BIGSCALE: AUTOMATIC SERVICE PROVISIONING FOR HADOOP CLUSTERS

Dan HURU [1], Cristian ESEANU [2], Cătălin LEORDEANU [3],
Elena APOSTOL [4], Valentin CRISTEA [5]

As the number of interconnected devices grows in the IoT space, data processing systems require increased resources, robustness and flexibility. In this sense the scalability of a system becomes very important. A scalable system can process variable data volumes, requires less costs for maintenance and allows for fault tolerance and high availability. While horizontal scalability is offered by multiple Cloud providers, vertical scalability is a less addressed topic. In this article we first define the meaning and outline the benefits of doing vertical scalability. We also present a scaling solution which can automatically provision services based on the needs and resource usage of the system.

Keywords: scalabilty, elasticity, resource provisioning, BigData, IoT

1. Introduction

Today various embedded devices are capable of communicating and sharing data using the Internet. In this manner traditional web services are enriched with physical world services. In addition to the IoT vision, which gives every device an IP address and interconnects them, there is also the notion of Web of Things (WoT) which enables the devices to speak the same language.Current real-time processing is mainly done on existing web data but the extension to considerably larger amounts of data produced by multiple sensor networks requires research and design of robust and scalable processing platforms.

[1] Ph.D. student, Faculty of Automatic Control and Computers, University Politehnica of Bucharest, e-mail: alexandru.huru2208@cti.pub.ro

[2] Master student, Faculty of Automatic Control and Computers, University Politehnica of Bucharest, e-mail: eseanu.cristian@cti.pub.ro

[3] Lecturer, Ph.D., Faculty of Automatic Control and Computers, University Politehnica of Bucharest, e-mail: catalin.leordeanu@cs.pub.ro

[4] Lecturer, Ph.D., Faculty of Automatic Control and Computers, University Politehnica of Bucharest, e-mail: elena.apostol@cs.pub.ro

[5] Professor, Ph.D., Faculty of Automatic Control and Computers, University Politehnica of Bucharest, e-mail: valentin.cristea@cs.pub.ro

Scalability can be described as the capacity to handle increasing workloads [1], or the ability to improve performance when resources are added [2]. In many articles (e.g. [3] and [4]) scalability is divided into two main categories:

- Vertical scalability : adding resources to the same logical unit (e.g. to a cluster node)
- Horizontal scalability: adding multiple "units of resources" [4] (e.g. adding multiple nodes to a cluster)

The multiple definitions of scalability try to take into account what is useful for the domain and to prove a point about the system/algorithm/application performance, how certain workflows affect the system, cost efficiency and the ability of a system/application to scale.

While horizontal scalabilty is achieved at the infrastructure level by many Cloud providers, services can also be scaled to further optimize existing applications.

Some of the benefits of this type of scaling can be: greater precision when measuring service utilization; enforcing SLAs or quality levels; cost optimization; personalized usage/Usage patterns; less interventions from cluster administrator.

In this paper we propose a solution that enables automatic scaling for Hadoop based applications, in a Cloud environment. It employs three stategies: utilize fewer resources, maximum throughput, keep resource utilization under a threshold. The application includes automatic and manual resource allocation and a metrics monitor.

The paper has the following structure: first we review related work and state of the art in *Section II*. In *Section III* we propose our high-level solution, while in *Section IV* we showcase implementation details. *Section V* describes the experimental results and we conclude our work in *Section VI*.

2. **Related work**

The most established Cloud Providers that achieve automatic scaling are described below.

Amazon Elastic Cloud Compute (EC2) is a web service which offers computing power to users [5]. Auto scaling in EC2 has the following components [6]: groups (a collection of EC2 instances), launch configurations (used when creating new instances) and scaling plans (they choose how to scale a group).

There are several scaling plans: maintain current number of instances running, manual scaling, scale based on a schedule and scale on demand. Maintain current number of instances running is accomplished by doing regularly health checks. If necessary, the unhealthy instance is terminated and a new one is launched instead. This is the default plan. Scale based on a schedule is done by performing time-based scaling operation Scale based on demand aka policy-based scaling. A policy is a set of rules executed by Auto Scaling in response to an alarm. An alarm is an object that monitors a metric for a

specified amount of time. An Auto scaling Group can have multiple scaling policies.

