CHAPTER 3

Models and Techniques for Cloud-Based Data Analysis

This chapter discusses the main models and techniques used for designing cloud-based data analysis applications. The models presented here are based on MapReduce, workflows, and NoSQL database management systems. In the next sections, we explain how these three main approaches offer scalability for mining Big Data repositories on clouds. Section 3.1 introduces the MapReduce model and how it can be used to implement scalable data analysis algorithms and applications. Section 3.2 discusses the workflow systems, presents some workflow management systems (WMSs) implemented on cloud architectures and discusses their main features to implement data analysis applications. Finally, Section 3.3 describes NoSQL database systems that were recently developed to efficiently manage large volumes of data. In several application cases NoSQL databases are more scalable and provide higher performance than relational databases. Here we describe some representative NoSQL systems, and discuss use cases for NoSQL databases with a focus on data analytics.

3.1 MapReduce FOR DATA ANALYSIS

MapReduce is a system and method for efficient large-scale data processing proposed by Google in 2004 (Dean and Ghemawat, 2004) to cope with the challenge of processing very large input data generated by Internet-based applications. Since its introduction, MapReduce has proven to be applicable in a wide range of domains, including machine learning and data mining, social data analysis, financial analysis, scientific simulation, image retrieval and processing, blog crawling, machine translation, language modeling, and bioinformatics. Today, MapReduce is widely recognized as one of the most important programming models for cloud computing environments, as it is supported by Google and other leading cloud providers such as Amazon, with its Elastic MapReduce service,[1] and Microsoft, with its HDInsight,[2] and used on top of private cloud infrastructures such as OpenNebula, with its Sahara service.[3]

The MapReduce abstraction is inspired by the *map* and *reduce* primitives designed in Lisp and other functional languages. A user defines a MapReduce application in terms of a *map* function that processes a (*key, value*) pair, to generate a list of intermediate (*key, value*) pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Current MapReduce implementations, like Hadoop (White, 2009), are based on a master–slave architecture. A user node submits a job to a master node, which selects idle workers and assigns a map or reduce task to each one. When all the tasks are complete, the master node returns the result to the user node.

For years, grid and distributed computing systems have been widely used for data processing. These systems work well with compute-intensive jobs, but require a lot of network bandwidth to handle huge amounts of distributed data. To reduce the network bandwidth bottleneck, which often affects distributed data analysis systems, MapReduce implements a data locality feature, so that the computation tasks are located near the input data (e.g., same node, same rack) to optimize performance and save energy (Zhenhua Guo et al., 2012). In contrast to RDBMS, that is ideal for storing and processing structured data, MapReduce can be used to process semistructured or unstructured data in parallel, since data is evaluated at processing time. Moreover, MapReduce is designed to process very large amounts of data using hundreds or thousands of machines in distributed/parallel environments, so the model must tolerate machine failures. The failure of a worker is managed by re-executing its task on another worker. An optimization related to failure handling provided by MapReduce frameworks is the scheduling of redundant execution of tasks near the end of the job for reducing its completion time in case of machine failures and data loss.

Thanks to the features described above, MapReduce is widely used to implement scalable data analysis algorithms and applications executed on multiple machines to efficiently analyze big amounts of data.

[1]http://aws.amazon.com/elasticmapreduce/

[2]http://azure.microsoft.com/services/hdinsight/

[3]http://wiki.openstack.org/wiki/Sahara

3.1.1 MapReduce Paradigm

This section briefly describes various operations that are performed by a generic application to transform input data into output data according to the MapReduce model.

Users must define a *map* and a *reduce* function (Dean and Ghemawat, 2008). The *map* function processes a (*key, value*) pair and returns a list of intermediate (*key, value*) pairs:

$$map(k1, v1) \rightarrow \text{list}(k2, v2)$$

The *reduce* function merges all intermediate values having the same intermediate key:

$$reduce(k2, \text{list}(v2)) \rightarrow \text{list}(v3)$$

As an example, let us consider the creation of an inverted index for a large set of Web documents. In its basic form, an inverted index contains a set of words (index terms), and for each word it specifies the IDs of all the documents that contain that word. Using a MapReduce approach, the *map* function parses each document and emits a sequence of (word, documentID) pairs. The *reduce* function takes all pairs for a given word, sorts the corresponding document IDs, and emits a (word, list(documentID)) pair. The set of all output pairs generated by the reduce function forms the inverted index for the input documents.

In general, the whole transformation process performed in a MapReduce application can be described through the following steps (Figure 3.1):

1. A master process receives a job descriptor, which specifies the MapReduce job to be executed. The job descriptor contains, among other information, the location of the input data, which may be accessed using a distributed file system.
2. According to the job descriptor, the master starts a number of mapper and reducer processes on different machines. At the same time, it starts a process that reads the input data from its location, partitions that data into a set of splits, and distributes those splits into various mappers.
3. After receiving its data partition, each mapper process executes the *map* function (provided as part of the job descriptor) to generate a

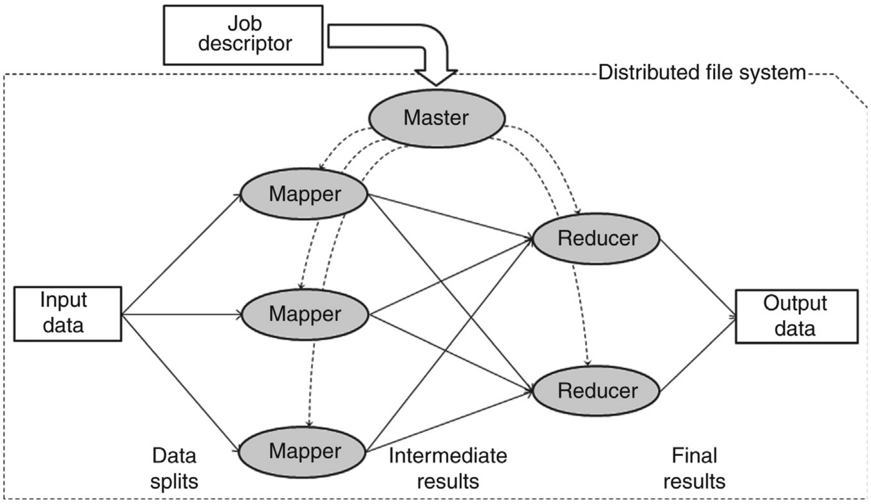

Fig. 3.1. Execution phases of a generic MapReduce application.

list of intermediate key/value pairs. Then these pairs are grouped on the basis of their keys.

4. All pairs with the same keys are assigned to the same reducer process. Hence, each reducer process executes the *reduce* function (defined by the job descriptor), which merges all the values associated with the same key to generate a possibly smaller set of values.
5. Then results generated by each reducer process is collected and delivered to a location specified by the job descriptor, so as to form the final output data.

Distributed file systems are the most popular solutions for accessing input/output data in MapReduce systems, particularly in standard computing environments like a data center or a cluster of computers.

To define complex applications that cannot be coded with a single MapReduce job, users need to compose chains or, in a more general way, workflows of MapReduce jobs. Chains can be easily implemented with the output of a job that goes to a distributed file system and is used as an input for the next job. Figure 3.2 shows an example of MapReduce chain, in which some data sources are analyzed sequentially by three MapReduce jobs, to produce the final output.

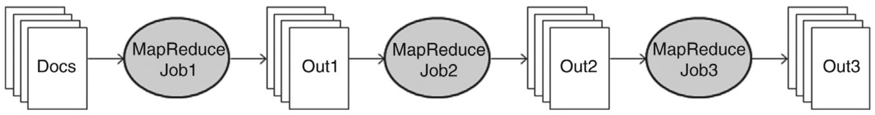

Fig. 3.2. Example of MapReduce chain.

