14  int confidence = 0.1;
15  datafile DRef <"Dataset">;
16  datafile TTRef[] <ext; exec = PartitionerTT, percTrain = perc; source = DRef>;
17  datafile PRef[]  <ext; exec = Partitioner; source = TTRef[0], n>;
18  datafile MRef[] <filesys_mapper; Prefix = "Model">;
19
20  foreach i in PRef {
21    MRef[i] = J48(PRef[i], confidence);
23  }
24  datafile CRef[] <filesys_mapper; Prefix = "ClassTestSet">;
25
26  foreach i in MRef {
27    CRef[i] = Predictor(TTRef[1], MRef[i]);
28  }
29  datafile FRef <"FinalClassTestSet">;
30  FRef = Voter(CRef);
```

Fig. 4.15. Swift code of a parallel classification example.

Swift is a scripting language that executes external programs in parallel. The Swift code in Figure 4.15 includes the definition of *mapped types* needed to declare the data elements that refer to files external to the Swift script (Wilde et al., 2011). These files can then be read and written by application programs called by Swift. Line 1 declares `datafile` type used for datsets. Lines 3–10 include the `app` declarations that in Swift describe how an external application program is invoked. Here are declared three application programs: the `J48` classifier, the `Predictor` and the `Voter`. These declarations specify the application parameters and the program invocation string. Lines 12–14 declare program variables, line 15 declares the input dataset (`Dataset`) and lines 16–18, 24, and 29 declare the intermediate and output datasets. In particular, those declarations use mapping primitives (*mappers*) that make a given variable name refer to a filename. A mapper associated with a structured Swift variable can represent a large, structured dataset. Here are used built-in mappers

(see lines 18 and 24) and external mappers (see lines 16–17) that include the reference to the executable of eternal programs that generate the datasets (e.g., `PartitionerTT` and `Partitioner`). Parallelism in the application is exploited during the execution of the `foreach` loop that, as the `for` loop in JS4Cloud, is executed in parallel on each element of the dataset array (lines 20–23 and 26–28). In fact, similarly to the JS4Cloud `for` statement, a `foreach` statement in Swift applies its body of statements to elements of an array in a fully concurrent, pipelined manner, as they are set to a value.

As we can see from the manner in which this application example is programmed, although syntax is different is several aspects, the programming style of the Swift language is similar to the J4Cloud programming style. Also the data driven approach and the exploitation of parallelism is similar in the two languages.

4.5 SUMMARY

In the latest years new commercial and research-oriented software tools and frameworks have been designed, implemented, and used for executing data analysis applications on clouds systems. This chapter focused on a selection of those frameworks and discussed their main features, architectures, and programming approaches. Among those systems we mention here Pegasus, Hunk, BigML, Swift, DMCF, Mahout, and Spark. Since a deep description of how all the systems can be used for implementing data analysis applications could not be accommodated in this chapter, we focused on one of them, the DMCF environment, and discussed its cloud-based software architecture, how it implements scalability of data analysis applications, and presented solutions that have been adopted for its implementation on a public cloud.

The DMCF system offers two high-level programming languages for developing data analysis workflows that can be executed in parallel on a cloud. *VL4Cloud* (Visual Language for Cloud), a visual language that lets users develop applications by programming the workflow components graphically, and *JS4Cloud* (JavaScript for Cloud), a scripting language based on JavaScript designed for programming data analysis workflows. Both languages have been presented and compared. In particular, language patterns and constructs for the exploitation of parallelism on data

analysis applications have been described. A few programming examples using the key programming abstractions (*Data* elements to denote files and storage elements, and *Tool* elements to denote algorithms, software tools and applications executed on *Data* elements) have been illustrated and discussed.

The final section of the chapter has been devoted to four significant case studies of data analysis. Trajectory mining, ensemble learning, and parallel classification based on a MapReduce-workflow hybrid approach have been discussed. Finally, a comparison between JS4Cloud and Swift is done through the programming of a parallel classification workflow. The contents of this chapter can be a useful contribution for learning how scalable solutions for data analysis can be designed and used. Through the analysis and presentation of some of the most advanced tools in this area, it is possible to understand how cloud solutions can effectively support the development of complex and compute intensive big data analysis.