Google App Engine [7] is formed by multiple services. Each service has two components: source code and configuration file. The scaling type is specified in the configuration file and has three choices: manual scaling, basic scaling and automatic scaling. In basic scaling a instance is created when a request is received and is destroyed when the application is idle. In automatic scaling a instance is created/turned off on demand based on different application metrics.

Microsoft Azure[8] supports Azure Autoscale: dynamically add or remove instances based on schedule and/or on runtime metrics. In addition, Azure Resource Manager Rest API and/or Azure Service Management Rest API can be used for autoscaling. Azure can also use third-party services like Paraleap AzureWatch.

3. The Proposed Architecture

The system we propose can be functionally described in figure 1 and consists of six main components: the Metrics Monitor, Hadoop Sinks, Metrics Collector, Ambari Server, Ambari Rest API, BigScale and Hadoop Cluster.

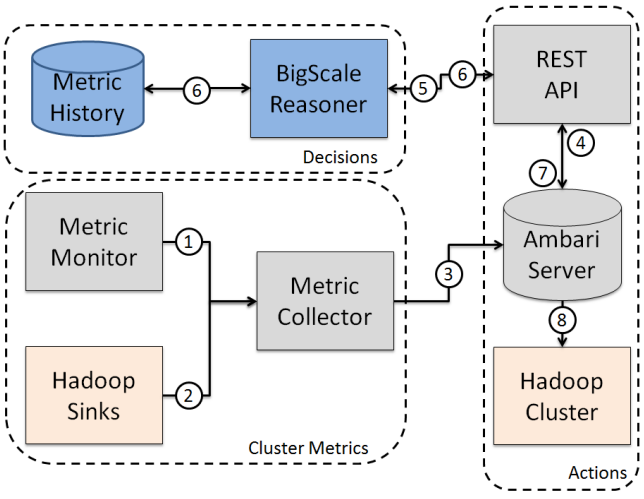

FIG. 1. The Proposed Architecture

The resource provisioning application dynamically scales slave components (Data nodes and Name nodes) to ensure the required infrastructure for a YARN application and to conform with system administrator requirements. In order to offer different options to users, there are three automatic scaling strategies.

(1) Use fewer resources - This is done by having equal number of running Node Managers and Data Nodes

(2) Balanced with minimum resources - The goal is to recommission/decommission Node Managers to maintain a certain level of resource utilization. The condition for recommissioning a Node Manager is that the memory utilization is above threshold t1 or CPU utilization is above threshold t2; and for decommissioning a Nodemanager is that the memory utilization is below threshold t3 and CPU utilization is below threshold t4. The CPU and Memory threshold (t1, t2, t3, t4) may differ.
(3) Highest performance - Allocate all processing power, Node Managers must be running - to achieve less running time for an application.

In strategy 1 the number of data nodes is equal to the number of node managers and new node managers are added if the containers exceed their allocated processing power. In strategy 2 scaling is done by setting thresholds for node managers. A new node manager is recommissioned if they exceed a certain threshold for a longer period of time. They are decommissioned if the resource utilization is below the specified threshold. In strategy 3 the running time of the applications is reduced by using all the available resources.

Data nodes behave the same regardless of the strategy: they scale up and down if the space and threshold requirements are not satisfied. Also, if there are no applications running, the recommission operation for data node/namenode is prohibited. For the decommissioning command, there is a different free space available condition for Data Nodes.

The number of Node Managers scales down to be equal to the number of Data Nodes. This operation is necessary for the system to be more cost-effective.

The application collects metrics offered by Ambari Metrics from the master services (YARN's Resource Manager and HDFS's Name node) and depending on the scaling strategy it sends commands to the Ambari Server. Commands are targeted for slave components of the master components mentioned above (master Resource Manager to slave Node Manager and master Name node to slave Data Node).

Decommissioning and recommissioning data nodes is done by sending a request through Ambari REST API to the Name node to include/exclude the data nodes hosts. Decommissioning and recommissioning Node Manager is done also by sending a request through Ambari REST API to Resource Manager to include/exclude node managers hosts. In addition, Resource Manager stops the decommissioned host. So when a Node Manager is recommissioned it is restored to its previous state.