3.1.2 MapReduce Frameworks

Hadoop is the best-known MapReduce implementation and it is commonly used to develop parallel applications analyzing big amounts of data. Hadoop can be adopted for developing distributed and parallel applications using many programming languages (e.g., Java, Ruby, Python, C++). It relieves developers from having to deal with classical distributed computing issues, such as load balancing, fault tolerance, data locality, and network bandwidth saving. The Hadoop project is not only about the MapReduce programming model (Hadoop MapReduce module), but also includes other modules such as:

- *Hadoop Distributed File System* (*HDFS*): a distributed file system providing fault tolerance with automatic recovery, portability across heterogeneous commodity hardware and operating systems, high throughput access and data reliability.
- *Hadoop YARN*: a framework for cluster resource management and job scheduling.
- *Hadoop Common*: common utilities that support the other Hadoop modules.

With the introduction of YARN in 2013, Hadoop turns from a batch processing solution into a platform to run a large variety of data applications, such as streaming, in-memory, and graphs analysis. As a result, Hadoop became a reference for several other frameworks, such as Giraph[4] for graph analysis; Storm[5] for streaming data analysis; Hive[6], which is a data warehouse software for querying and managing large datasets; Pig[7], which is as a dataflow language for exploring large datasets; Tez[8] for executing complex directed-acyclic graphs of data

[4]http://giraph.apache.org/
[5]http://storm.apache.org
[6]http://hive.apache.org
[7]http://pig.apache.org
[8]http://tez.apache.org/

processing tasks; Oozie[9], which is a workflow scheduler system for managing Hadoop jobs; Spark[10], which is a cluster computing framework for in-memory machine learning and data analysis. In contrast to Hadoop's two-stage MapReduce paradigm, in which intermediate data are always stored in distributed file systems, Spark stores data in cluster's memory and queries it repeatedly so as to obtain better performance for some class of applications (e.g., iterative machine learning algorithms) (Xin et al., 2013).

Besides Hadoop and its ecosystem, several other MapReduce implementations have been implemented within other systems, including GridGain,[11] Skynet,[12] MapSharp,[13] and Twister (Ekanayake et al., 2010). One of the most popular alternatives to Hadoop is Disco,[14] which is a lightweight, open-source framework for distributed computing. The Disco core is written in Erlang, a functional language designed to build fault-tolerant distributed applications. Disco has been used for a variety of purposes, such as log analysis, text indexing, probabilistic modeling, and data mining.

Some other frameworks focused toward adapting the MapReduce model to specific computing environments. Among them, Phoenix (Ranger et al., 2007) is an implementation of MapReduce for shared-memory systems that includes a programming API and a runtime system. It uses threads to spawn parallel map or reduce tasks and shared-memory buffers to facilitate communication without excessive data copying. Overall, Phoenix proves that MapReduce is a useful programming and concurrency management approach also for multicore and multiprocessor systems.

MOON (Lin et al., 2010) is a system designed to support MapReduce jobs in opportunistic environments. It extends Hadoop with adaptive task and data scheduling algorithms to offer reliable MapReduce

[9]http://oozie.apache.org/
[10]http://spark.apache.org
[11]http://www.gridgain.com/
[12]http://github.com/skynetservices/skynet
[13]http://mapsharp.codeplex.com
[14]http://discoproject.org/

services on a hybrid resource architecture, where volunteer computing systems are supplemented by a small set of dedicated nodes. The adaptive task and data scheduling algorithms in MOON distinguish different types of MapReduce data and node outages, in order to place tasks and data on volatile and dedicated nodes.

Tang et al. (2010) designed a system to support MapReduce applications in desktop grids. The proposed system exploits the BitDew middleware (Fedak et al., 2009), which is a programmable environment for automatic and transparent data management on desktop grids. BitDew relies on a specific set of metadata to drive key data management operations, namely life cycle, distribution, placement, replication, and fault tolerance with a high level of abstraction.

MISCO (Dou et al., 2010) is a framework for supporting MapReduce applications on mobile systems. Although Misco follows the general design of MapReduce, it differs in two main aspects: task assignment and data transfer. The first aspect is managed with a polling strategy. For data transfer, instead of a distributed file system that is not practical in a mobile scenario, Misco uses HTTP to communicate requests, task information, and transfer data.

Finally, P2P-MapReduce (Marozzo et al., 2012b) is a framework that exploits a peer-to-peer model to manage node churn, master failures, and job recovery in a decentralized but effective way, so as to provide a more reliable MapReduce middleware, which can be effectively exploited in dynamic cloud infrastructures. The P2P-MapReduce framework does not suffer from job failures, even in the presence of very high churn rates, thus enabling the execution of reliable MapReduce applications in dynamic cloud infrastructures.

3.1.3 MapReduce Algorithms and Applications

In the last few years, all major data mining algorithms have been re-implemented in MapReduce such as K-means (Ekanayake et al., 2008), Apriori (Lin et al., 2012), C4.5 (Gongqing et al., 2009), and support vector machines (Sun and Fox, 2012). Chu et al. (2007) demonstrated that MapReduce shows a linear speedup with an increasing number of processors in a variety of learning algorithms such as Naive Bayes, neural networks, and Expectation–Maximization probabilistic clustering.

Ricardo project (Das et al., 2010) is a platform that integrates R[15] statistical tools and Hadoop to support parallel data analysis.

Mahout[16] is a machine learning framework that provides scalable machine learning libraries on top of Hadoop. It supplies various algorithms for clustering, classification, and collaborative filtering, which can be run in parallel on a Hadoop cluster. MLlib[17] is a scalable machine learning library on top of Spark. It implements many common machine learning and statistical algorithms to analyze large-scale data hosted in the memory.

As mentioned earlier in this section, MapReduce has proven to be applicable to a wide range of domains, including social data analysis (Tang et al., 2009), financial analysis (Coleman et al., 2012), image retrieval and processing (Cary et al., 2009), trajectory analysis (Ma et al., 2009) and bioinformatics (Ekanayake et al., 2008).

Tang et al. (2009) analyzed a large social network to find out how nodes (users and entities) are influenced by others for various reasons. To address this issue the authors proposed a model, called Topical Affinity Propagation, to describe the topic-level social influence on large networks. Their learning task was performed in a distributed system using MapReduce.

Coleman et al. (2012) showed how numerically intensive tasks for pricing, risk analysis, forecasting, and automated trading can be efficiently dealt through MapReduce algorithms. Cary et al. (2009) showed how MapReduce can be used to analyze spatial data, that is, raster data (satellite/aerial digital images) or vector data (points, lines, polygons), while Ma et al. (2009) show the use of MapReduce for query processing over trajectory data. The use of MapReduce for data intensive scientific analysis and bioinformatics is deeply analyzed in Ekanayake et al. (2008). The authors showed that most scientific data analyses (e.g., SMPD algorithm) can benefit from MapReduce to achieve speedup and scalability.

[15]http://www.r-project.org
[16]http://mahout.apache.org
[17]http://spark.apache.org/mllib/

3.2 DATA ANALYSIS WORKFLOWS

A workflow consists of a series of activities, events, or tasks that must be performed to accomplish a goal and/or obtain a result. For example, a data analysis workflow can be designed as a sequence of preprocessing, analysis, postprocessing, and interpretation steps. At a practical level, a workflow can be implemented as a computer program, can be expressed in a programming language or paradigm that allows expressing the basic workflow steps, and includes mechanisms to orchestrate them.