REFERENCES

Altomare, A., Cesario, E., Comito, C., Marozzo, F., Talia, D., 2014. Trajectory pattern mining over a cloud-based framework for urban computing. Proceedings of the Sixteenth International Conference on High Performance Computing and Communications (HPCC 2014), Paris, France, pp. 367–374, IEEE.

Anil, R., Owen, S., Dunning, T., Friedman, E., 2012. Mahout in action. Manning, Shelter Island, NY.

Ball, M., Barnhart, C., Dresner, M., Hansen, M., Neels, K., Odoni, A., Peterson, E., Sherry, L., Trani, A.A., Zou, B., 2010. Total delay impact study: a comprehensive assessment of the costs and impacts of flight delay in the United States. NEXTOR Final Report prepared for the Federal Aviation Administration.

Bharathi, S., Chervenak, A., Deelman, E., Mehta, G., Su, M.-H., Vahi. K., 2008. Characterization of scientific workflows. In: Workflows in Support of Large-Scale Science. WORKS 2008. Third Workshop, pp. 1–10.

Deelman, E., Gannon, D., Shields, M., Taylor, I., 2009. Workflows and e-science: an overview of workflow system features and capabilities. Future Gener. Comput. Syst. 25 (5), 528–540.

Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P.J., Mayani, R., Chen, W., Ferreira da Silva, R., Livny, M., Wenger, K., 2015. Pegasus: a workflow management system for science automation. Future Gener. Comput. Syst. 46, 17–35.

Giardine, B., et al., 2005. Galaxy: a platform for interactive large-scale genome analysis. Genome Res. 15, 1451–1455.

Gu, Yunhong, Grossman, Robert L., 2009. Sector and sphere: the design and implementation of a high-performance data cloud. Philos. Trans. A. Math. Phys. Eng. Sci. 367 (1897), 2429–2445.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., 2009. The weka data mining software: an update. SIGKDD Explor. Newsl. 11 (1), 10–18.

Juve, G., Deelman, E., Vahi, K., Mehta, G., Berriman, B., Berman, B.P., Maechling, P., 2010. Data Sharing Options for Scientific Workflows on Amazon EC2. Int. Conference on High Performance Computing, Networking, Storage and Analysis (SC 2010).

Kranjc, J., Podpecan,V., Lavrac, N., 2012. ClowdFlows: a cloud based scientific workflow platform In: Machine Learning and Knowledge Discovery in Databases, ser. Lecture Notes in Computer Science, vol. 7524. Springer, Heidelberg, Germany, pp. 816–819.

Kuncheva, L.I., 2004. Combining Pattern Classifiers: Methods and Algorithms. Wiley-Interscience. Chichester.

Marozzo, F., Talia, D., Trunfio, P., 2015. JS4Cloud: script-based workflow programming for scalable data analysis on cloud platforms. Concurrency and Computation: Practice and Experience, Wiley InterScience. Chichester.

Nagavaram, A., Agrawal, G., Freitas, M., Mehta, G., Mayani, R., Deelman, E., 2011. A cloud-based dynamic workflow for mass spectrometry data analysis. In: Proceedings of the Seventh IEEE International Conference on e-Science (e-Science 2011).

Quinlan, J.R., 1993. C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers Inc. San Francisco, CA.

Shahrivari, S., 2014. Beyond batch processing: towards real-time and streaming big data. Computers 3 (4), 117–129.

Wilde, M., Hategan, M., Wozniak, J.M., Clifford, B., Katz, D.S., Foster, I., 2011. Swift: a language for distributed parallel scripting. Parallel Comput. 37 (9), 633–652.

Wozniak, J.M., Wilde, M., Foster, I.T., 2014. Language features for scalable distributed-memory dataflow computing. In: Proceedings of Data-flow Execution Models for Extreme-scale Computing at PACT.

Xin, R.S., Rosen, J., Zaharia, M., Franklin, M.J., Shenker, S., Stoica. I., 2013. Shark: SQL and rich analytics at scale. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13). New York, USA.

Zhou, Z.-H., Li, M., 2010. Semi-supervised learning by disagreement. Knowl. Inf. Syst. 24 (3), 415–439.