4. **Implementation details**

4.1. **Workload types**

In this subsection we analyze the projected results when different types of YARN applications run in an Apache Ambari environment. Regardless of

the workload, an increase in in the number of Node Manager components will result in a decrease in the running time of the YARN application.

CPU and I/O. The Wordcount program, like the Pi calculation program, is interesting because is has a CPU and a I/O component. In this case, it is expected a reduction in time by recommissioning Node Managers but the monitoring is trickier in some situations because CPU usage level is dependent of I/O part.

I/O intensive. For an I/O intensive program, like Teragen or TESTDFSIO, adding Node Managers will reduce time and lower the resource utilization levels. The workload's objective is to write large quantities of data so adding Data Nodes will suffice. In most cases, adding a Data Node is followed by adding a new Node Manager and therefore the resource utilization will probably be lower. Because of the Hadoop write-once policy we expect applications like Teragen not to produce spikes in CPU and Memory utilization levels.

4.2. Experimental setup

Our experimental setup is based on OpenStack [10], an open-source cloud solution used for management and deploying IaaS infrastructure. It can scale "up to 1 million physical machines, up to 60 million virtual machines and billions of stored objects". The cluster we used for prototyping has the following configuration:

VCPUs	RAM (MB)	Disk (GB)	State
4	4096	24	Active
1	1536	16	Active
1	1536	16	Active
1	1536	16	Active
2	4096	10	Active
4	4096	24	Active
1	1536	16	Active
1	1536	16	Active

Although we achieve promising results with 8 machines, part of future work is to extend the experiments to larger clusters.

The metrics are collected as follows:

- Node Managers - CPU and Memory Utilization
- Data Nodes - Used space percentage and free space

On top of the cluster we installed an Apache Ambari Server[9] with the following services: HDFS[11], YARN[12], MapReduce[13], Ambari Metrics[14] and Zookeeper Server[15]. Zookeeper is used for coordinating distributed applications.

5. Experimental results

5.1. Test applications

The experiments used in this chapter express different workloads. The dynamic scheduling application's responsibility is to handle these workload properly. The following test applications are used: Teragen, Wordcount and Pi.

Teragen. Teragen application is an I/O write application that generates specified quantities of data.([34]) thus the nodemanager and data node scaling operations are tested by generating large quantities of data.

WordCount. The wordcount application sums up the number of appearances of each word in the input text([35]). In this experiment, we want to see how a workload of a program that utilises both CPU and I/O is handled.

Pi. The pi benchmark approximates the value of pi using quasi-Monte Carlo method.([36]). The experiment utilizes this program only for demonstrating the decrease of application running time when Node Managers are recommissioned.

5.2. Assumptions and results

In this subsection the truthfulness of the assumptions made and how the application behaves in different benchmarks are tested.
More computational resources decrease running time. if we add more Node Manager components the application running time will decrease.

For this test we used the PI benchmark, because in other benchmarks the I/O part may interfere and, as a consequence, the results may be inconsistent. We obtained the following results:

- 2 Manager Nodes: 204.341 s (runtime)
- 4 Manager Nodes: 108.144 s (runtime)
- 8 Manager Nodes: 66.881 s (runtime)

Scale in when no application is running. If there are no applications running, the slave components will scale in or out so the number of node manager components match the number of data node components and data node components will scale in if the free space condition specified in the configuration file is fulfilled.

In the current setup, free space threshold is set to 20 GB, 8 active Node Manager components, 5 active Data Node components and the *dfs* replication is set to 2. There is a difference as shown in the figure 2, between the time a Data Node was decommissioned and the time the remaining free space decreased. This happens because the name node needs some time to keep up with the system changes. The node manager components will scale in order to match the number of data nodes.