Workflows have emerged as an effective paradigm to address the complexity of scientific and business applications. The wide availability of high-performance computing systems, grids and clouds, allowed scientists and engineers to implement more complex applications to access and process large data repositories and run scientific experiments *in silico* on distributed computing platforms. Most of these applications are designed as workflows that include data analysis, scientific computation methods, and complex simulation techniques.

The design and execution of many scientific applications require tools and high-level mechanisms. Simple and complex workflows are often used to reach this goal. For this reason, in the past years, many efforts have been devoted towards the development of distributed WMSs for scientific applications. Workflows provide a declarative way of specifying the high-level logic of an application, hiding the low-level details that are not fundamental for application design. They are also able to integrate existing software modules, datasets, and services in complex compositions that implement scientific discovery processes.

According to the definition of the Workflow Management Coalition (WFMC, 1999; Hollingsworth, 1995), a workflow is "the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules." The same definition can be used for scientific processes composed of several processing steps that are connected together to express data and/or control dependencies (Liu et al., 2004). The term process here indicates a set of tasks linked together with the goal of creating a product, calculating a result, or providing a service. Hence, each task (or activity) represents a piece of work that forms one logical step of the overall process (Georgakopoulos et al., 1995).

A workflow is a well defined, possibly a repeatable pattern, or systematic organization of activities designed to achieve a certain transformation of data. Workflows, as practiced in scientific computing, derive from several significant precedent programming models that are worth noting because they have greatly influenced the way we think about workflows in scientific applications. We can call them the dataflow models, in which data is streamed from one actor to another. Although the pure dataflow concept is extremely elegant, it is very hard to make it work in practice because distributing control in a parallel or distributed system can create applications that are not very fault tolerant.

Consequently, many workflow systems use a dataflow model for expressing computation as it can have an implicit centralized control program that sequences and schedules each step. An important benefit of workflows is that, once defined, they can be stored and retrieved for modifications and/or re-execution. This allows users to define typical patterns and reuse them in different scenarios (Bowers et al., 2006). The definition, creation, and execution of workflows are supported by a WMS. A key function of the WMS during a workflow's execution (or enactment) is coordinating the operations of the individual activities, which constitute the workflow.

Through the integrated use of computer science methods and scientific discovery processes, science progressed into a new era where scientific methods changed significantly by the use of computational methods, and new data analysis strategies created the e-science paradigm (Bell et al., 2009). For example, the Pan-STARRS astronomical survey (Deelman et al., 2009) uses workflows from Microsoft Trident Scientific Workflow Workbench to load and validate telescope detections running at about 30 TB/year. Similarly, the USC Epigenome Center is currently using the Pegasus workflow system to exploit the Illumina Genetic Analyzer (GA) system to generate high throughput DNA sequence data (up to eight billion nucleotides per week) to map the epigenetic state of human cells on a genome-wide scale. In this scenario, scientific workflows demonstrate their effectiveness for programming at high-level complex scientific applications that in general run on supercomputers or on distributed computing infrastructures such as grids, peer-to-peer systems and clouds (Sonntag et al., 2010; Yu and Buyya, 2005; Al-Shakarchi et al., 2007).

3.2.1 Workflow Programming

A main issue in WMSs is the programming structure provided to the developers who implement a scientific application. While some systems provide a textual programming interface, others are based on a visual programming interface. These two different interfaces imply different programming approaches.

Often a scientific workflow is programmed as a graph of several data and processing nodes that include predefined procedures written in programming languages such as Java, C++, Perl, Python, etc. According to this approach, a scientific workflow is a methodology to orchestrate predefined programs that run single tasks, but are composed together to represent a complex application that generally needs large resources to run and may take a long running time to complete. For an introduction, refer to the work by Taylor et al. (2007), which provides a taxonomy of the main features of scientific workflows. In fact, it is not rare to have scientific workflows, for example in the astronomy domain or in bioinformatics (Cannataro et al., 2004), which take several days or weeks to complete their execution.

Workflow tasks can be composed together following a number of different patterns, whose variety helps designers addressing the needs of a wide range of application scenarios. A comprehensive collection of workflow patterns, focusing on the description of control flow dependencies among tasks, has been described in (van der Aalst et al., 2003). The most common programming structure used in WMSs is the directed acyclic graph (DAG) (Figure 3.3) or its extension that includes loops, which is the directed cyclic graph (DCG).

Of the many possible ways to distinguish workflow computations, one is to consider a simple complexity scale. At the most basic level one

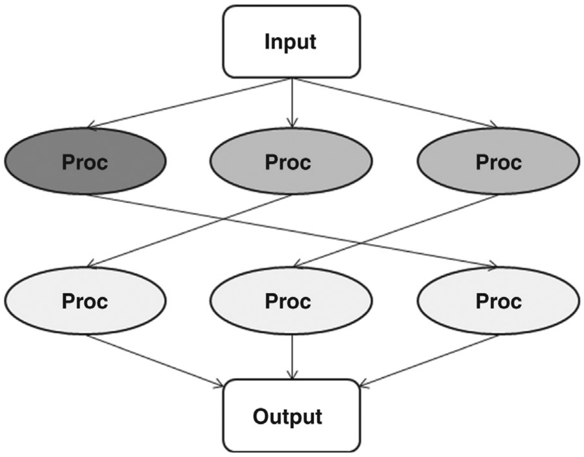

Fig. 3.3. A simple DAG including data and control nodes.

can consider linear workflows, in which a sequence of tasks must be performed in a specified linear order. The first task transforms an initial data object into new data object, which is used as input in the next data-transformation task. The execution of the entire chain of tasks may take a few minutes, or it may take days, depending on the computational grain of each single task in the pipeline. When the execution time is short, the most common workflow-programming tool is a simple script written, for instance, in Python, Perl, or Matlab. The case of longer running workflows often requires more sophisticated tools or programming languages.

At the next level of complexity, one can consider workflows that can be represented by a DAG, where nodes of the graph represent tasks to be performed and edges represent dependencies between tasks. Two main types of dependencies can be considered: data dependencies (where the output of a task is used as input by the next tasks) and control dependencies (where before to start one or a set of tasks some tasks must be completed). This is harder to represent with a scripting language without a substantial additional framework behind it, but it is not at all difficult to represent with a tool like Ant. It is the foundation of the Directed Acyclic Graph Manager (DAGMan), a meta-scheduler of Condor (Litzkow et al., 1988), which is a specialized workload management system developed by the University of Wisconsin, and the execution engine of Pegasus. Applications that follow this pattern can be characterized by workflows, in which some tasks depend upon the completion of several other tasks that may be executed concurrently (Deelman et al., 2009).

A DAG can be used to represent a set of programs where the input, output, or execution of one or more programs is dependent on one or more other programs. According to this model, also in DAGMan programs are nodes in the graph, the edges (arcs) identify the program dependencies. For task execution, Condor finds computers for the execution of programs, but it does not schedule programs based on dependencies (Couvares et al., 2007). DAGMan submits tasks to Condor in an order represented by a DAG and processes the task results. An input file defined prior to submission describes the workflow as a DAG, and Condor uses a description file for each program in the DAG. DAGMan is responsible for scheduling, recovery, and reporting for the set of programs submitted to Condor.

The next level of workflow complexity can be characterized by cyclic graphs, where cycles represent some form of implicit or explicit loop or iteration control mechanisms. In this case, the workflow graph often describes a network of tasks where the nodes are either services, some form of software component instances, or represent more abstract control objects. The graph edges represent messages, data streams, or pipes that exchange work or information among services and components.