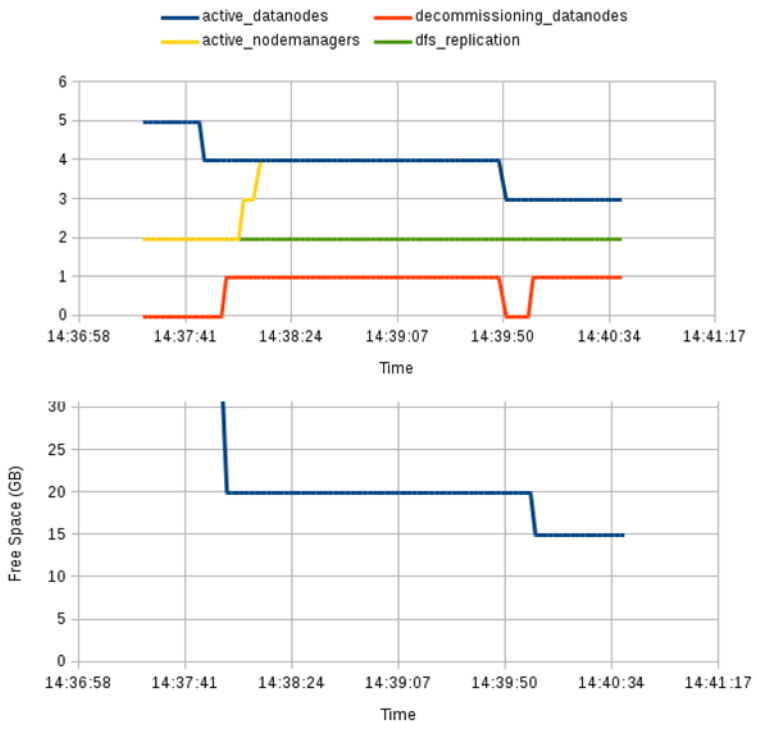

FIG. 2. Components scale in

Even though the free space condition is fulfilled, we cannot decommission a Data Node, because the number of decommissioning data nodes is equal to *dfs* replication plus 1. This condition is necessary for having a backup of the data in other active Data Node components.

Scaling while application are running. Teragen Mapreduce application is designed to write a large file for the terasort benchmark.

Scaling Strategy 1. The following setup is employed: 2 Data Nodes, 2 Node Manager, free space 9.9 GB, free space threshold 10 GB if used space is above 80% and 5 GB if less, node manager and data node add cooldown 30 s

In Figure 3 it can be observed that a node manager is recommissioned after a data node is recommissioned. It also shows that when a data node is recommissioned, the free space increases. A data node is recommissioned, in this scenario, if the free space reaches under 15 GB free and used space is above 80%. The second condition, free space under 5 GB and used percent under 80% is never fulfilled. The free space continues to go down even if the data nodes are recommissioned because it takes some time for Name node to take notice of the system change. This will happen in all strategy plans if the data node scaling conditions are satisfied. Figure 4 shows that the increase of Node Managers will decrease for this program the CPU and memory usage.

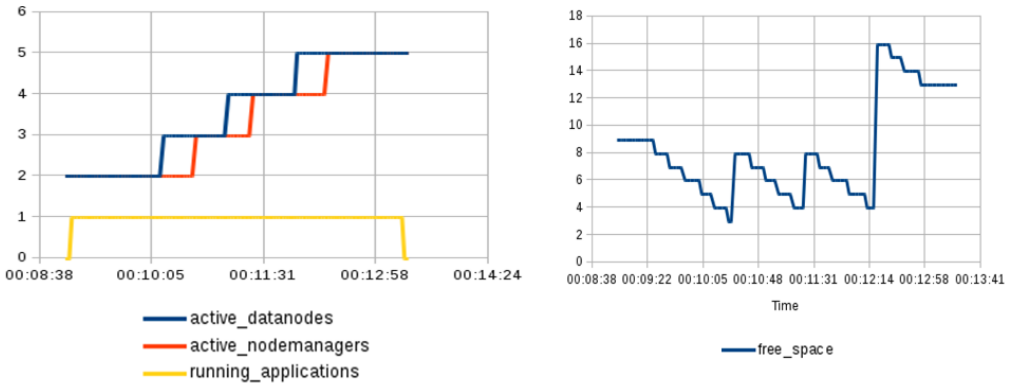

FIG. 3. Node manager recommissioned after data node

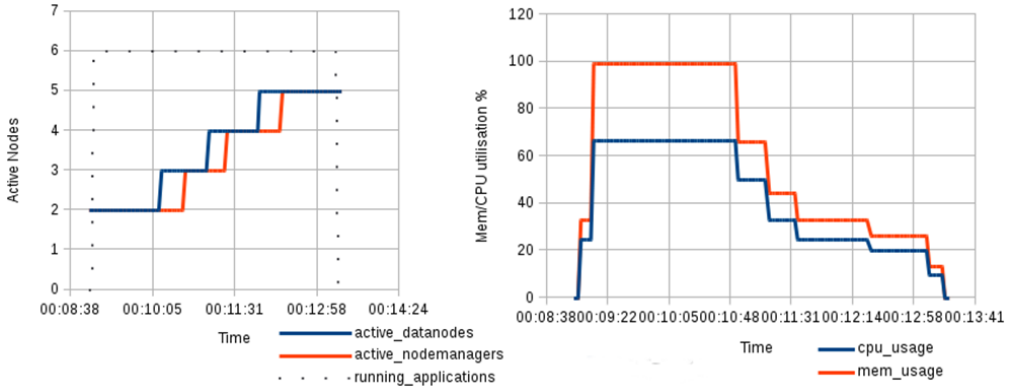

FIG. 4. Scaling out

Scaling Strategy 2. The following setup is employed: 2 Data Nodes, 2 Node Managers, free space 9.9 GB, free space threshold 11 GB if used space percent is above 80% and 5 GB if less, node manager and data node add cooldown 30 s, recommission node manager CPU threshold 50% and memory 60%.

Figure 5 shows how the CPU and memory usage lowers as we recommission Node Managers.

Scaling Strategy 3. The following setup is employed: 2 Data Nodes, 2 Node Managers, free space 9.9 GB, free space threshold 11 GB if used space is above 80% and 5 GB if less, maximum Node Managers 8.

In this strategy plan, the node managers will increase at the maximum capacity, independent of the CPU and memory metrics.

As a conclusion, the strategy that utilizes the least amount of resources is scaling strategy 2. However, it can recommission more Node Managers than necessary. For this reason it is not recommended for this type of program.

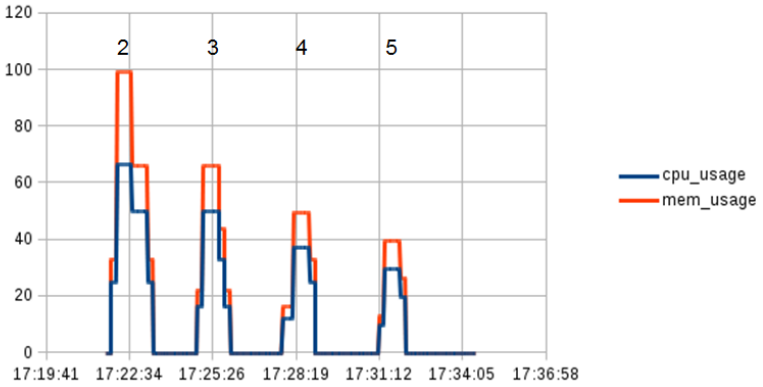

FIG. 5. Scaling out node managers

5.**2**.1. *Wordcount.* Setup: 2 Data Nodes, 2 Node Managers, free space 9.9 GB, free space threshold 11 GB if used space percent is above 80% and 5 GB if less, 8 maximum Node Managers

In the current setup, data node scaling operations are not needed, because the scaling conditions are not satisfied (the free space available does not lower down enough). As a consequence, applying scaling strategy 1 has no effect on the system.

There is little difference between scaling strategy 2 and 3, as it can be seen in 6.