The highest level of workflow structure is one in which a compact graph model is inappropriate. This is the case when the graph is simply too large and complex to be effectively designed. However, some tools allow one to turn a graph into a new first-class component or service, which can then be included as a node in another graph (a workflow of workflows or hierarchical workflow). This technique allows graphs of arbitrary complexity to be constructed. This nested-workflow approach requires the use of more sophisticated tools both at the composition stage and at the execution stage. In particular, this approach makes the task-to-resource mapping and the task scheduling harder during the workflow execution done on a large scale distributed or parallel infrastructures such as HPC systems or cloud computing platforms (Mirto et al., 2010).

In the case of workflow enactment, two issues must be taken into account: efficiency and robustness. In terms of efficiency, the critical issue is the ability to quickly bind workflow tasks to the appropriate computing resources. It heavily depends on the mechanisms used to assign data to tasks, to move data between tasks that need them at various stages of the enactment. For instance, considering a service-oriented scenario, we cannot assume that Web service protocols like SOAP should be used in anything other than task/service control and simple message delivery (Zhang et al., 2007; Lin et al., 2008). Complex and/or data movement between components of the workflow must be either via an interaction with a data movement service, or through specialized binary-level data channel running directly between the tasks involved.

Robustness is another issue, making the reasonable assumption that some parts of a workflow may fail. It is essential that exception handling must include mechanisms to recover from failure as well as to detect it. Besides, failure is something that can happen to a workflow enactment engine. A related issue is the monitoring of the workflow. Additionally, while restarting a workflow from a failure checkpoint, a user may wish to track the progress of enactment. In some cases the workflow is event driven, and a log of these events that trigger the workflow processing can be analyzed to understand how the workflow is progressing. This is also an important aspect of debugging a workflow. A user may wish to execute the workflow step-by-step to understand the potential errors in the flow logic.

Tools and programming interfaces for scientific workflow composition are important components of workflow systems. Through tools and interfaces, a developer can compose her/his scientific workflow and express details about data sources, task dependencies, resource availability, and other design or execution constraints. Most scientific workflow systems offer graphical interfaces that offer high-level mechanisms for composition, although a few systems exhibit traditional text-based programming interfaces. Workflow users exploit the features of the interfaces that scientific workflow systems expose in order to build their workflows. This corresponds to the design stage of a workflow. In general terms, two main workflow levels can be found in this regard, though there are other approaches that even differentiate more abstraction levels:

- *Abstract workflows*: at this high level of abstraction a workflow contains just information about what have to be done at each task along with information about how tasks are interconnected. There is no notion of how input data is actually delivered or how tasks are implemented.
- *Concrete workflows*: the mapping stage to a concrete workflow annotates each of the tasks with information about the implementation and/or resources to be used. Information about method invocation and actual data exchange format are also defined.

In case a user is familiar with the technology and resources available, they can even specify concrete workflows directly. Once a workflow specification is produced, it is sent to the workflow engine for the execution phase. At this stage, workflow tasks are mapped also onto third-party, distributed, and heterogeneous resources and the scientific computations are accomplished (Juve and Deelman, 2008).

3.2.2 Workflow Management Systems

As discussed earlier in this section, WMSs are software environments providing tools to define, compose, map, and execute workflows. There are several WMSs on the market, most of them targeted to a specific application domain. Here we focus on some significant WMSs developed by the research community, with the goal of identifying the most important features and solutions that have been proposed for workflow management in the domain of data analysis.

Although a standard workflow language like Business Process Execution Language (Juric, 2006) has been defined, scientific workflow systems often have developed their own workflow model for allowing users to represent workflows. Other than BPEL, other formalisms like UML, Petri nets (Guan et al., 2006), and XML-based languages (Atay et al., 2007) are used to express workflows. This feature makes it difficult to share workflow specifications and limits interoperability among workflow-based applications that are developed by using different WMSs. Nevertheless, there are some historical reasons for that, as many scientific workflow systems and their workflow models were developed before BPEL existed (Andrews et al., 2003).

WMSs are very useful in solving complex problems and offering real solutions for improving the way business analysis and scientific discovery is done in all domains, as discussed further. Solutions adopted are useful for taking into account new WMSs and represent features that workflow systems must include to support users in scientific application development.

Existing workflow models can be grouped roughly into two main classes:

- *Script-like systems*: where workflow descriptions specify workflows by means of a textual programming language that can be described by a grammar in an analogous way to traditional programming languages such as Perl, Ruby, or Java. They often have complex semantics and an extensive syntax. These types of descriptions declare tasks and parameters by means of a textual specification. Typically, data dependencies can be established between them by annotations. These languages contain specific workflow constructs, such as sequences, loops, while–do, or parallel constructs in order to build up a workflow. Examples of script workflow descriptions are JS4Cloud, Swift, and Karajan. A commonly used script-based approach to describe workflows, mainly in the business workflow community, is BPEL and its version for Web services that builds on IBM's Web Service Flow Language, WSFL.
- *Graphical-based systems*: where workflow models specify the workflow with only a few basic graphical elements that correspond to the graph components such as nodes and edges. Compared with script-based descriptions, graphical-based systems are easier to use and more intuitive for the unskilled user mainly because of graphical representation. While the nodes typically represent workflow tasks, communications (or data dependencies) between different tasks are represented as links going from one node to another. Workflow systems that support graph-based models often incorporate graphical user interfaces (GUIs) that allow users to model workflows by dragging and dropping graph elements. Purely graph-based workflow descriptions generally utilize DAGs. As mentioned, DAG-based languages offer a limited expressiveness, so that they cannot represent complex workflows (e.g., loops cannot be expressed directly).

3.2.3 Workflow Management Systems for Clouds

Several systems have been designed to develop scientific and data analysis workflows on high performance computing systems using script-based or visual formalisms (Talia, 2013). Some of them have been implemented on parallel computing systems, other on grids, recently some have made available on clouds (Hidders et al., 2015). In the following, we shortly discuss the most representative WMSs that support either script-based or visual workflow design and implementation in cloud environments such as Pegasus, Taverna, Kepler, WS-PGRADE, CloudFlows, E-Science Central, Data Mining Cloud Framework (DMCF), COMPSs, and Swift. Some of these systems (Pegasus, Swift, ClowdFlows) and – in more detail – DMCF, will be further discussed in Chapter 4.

Pegasus (Deelman et al., 2009), developed at the University of Southern California, includes a set of technologies to execute workflow-based applications in a number of different environments, including desktops, clusters, and grids. It has been used in several scientific areas including bioinformatics, astronomy, earthquake science, gravitational wave physics, and ocean science. The Pegasus WMS can manage the execution of an application formalized as a visual workflow by mapping it onto available resources and executing the workflow tasks in the order of their dependencies.

Taverna (Wolstencroft, 2013) is a WMS developed at the University of Manchester. Its primary goal is supporting the life sciences community (biology, chemistry, and medicine) to design and execute scientific workflows and support *in silico* experimentation, where research is performed through computer simulations with models closely reflecting the real world. Even though most Taverna applications lie in the bioinformatics domain, it can be applied to a wide range of fields since it can invoke any REST or SOAP-based Web service. This feature is very useful to allow users of Taverna to reuse code (represented as a service) that is available on the Internet. Taverna can orchestrate Web services and these may be running in the cloud, but this is transparent for Taverna, as demonstrated in the BioVel project.

Kepler (Ludäscher et al., 2006) is a graphical WMS that has been used in several projects to manage, process, and analyze scientific data.

Kepler provides a GUI for designing scientific workflows, which are a structured set of tasks linked together that implement a computational solution to a scientific problem. Data is encapsulated in messages or tokens, and transferred between tasks through input and output ports. Kepler provides an assortment of built-in components with a major focus on statistical analysis and supports task parallel execution of workflows using multiple threads on a single machine.

WS-PGRADE (Kacsuk et al., 2012) is a general-purpose WMS that allows users to create and run workflows on distributed computing systems such as grid and cloud platforms. The system allows users to define workflows through a graphical interface and to execute them on different distributed computing infrastructures (DCIs), including popular cloud systems such as Amazon EC2 and Google App Engine. The visual formalism expresses parallelism through parallel paths or through parametric input nodes. A parametric input node will be executed in as many instances as many files arrive on its port. End-users may use the system through a simplified interface where they can download a workflow from a repository, configure its parameter, and launch and monitor its execution on the underlying DCI.

ClowdFlows (Kranjc et al., 2012) is a cloud-based platform for the composition, execution, and sharing of interactive data mining workflows. According with the Software as a Service approach, ClowdFlows provides a user interface that allows programming visual workflows in any Web browser. Additionally, its service-oriented architecture allows third party services (e.g., Web services wrapping open-source or custom data mining algorithms). The server side consists of methods for the client side workflow editor to compose and execute workflows, and a relational database of workflows and data.

E-Science Central (e-SC) (Hiden et al., 2013) is a system that allows scientists to store, analyze, and share data in the Cloud. Like ClowdFlows, e-SC provides a user interface that allows programming visual workflows in any Web browser. Its in-browser workflow editor allows users to design a workflow by connecting services, either uploaded by themselves or shared by other users of the system. One of the most common use cases for e-SC is to provide a data-analysis back end to a standalone desktop or Web application. To this end, the e-SC API provides a set of workflow control methods and data structures. In the current implementation, all the workflow services within a single invocation of a workflow execute on the same cloud node.

Differently from the systems above, which support visual workflow design, the Data Mining Cloud Framework (DMCF) provides both visual and script-based workflow programming with the JS-4Cloud script language and the VL4Cloud visual formalism, so as to meet the needs of both high level users and who prefer to program. Moreover, DMCF natively supports the execution of workflow tasks on distributed environments composed by multiple machines, and is able to parallelize the execution of the tasks of each workflow, an important feature to ensure scalable data analysis workflows execution on the cloud.

COMPSs (Marozzo et al., 2012a) is a programming model and an execution runtime, whose main objective is to ease the development of workflows for distributed environments, including private and public clouds. With COMPSs, users create a sequential application and specify which methods of the application code will be executed remotely. Providing an annotated interface where these methods are declared with some metadata about them and their parameters does this selection. The runtime intercepts any call to a selected method creating a representative task and finding the data dependencies with all the previous ones during application execution.

Swift (Wilde et al., 2011) is a parallel scripting language that runs workflows across several distributed systems, like clusters, clouds, grids, and supercomputers. It provides a functional language in which workflows are modeled as a set of program invocations with their associated command-line arguments, input and output files. Swift uses a C-like syntax consisting of function definitions and expressions that provide an implicit data-driven task parallelism. The 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.

3.3 NoSQL MODELS FOR DATA ANALYTICS

With the exponential growth of data to be stored in distributed network scenarios, relational databases exhibit scalability limitations that significantly reduce the efficiency of querying and analysis (Abramova et al., 2014). In fact, most relational databases have little ability to scale horizontally over many servers, which makes challenging storing and managing the huge amounts of data produced everyday by many applications.

The *NoSQL* or *non-relational* database approach became popular in the last years as an alternative or as a complement to relational databases, in order to ensure horizontal scalability of simple read/write database operations distributed over many servers (Cattell, 2010). Compared to relational databases, NoSQL databases are generally more flexible and scalable, as they are capable of taking advantage of new nodes transparently, without requiring manual distribution of information or additional database management (Stonebraker, 2010). Since database management may be a challenging task with huge amounts of data, NoSQL databases are designed to ensure automatic data distribution and fault tolerance (Gajendran, 2012).

In the remainder of this section, we introduce the key features of the NoSQL approach, provide a classification of NoSQL databases, describe some representative NoSQL systems, and discuss some use cases for NoSQL databases, with a focus on data analytics.

3.3.1 Key Features of NoSQL

According to Cattell (2010), Not Only SQL (NoSQL) systems are generally characterized by six key features:

- The capability to horizontally scale simple operations over many servers
- The support to replication and partitioning of data over many servers
- A simple call level interface, in contrast to an SQL binding
- An efficient use of distributed indexes and RAM for data storage
- The capability to dynamically add new attributes to data records
- Do not provide ACID transactional properties of most relational databases

The first feature emphasizes the need to provide *horizontal scalability* on *simple operations*, for example, reading or writing a small number of records in each operation. This happens, for instance, in applications searching and updating multiserver databases of personal profiles, Web postings, and customer records. The term horizontal scalability refers to the capability of distributing both data and load of these simple operations over many servers, with no RAM or disks shared among the servers.

Support to replication and partitioning is a key feature to ensure both fault tolerance and scalability. Most NoSQL systems allow *sharding*, that is the horizontal partitioning of data by storing records on different servers according to some key. Some systems also allow vertical partitioning, which stores parts of a single record on different servers.

Regarding the third key feature, even if most NoSQL databases provide a simple call level interface, in contrast to Structured Query Language (SQL), which is the language most commonly associated with relational databases, partial support to SQL-based querying is possible even though the underlying database is not relational.

The fourth key feature usually associated with NoSQL databases is an efficient use of distributed indexes and volatile memory for data processing. In particular, since I/O data access is relatively slow, mapping data into RAM increases performance and reduces the time necessary for querying. In addition, in-memory processing can be highly scalable, because data can be replicated and partitioned over the RAM of multiple servers.

In contrast to relational databases, NoSQL databases are schemaless, which means that new data items can be stored even if they do not adhere to a predefined structure. This is another key feature of NoSQL databases, as it provides applications with the ability to dynamically add new attributes to data records, without modifications to predefined schemas.

Finally, NoSQL databases generally do not satisfy the atomicity, consistency, isolation, and durability (ACID) properties of most relational databases. On the other hand, most NoSQL databases adhere to the basically available, soft state, and eventually consistent (BASE) principle, which is characterized by high availability of data, while sacrificing its consistency (Abramova et al., 2014; Pritchett, 2008; Cook, 2009). The BASE principle derives from the CAP theorem (Gilbert and Lynch, 2002), which states that a distributed system can have only two out of three of the following properties: consistency (C), availability (A), and partition tolerance (P). The NoSQL systems generally give up strong consistency, which is hard to achieve across multiple servers, in favor of availability and partition tolerance. According to the BASE principle, most NoSQL databases provide *eventual consistency*, meaning that updates are *eventually* propagated to all nodes.

3.3.2 Classification of NoSQL Databases

NoSQL databases provide ways to store scalar values (e.g., numbers, strings), binary objects (e.g., images, videos), or more complex values. According to their data model, NoSQL databases can be grouped into three main categories (Cattell, 2010): *key-value stores*, *document stores*, *extensible record stores*.