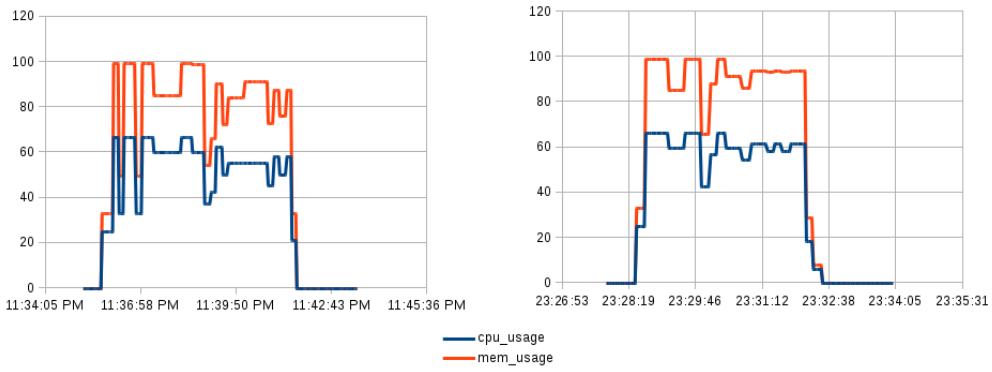

FIG. 6. CPU and Memory usage in scaling: Strategy 1 (left); Strategy 2 (right)

The difference is given by the time needed to gather the measurements, satisfying Node Manager thresholds and the overhead between two successive scaling operations in scaling strategy 2.

6. Lessons and Limitations

A data node is not recommended to be decommissioned if there are $dfs.replication - 1$ nodes in the decommissioning state, because there is a possibility that a certain file would not exist on any available nodes if the decommissioning operation is done

If there is no application running there must be an add prohibition for slave components. This option would not be so relevant if the application has a prediction component. In the current state, there is no way to know when a new application will start so the best action is to keep resource utilization to a minimum.

The reaction time of master components to system changes must be taken into consideration when the metrics are obtained, because some metrics may be compromised during this time interval and should not be taken into consideration.

There must be a time interval between two scaling operations applied on the same component type so not to overwhelm the system with requests and to leave time for Resource Manager to allocate those new resources, master components to notice the system changes and to collect enough metrics for calculated decision.

If a slave component is deleted, then the master component must be restarted. This is happening because Resource Manager and Name node does not run in high availability mode. A consequence is that there is an overhead time caused by restarting the master and the slave components, in order to determine them to resume their previous state. Luckily, if a component is added, there is no such restriction.

In Hadoop only one writer is allowed at a certain moment, but there can be many readers. This will limit the throughput for write operations, but will increase it for read operations.

Ambari Metrics Collector runs in embedded mode (default option), because the cluster has a small size (eight nodes). By running in distributed mode, metrics are stored in HDFS and therefore there is an additional network overhead that will limit the applications throughput. Additionally, the Name node restart will take longer because it has to index all the files in the HDFS. The metrics are obtained through Hadoop services master components: YARN's Resource Manager and HDFS's Name node. The same information can be obtained through the slave services: Data Node and Node Managers. The upside is that there is no overhead time to take notice of system changes (e.g. decommissioning a data node, recommissioning a Node Manager), but the downside is that the time for getting the metrics through increases with the growth of the slaves.

The time for getting a metric through Ambari REST is fairly long. In order to reduce the time of a regular Ambari REST get command, a partial

request is used to select and retrieve only the metrics from the master components that are important to decide what scaling operations should be done.

The small files are a huge problem for Hadoop, because it spends a lot of time managing their metadata information and as pointed [16] the memory usage is also high (63.53%). That happens because the metadata information is stored in the Name node memory [17]. Another issue for small files is that Name node restricts the number of files stored in the HDFS and according to [17], "accessing a large number of these files results in a bottleneck in Name node". Furthermore, high latency is expected when reading small files and the throughput falls below expectations.

7. Conclusions and future work

In this article we have argued for the importance of scalability in distributed systems, specifically vertical scalability and its implications. We have outlined the benefits of this approach and proposed a solution capable of automatic scaling in and out of a Hadoop cluster. We have described 3 scaling strategies and ran experiments on multiple types of workloads. The experiments demonstrate that such an approach is feasible and can integrate easily with other cluster components.

Although we achieved promising results, we intend to extend our experiments to larger clusters. The experiments will also involve multiple concurrent applications competing for the same resources. This will imply the development of a scheduling algorithm and the encryption of the transmitted data. The scheduling algorithm will need to employ a check-pointing mechanism in order to resume the applications once they are able to run.