- *Key-value stores*: provide mechanisms to store data as (*key, value*) pairs over multiple servers. In these databases, a distributed hash table (DHT) can be used to implement a scalable indexing structure, where data retrieval is performed by using *key* to find *value*. However, key-value stores typically go beyond the standard insert, delete, and lookup operations primitives of DHTs, by providing additional functionalities for replication, versioning, locking, transactions, sorting, and other features (Cattell, 2010).
- *Document stores*: are designed to manage data stored in documents that use different formats (e.g., JSON), where each document is assigned a unique key that is used to identify and retrieve the document. Therefore, document stores extend key-value stores because they provide for storing, retrieving, and managing semistructured information, rather than single values. Unlike the key-value stores, document stores generally support secondary indexes and multiple types of documents per database, and provide mechanisms to query collections based on multiple attribute value constraints (Cattell, 2010).
- *Extensible record stores*: also called *Wide column stores,* provide mechanisms to store *extensible records* that can be partitioned

across multiple servers. In this type of database, records are said to be *extensible* because new attributes can be added on a per record basis. Extensible record stores provide both horizontal partitioning (storing records on different nodes) and vertical partitioning (storing parts of a single record on different servers). In some systems, columns of a table can be distributed over multiple servers by using *column groups*, where predefined groups indicate the columns that are best stored together.

3.3.3 NoSQL Systems

There are currently tens of NoSQL databases in the market, each one with different solutions and optimizations. In the following we present three examples of NoSQL databases belonging to the three main categories introduced earlier: Amazon's Dynamo (De Candia et al., 2007) as an example of key-value store; MongoDB (Plugge et al., 2010) as an example of document store; and Google's Bigtable (Chang et al., 2006) as an example of extensible record store. These databases will be shortly described based on their data model, query model, architecture, replication and partitioning strategies, consistency model, and failure handling.

3.3.3.1 Dynamo

Developed by Amazon to support its services, Dynamo is one of the most important NoSQL systems that has influenced the design of several other key-value stores.

- *Data model*: Dynamo adopts a simple *key-value* data model, in which values are stored as binary objects. Despite its simplicity, this is a very effective data model for services like product catalogue management, where operations are mostly limited to objects that can be accessed by key.
- *Query model*: Dynamo provides a simple querying interface that reflects the underlying data model. Basically, two operations are available: *get*, to retrieve an object given its key, and *put,* to store an object with associated key.
- *Architecture*: Dynamo adopts a "shared nothing" architecture, without any master or shared file system. Servers are linked to each other to form a ring overlay, in which every server has the ability to route requests for any key to the appropriate server. The ring topology and associated routing strategy ensures load balancing and scalability.

- *Replication and partitioning*: given an object and its key, the object is first stored on the server responsible for the interval in which the key lies. Then, the server forwards a replication message to a predefined number of its successors along the ring. In this way, the object is replicated on a number of nodes, all of which has the ability of returning that object given the corresponding key. Data partitioning is based on a variant of the consistent hashing technique typically adopted in DHTs.
- *Consistency*: the Dynamo database is eventually consistent. All the write operations are completed immediately without waiting that all replicas have received the update. Thus, no commit is required on concurrent writes. Sequential reads may return different values if made on different replicas. For this reason, Dynamo implements Multi-Version Concurrency Control (MVCC), which creates a new immutable version of an object after each write. In fact, when an MVCC database updates a data element, it does not overwrite the old value of that data element with new data, but instead it marks the old data as obsolete and adds the newer *version*.
- *Failure handling*: Dynamo implements a *gossip*-based strategy that allows every node to notify all the other nodes in the ring about topological variations, and the consequent necessity of keys-to-nodes remapping, originated by nodes leaving the network for a failure.

3.3.3.2 MongoDB

MongoDB is one of the most popular NoSQL databases based on documents, which can also be used to store and distribute large binary files such as images or video.

- *Data model*: in MongoDB, documents are saved in Binary JSON (BSON) format, which is an extension of JSON with support for additional types. The internal structure of documents is not tied to a scheme. Therefore, applications can add or remove a field without limitations. Besides documents, MongoDB introduces the concept of Collection that allows grouping similar documents together.
- *Query model*: client applications can communicate with MongoDB through a driver that handles interactions with an appropriate language. Similarly, as in the case for relational databases, the driver

supports Create, Read, Update, and Delete (CRUD) operations, which are specified through a document in BSON format.

- *Architecture*: the architecture of MongoDB includes three main components: shards, routers, and config servers. Shards are the nodes that store the data in the form of chunks. Routers interface with client applications and route operations to the appropriate shards. Config server keep track of which chunks is responsible each shard in the network.
- *Replication and partitioning*: to ensure reliability and availability, shards are typically organized in clusters with internal replication. Within a sharded cluster, partitioning is enabled on a per-database basis. After enabling sharding for a database, it is possible to choose which collections to shard. For each sharded collection, a shard key determines the distribution of the collection's documents among the cluster's shards.
- *Consistency*: MongoDB does not support multidocument transactions. However, it provides atomic operations on a single document. In many cases, these document-level atomic operations are sufficient to solve problems that would require ACID transactions in a relational database.
- *Failure handling*: MongoDB has different behaviors depending on the type of failure that has occurred in the system. For example, the failure of a replica does not affect availability because write operations are always assigned to the primary node of a cluster. If it is the primary node to fail, however, data remains temporarily unreachable until a new primary is elected.

3.3.3.3 Bigtable

Designed by Google, Bigtable is one of the most popular extensible record stores. Built above the Google File System, it is used in many services with different needs: some require low latencies to ensure the real-time response to users, and other more oriented to the analysis of large volumes of data.

- *Data model*: data in Bigtable are stored in sparse, distributed, persistent, multidimensional tables. Each value is an uninterpreted array of bytes indexed by a triplet (row key, column key, and timestamp). Data are maintained in lexicographic order by row key. The row range for a table is dynamically partitioned. Each row range is called a *tablet*, which is the unit of distribution and load balancing. Column keys are grouped into sets called *column families*. All data stored in the same column family are usually of the same type.

- *Query model*: Bigtable provides a C++ library that allows users to filter data based on row keys, column keys, and timestamps. It provides functions for creating and deleting tables and column families. The library allows clients to write or delete values in a table, look up values from individual rows, or iterate over subsets of data in a table.
- *Architecture*: Bigtable includes several components. The *Google File System* (*GFS*) is used to store log and data files. *Chubby* provides a highly-available and persistent distributed lock service. Each tablet is assigned to one *Tablet server*, and each tablet server typically manages up to a thousand tablets. A *Master server* is responsible for assigning tablets to tablet servers, detecting the addition or expiration of tablet servers, and balancing the load among tablet servers.
- *Replication and partitioning*: partitioning is based on the tablet concept introduced earlier. A tablet can have a maximum of one server that runs it and there may be periods of time in which it is not assigned to any server, and therefore cannot be reached by the client application. Bigtable does not directly manage replication because it requires that each tablet is assigned to a single server, but uses the GFS distributed file system that provides replication of tablets as files called SSTables.
- *Consistency*: the partitioning strategy assigns each tablet to one server. This allows Bigtable to provide strong consistency at the expense of availability in the presence of failures on a server. Operations on a single row are atomic, and can support even transactions on blocks of operations. Transactions across multiple rows must be managed on the client side.
- *Failure handling*: when a tablet server starts, it creates a file with a unique name in a default directory in the Chubby space and acquires exclusive lock. The master periodically checks whether the servers still have the lock on their files. If not, the master assumes the servers have failed and marks the associated tablets as unassigned, making them ready for reassignment to other servers.

3.3.4 Use Cases

The various features of NoSQL databases discussed here are particularly useful to perform data analysis on large data volumes.