Future work will also involve scaling services such as processing, messaging and storage. The application will also be transformed to allow for more abstract scaling expressions. In this sense the administrator will input QoS/SLA parameters (e.g data processing throughput or response time) and BigScale will adjust the cluster to those purposes. The underlying infrastructure will also receive horizontal scaling recommendations.

Currently, the application performs regular polling operations of the cluster metrics. As part of a modular solution, we will employ an event-based architecture to reduce the resulting overhead. Commissioning and decommissioning will be done in a parallel fashion, in order to speed up the short-term capability of the scaling process.

Acknowledgement

This work has been funded by University Politehnica of Bucharest, through the Excellence Research Grants Program, UPB - GEX. Identifier: UPB - EXCELENŢĂ - 2017 Data Analyzing in Real-time heterogeneous environments

REFERENCES

[1] *Garcia, Daniel F., G. Rodrigo, Joaquín Entrialgo, Javier Garcia, and Manuel Garcia.* "Experimental evaluation of horizontal and vertical scalability of cluster-based application servers for transactional workloads." In 8th International Conference on Applied Informatics and Communications (AIC'08), pp. 29-34. 2008.

[2] *Agrawal, Divyakant, Amr El Abbadi, Sudipto Das, and Aaron J. Elmore.* "Database scalability, elasticity, and autonomy in the cloud." In International Conference on Database Systems for Advanced Applications, pp. 2-15. Springer Berlin Heidelberg, 2011.

[3] *Mei, Lijun, Wing Kwong Chan, and T. H. Tse.* "A tale of clouds: paradigm comparisons and some thoughts on research issues." In Asia-Pacific Services Computing Conference, 2008. APSCC'08. IEEE, pp. 464-469. Ieee, 2008.

[4] *Anandhi, R., and K. Chitra.* "A challenge in improving the consistency of transactions in cloud databases-scalability." International Journal of Computer Applications 52, no. 2 (2012).

[5] Amazon EC2, [`https://aws.amazon.com/documentation/ec2/`

[6] *Vaquero, Luis M., Luis Rodero-Merino, and Rajkumar Buyya.* "Dynamically scaling applications in the cloud." ACM SIGCOMM Computer Communication Review 41, no. 1 (2011): 45-52.

[7] Google App Engine `https://cloud.google.com/appengine/docs/python/an-overview-of-app-engine`

[8] Microsoft Azure, `https://azure.microsoft.com/en-us/documentation/articles/best-practices-auto-scaling/`

[9] Ambari `https://cwiki.apache.org/confluence/display/AMBARI/Ambari`

[10] *Sefraoui, Omar, Mohammed Aissaoui, and Mohsine Eleuldj.* "OpenStack: toward an open-source solution for cloud computing."

[11] *Borthakur, Dhruba.* "HDFS architecture guide." HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf (2008): 39.

[12] *Vavilapalli, Vinod Kumar, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves et al.* "Apache hadoop yarn: Yet another resource negotiator." In Proceedings of the 4th annual Symposium on Cloud Computing, p. 5. ACM, 2013.

[13] *Dean, Jeffrey, and Sanjay Ghemawat.* "MapReduce: simplified data processing on large clusters." Communications of the ACM 51, no. 1 (2008): 107-113.

[14] *Wadkar, Sameer, and Madhu Siddalingaiah.* "Apache ambari." In Pro Apache Hadoop, pp. 399-401. Apress, 2014.

[15] *Hunt, Patrick, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed.* "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In USENIX Annual Technical Conference, vol. 8, p. 9. 2010.

[16] *Liu, Xuhui, Jizhong Han, Yunqin Zhong, Chengde Han, and Xubin He.* "Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS." In 2009 IEEE International Conference on Cluster Computing and Workshops, pp. 1-8. IEEE, 2009.

[17] *Chandrasekar, S., R. Dakshinamurthy, P. G. Seshakumar, B. Prabavathy, and Chitra Babu.* "A novel indexing scheme for efficient handling of small files in hadoop distributed file system." In Computer Communication and Informatics (ICCCI), 2013 International Conference on, pp. 1-8. IEEE, 2013.