Key-value stores can be one of the most effective solutions to perform real time analysis in scenarios where operations are mostly limited to objects, which can be accessed by key, due to the fact that these databases are very fast in retrieving items given their key. The case of Amazon's Dynamo shows that key-value store are particularly appropriate for e-commerce systems where is necessary to store and manage huge amount of data about products, customer preferences, shopping carts, sales rank and session management. This enables a wide range of analytics to be performed on such data, such as association rule learning aimed at uncovering connections between objects such as customers, sellers and products.

Document stores such as MongoDB are an effective choice to store and process documents of any structure, such as events data, time series data, text, and binary data. Through sharding, huge amounts of data can be horizontally partitioned across a large number of commodity servers, with complete application transparency, that is, without requiring users to build custom partitioning and caching layers. This scalability enables powerful analytics to be performed in real time, by exploiting distribution of data and computation. For example, MongoDB can run complex *ad hoc* analytics, thanks to its query support, which includes secondary, geospatial, and text search indexes, as well as native support to MapReduce.

Use cases for extensible record stores are similar to those for document stores, with multiple types of objects that can be queried by any field. However, extensible record store systems are generally aimed at higher throughput, and may provide stronger concurrency guarantees, at the cost of slightly more complexity than document stores (Cattell, 2010). A well-known example of data analytics service built on top of an extensible record store is Google Analytics, which uses Bigtable (Chang et al., 2006). Google Analytics provides aggregate statistics about web sites, such as the number of unique visitors per day, page views per day, as well as site-tracking reports, such as the percentage of customers who made a purchase, given the total number of page visitors. Whenever a page is visited, Google Analytics records a user identified and information about the page being fetched. Google Analytics summarizes this data and makes it available to webmasters. The service makes use of two tables. The raw click table maintains a row for each end-user session. The summary table contains various summaries for each website, generated from the raw click table by MapReduce jobs executed on a periodic basis.

3.4 SUMMARY

The development of data analysis applications on cloud computing systems is a complex task that needs to exploit smart software solutions and innovative technologies. Such software tools, frameworks, and solutions allow clouds to play as enablers for big data analytics by implementing the data-analysis-as-a-service model. The software tools and technologies we presented in this chapter are based on three paradigms and associated tools, which represent the main pillars for developing scalable data analysis on clouds: MapReduce, workflow systems, and NoSQL database management systems.

MapReduce is widely used to implement scalable data analysis algorithms and applications executed on multiple machines, to efficiently analyze big amounts of data. Hadoop is the best-known MapReduce implementation and it is commonly used to develop scalable applications analyzing big amounts of data. As discussed, Hadoop is also a reference tool for several other frameworks, such as Storm, Hive, Oozie, and Spark. Moreover, besides Hadoop and its ecosystem, several other MapReduce implementations have been implemented within other systems, including GridGain, Skynet, MapSharp, and Disco. Despite the scalability and large availability of MapReduce solutions, in some cases more general tools are needed to develop irregular concurrent data analysis applications that cannot be expressed as a combination of *map* and *reduce* functions. In those cases, systems based on the workflow paradigm are often used. For this reason, in the past years, many efforts have been done to develop distributed WMSs for complex data-driven applications. Workflows provide a declarative way of specifying the high level logic of an application, hiding the low level details. They are also able to integrate existing software modules, datasets, and services in complex compositions that implement discovery processes. Here we discussed the workflow programming models and presented several WMSs, which have been implemented on clouds and are used to develop scalable data analysis applications. Finally, we also discussed NoSQL database technology that became popular in the last few years, as an alternative or complement to relational databases. In fact, NoSQL systems in several application scenarios are more scalable and provide higher performance than relational databases. We introduced the basic principles of NoSQL, described representative NoSQL systems, and outlined interesting data analytics use cases where NoSQL tools are useful.

REFERENCES

Abramova, V., Bernardino, J., Furtado, P., 2014. Which NoSQL database? A performance overview. OJDB 1 (2).

Al-Shakarchi, E., Cozza, P., Harrison, A., Mastroianni, C., Shields, M., Talia, D., Taylor, I., 2007. Distributing workflows over a ubiquitous P2P network. Sci. Prog. 15 (4), 269–281.

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S., 2003. "Business process execution language for web services". OASIS specification.

Atay, M., Chebotko, A., Liu, D., Lu, S., Fotouhi, F., 2007. Efficient schema-based XML-to-relational data mapping. Inform. Syst. 32 (3), 458–476.

Bell, G., Hey, T., Szalay, A., 2009. Beyond the data deluge. Science 323 (5919), 1297–1298.

Bowers, S., Ludaescher, B., Ngu, A., Critchlow, T., 2006. Enabling scientific workflow reuse through structured composition of dataflow and control-flow. In: Proceedings of the Twenty-Second International Conference on Data Engineering Workshops (ICDEW'06). Washington, DC.

Cannataro, M., Guzzo, A., Comito, C., Veltri, P., 2004. Ontology-based design of bioinformatics workflows on PROTEUS. JDIM 2 (1), 87–92, Digital Information Research Foundation (DIRF) Press.

Cary, A., Sun, Z., Hristidis, V., Rishe, N., 2009. Experiences on processing spatial data with MapReduce. In: Winslett, M. (Ed.), In: Proceedings of the twenty-first International Conference on Scientific, Statistical Database Management (SSDBM'09), Springer- Verlag, Berlin, Heidelberg. pp. 302–319.

Cattell, R., 2010. Scalable SQL and NoSQL data stores. SIGMOD Record 39 (4), 12–27.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R., 2006. Bigtable: a distributed storage system for structured data. OSDI 2006.

Chu, C., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G., Ng, A.Y., Olukotun, K., 2007. Map-reduce for machine learning on multicore. Advances in neural information processing systems 19, 281.

Coleman, R., Ghattamaneni, U., Logan, M., Labouseur, A., 2012. Computational finance with Map-Reduce in Scala. Conference on Parallel and Distributed Processing (PDPTA'12). Las Vegas, NV, pp. 16–19.

Cook, J.D., 2009. ACID versus BASE for database transactions. http://www.johndcook.com/blog/2009/07/06/brewer-cap-theorem-base/.

Couvares, P., Kosar, T., Roy, A., Weber, J., Wenger, K., 2007. Workflow management in Condor. Workflows for e-Science. Springer, New York, pp. 357–375.

Das, S., Sismanis, Y., Beyer, K.S., Gemulla, R., Haas, P.J., McPherson, J., 2010. Ricardo: Integrating R and Hadoop. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10) ACM. New York, NY, USA, pp. 987–998.

De Candia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W., 2007. Dynamo: Amazon's highly available key-value store. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07), ACM. New York, NY, USA, pp. 205–220.

Dean, J., Ghemawat, S., 2004. MapReduce: Simplified data processing on large clusters. Sixth USENIX Symposium on Operating Systems Design and Implementation (OSDI'04). San Francisco, USA.

Dean, J., Ghemawat, S., 2008. MapReduce: simplified data processing on large clusters. Communications of the ACM 51 (1), 107–113.

Dou, A., Kalogeraki, V., Gunopulos, D., Mielikainen, T., Tuulos, V.H., 2010. Misco: A MapReduce Framework for Mobile Systems. Third International Conference on Pervasive Technologies Related to Assistive Environments (PETRA'10). New York, USA.

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

Ekanayake, J., Pallickara, S., Fox, G., 2008. Mapreduce for data intensive scientific analyses. Fourth IEEE International Conference on e-Science (e-Science'08). Indianapolis, USA, pp. 277–284.

Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., Fox, G., 2010. Twister: a runtime for iterative MapReduce. First International Workshop on MapReduce and its Applications (MAPREDUCE'10). Chicago, USA, pp. 110–119.

Fedak, G., He, H., Cappello, F., 2009. BitDew: A data management and distribution service with multi-protocol and reliable file transfer. J. Netw. Comput. Appl. 32 (5), 961–975.

Gajendran, S., 2012. A Survey on NoSQL Databases, http://ping.sg/story/A-Survey-on-NoSQL-Databases–Department-of-Computer-Science.

Georgakopoulos, D., Hornick, M., Sheth, A., 1995. An overview of workflow management: from process modeling to workflow automation infrastructure. Distrib. Parallel Dat. 3 (2), 119–153.

Gilbert, S., Lynch, N., 2002. Brewer's conjecture and the feasibility of consistent, available, and partition-tolerant web services. ACM SIGACT News 33 (2), 51–59.

Gongqing, W., Haiguang, L., Xuegang, H., Yuanjun, B., Jing, Z., Xindong, W., 2009. MReC4.5: C4.5 ensemble classification with MapReduce. ChinaGrid Annual Conference, 2009. ChinaGrid '09. China, vol. 4, pp. 249, 255.

Guan, Z., et al., 2006. Grid-flow: a grid-enabled scientific workflow system with a petri net-based interface. Ph.D. Thesis, University of Alabama at Birmingham.

Guo, Z., Fox, G., Zhou, M., 2012. Investigation of data locality in MapReduce. In: Proceedings of the 2012 Twelfth IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'12). Washington, DC, USA.

Hidders, J., Missier, P., Sroka, J. (Eds.), 2015. Recent advances in Scalable Workflow Enactment Engines and Technologies. Future Gener. Comp. Syst. 46, 1–2.

Hiden, H., Woodman, S., Watson, P., Cala, J., 2013. Developing cloud applications using the e-Science central platform, philosophical transactions of the royal society A. Math. Phys. Eng. Sci. 371 (1983), 1–12.

Hollingsworth, D., 1995. Workflow management coalition specification: the workflow reference model, Document Number TC00-1003, v. 1.1.

Juric, M.B., 2006. Business Process Execution Language for Web Services BPEL and BPEL4WS, Second ed. Packt Publishing.

Juve, G., Deelman, E., 2008. Resource provisioning options for large-scale scientific workflows. Third International Workshop on Scientific Workflows and Business Workflow Standards in e-Science. Indianapolis, USA.

Kacsuk, P., Farkas, Z., Kozlovszky, M., Hermann, G., Balasko, A., Karoczkai, K., Marton, I., 2012. Ws-pgrade/guse generic dci gateway framework for a large variety of user communities. J. Grid Comp. 10 (4), 601–630.

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

Lin, C., Lu, S., Lai, Z., Chebotko, A., Fei, X., Hua, J., Fotouhi, F., 2008. Service-oriented architecture for VIEW: a visual scientific workflow management system. In: Proceedings of IEEE International Conference Services Computing (SCC '08). Washington, DC, USA, pp. 335–342.

Lin, H., Ma, X., Archuleta, J., Feng, W.-C., Gardner, M., Zhang, Z., 2010. MOON: MapReduce on opportunistic environments. In: Nineteenth International Symposium on High Performance Distributed Computing (HPDC'10). Chicago, USA.

Lin, M., Lee, P., Hsueh, S., 2012. Apriori-based frequent itemset mining algorithms on MapReduce. In: Proceedings of the Sixth International Conference on Ubiquitous Information Management and Communication (ICUIMC '12). New York, USA.

Liu, L., Pu, C., Ruiz, D., 2004. A systematic approach to flexible specification, composition, and restructuring of workflow activities. J. Dat. Manag. 15 (1), 1–40.

Litzkow, M., Livny, M., Mutka, M., 1988. Condor – a hunter of idle workstations. In: Proceedings of the Eighth International Conference on Distributed Computing Systems, IEEE Computer Society, New York, pp. 104–111.

Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E., Tao, J., Zhao, Y., 2006. Scientific workflow management and the Kepler system. Concurr. Comput. 18 (10), 1039–1065.

Ma, Q., Yang, B., Qian, W., Zhou, A., 2009. Query processing of massive trajectory data based on mapreduce. In: Proceedings of the First International Workshop on Cloud Data Management (CloudDB '09). New York, NY, USA, pp. 9–16.

Marozzo, F., Lordan, F., Rafanell, R., Lezzi, D., Talia, D., Badia, R.M., 2012a. Enabling cloud interoperability with Compss. In: Proceedings. of the Eighteenth International European Conference on Parallel and Distributed (Europar 2012), vol. 7484. Lecture Notes in Computer Science. Rhodes Island, Greece, pp. 16–27.

Marozzo, F., Talia, D., Trunfio, P., 2012b. P2P-MapReduce: parallel data processing in dynamic Cloud environments. J. Comput. Syst. Sci. 78 (5), 1382–1402.

Mirto, M., Passante, M., Aloisio, G., 2010. A grid meta scheduler for a distributed interoperable workflow management system. In: Proceedings of the 2010 IEEE Twentieth International Symposium on Computer-Based Medical Systems, CBMS'10, IEEE Computer Society. Washington, DC, USA, pp. 138–143.

Plugge, E., Hawkins, T., Membrey, P., 2010. The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing, First ed. Apress, Berkely, CA, USA.

Pritchett, D., 2008. BASE: an acid alternative. ACM Queue 6 (3), 48–55.

Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C., 2007. Evaluating MapReduce for multi-core and multiprocessor systems. Thirteenth International Symposium on High-Performance Computer Architecture (HPCA'07). Phoenix, USA.

Sonntag, M., Karastoyanova, D., Deelman, E., 2010. Bridging the gap between business and scientific workflows. In: Proceedings of the Sixth IEEE International Conference on e-Science, IEEE Computer Society.

Stonebraker, M., 2010. SQL databases vs. NoSQL databases. Commun. ACM 53 (4), 10–11.

Sun, Z., Fox, G., 2012. Study on parallel SVM based on MapReduce. International Conference on Parallel and Distributed Processing Techniques and Applications. Las Vegas, USA, pp. 16–19.

Talia, D., 2013. Workflow systems for science: concepts and tools. ISRN Software Engineering.

Tang, J., Sun, J., Wang, C., Yang, Z., 2009. Social influence analysis in large-scale networks. In: Proceedings of the Fifteenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '09). New York, USA.

Tang, B., Moca, M., Chevalier, S., He, H., Fedak, G., 2010. Towards MapReduce for desktop Grid computing. In: Fifth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC'10). Fukuoka, Japan.

Taylor, I.J., Deelman, E., Gannon, D.B., Shields, M. (Eds.), 2007. Workflows for e-Science: Scientific Workflows for Grids. Springer, London.

van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P., 2003. Workflow patterns. Distrib. Parallel Dat. 14 (1), 5–51.

White, T., 2009. Hadoop: The Definitive Guide, First ed. O'Reilly Media, Inc.

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.

Wolstencroft, K., et al., 2013. The Taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. Nucleic Acids Res. 41 (W1), W557–W561.

Workflow Management Coalition, Terminology and Glossary, Document Number WFMC-TC-1011, Issue 3.0, 1999.

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.

Yu, J., Buyya, R., 2005. A taxonomy of scientific workflow systems for grid computing. SIGMOD Record 34 (3), 44–49.

Zhang, L., Zhang, J., Cai, H., 2007. Services Computing. Springer-Verlag, Berlin, Heidelberg.