On the Elasticity of NoSQL Databases over Cloud Management Platforms

Ioannis Konstantinou, Evangelos Angelou, Christina Boumpouka,
Dimitrios Tsoumakos, Nectarios Koziris
Computing Systems Laboratory, School of Electrical and Computer Engineering
National Technical University of Athens, Greece
{ikons, eangelou, christina, dtsouma, nkoziris}@cslab.ece.ntua.gr

ABSTRACT

NoSQL databases focus on analytical processing of large scale datasets, offering increased scalability over commodity hardware. One of their strongest features is elasticity, which allows for fairly portioned premiums and high-quality performance and directly applies to the philosophy of a cloud-based platform. Yet, the process of adaptive expansion and contraction of resources usually involves a lot of manual effort during cluster configuration. To date, there exists no comparative study to quantify this cost and measure the efficacy of NoSQL engines that offer this feature over a cloud provider. In this work, we present a cloud-enabled framework for adaptive monitoring of NoSQL systems. We perform a study of the elasticity feature on some of the most popular NoSQL databases over an open-source cloud platform. Based on these measurements, we finally present a prototype implementation of a decision making system that enables automatic elastic operations of *any* NoSQL engine based on administrator or application-specified constraints.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed Systems

General Terms

Design, Performance

Keywords

Cloud Computing, NoSQL, HBase, Cassandra, Riak.

1. INTRODUCTION

Computational and storage requirements of applications such as business intelligence and social networking over petabyte datasets have pushed sql-like centralized databases to their limits [8]. This led to the development of horizontally scalable, distributed non-relational data stores, called NoSQL databases, such as Google's Bigtable [9] and its open-source implementation HBase [5] and Facebook's Cassandra [12]. NoSQL systems exhibit the ability to store and index arbitrarily big data sets while enabling a large amount of concurrent user requests. They are perfect candidates for IaaS clouds such as Amazon's EC2 [1] or its open-source alternative, OpenStack [6]: NoSQL admins can utilize the cloud API to throttle the number of acquired resources (i.e., number of virtual machines – VMs and storage space) according to application needs.

This is highly-compatible with NoSQL stores that offer *elasticity* and *sharding*. The former refers to the ability to expand or contract resources in order to meet the exact demand. The latter refers to the horizontal partitioning over a shared-nothing architecture. It is obvious that these two properties (henceforth referred to as *elasticity*) are intertwined: as computing resources grow and shrink, data partitioning must be done in such a way that no loss occurs and the right amount of replication is conserved.

Many NoSQL systems (e.g., [5, 12, 7]) claim to offer adaptive elasticity according to the number of participant commodity nodes. Nevertheless, the "throttling" is usually performed manually, making availability problems due to unanticipated high loads not infrequent (e.g., the recent Foursquare outage [3]). Adaptive frameworks are offered by major cloud vendors as a service: Amazon's SimpleDB [2] and Google's AppEngine [4] are proprietary systems offering (virtually) unlimited processing power and storage. However, these services run on dedicated servers, their architecture is not publicly documented, their cost is prohibitive and their performance is questionable [11].

Recent works [11, 10] have dealt with the performance of various analytics platforms, without dealing with elasticity in virtualized cloud resources. The studies presented in [13, 15] deal with this feature but do not address NoSQL databases. Thus, although both NoSQL and cloud infrastructures are inherently elastic, there exists no actual study to report how effective this is in practice. To the best of our knowledge, there also exists no actual system that combines these two technologies to offer automated NoSQL cluster resize according to dynamic workload changes.

Having reviewed the most popular NoSQL solutions and the OpenStack IaaS [6], our contributions are the following: We provide a generic control module that monitors NoSQL clusters, We identify how each metric of interest (CPU, RAM, etc) varies under various workload types and rates. We document the costs and gains after a cluster resize. We register the performance gains when increasing the cluster size in varying workloads. We demonstrate the applicability of our framework by presenting a prototype implementation that allows for adaptive and automatic cluster resize. To show our framework's modularity, we incorporate three popular NoSQL implementations, HBase [5], Cassandra [12]

CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.

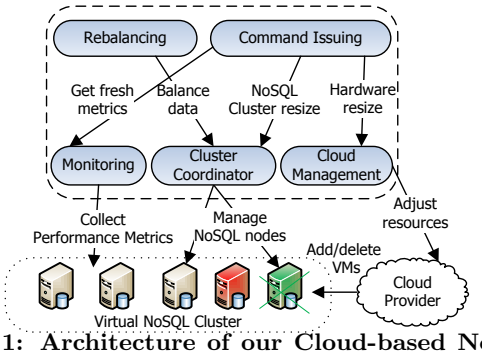

Figure 1: Architecture of our Cloud-based NoSQL elasticity-testing framework.

and Riak [7] that support elasticity and we utilize YCSB [10], Yahoo's open-source client. Our findings are presented more thoroughly in the following technical report[1].

2. ARCHITECTURE

Our elasticity-testing framework, an open-source project[2] with over 2K lines of Python code, features an architecture that is illustrated in Figure 1. The *Command Issuing* module is used to initiate a cluster resize operation. It interacts with the *Cloud Management* module that contacts the cloud vendor to adjust the cluster's physical resources by releasing or acquiring more virtual machines. The *Rebalancing* module makes sure that newly arrived nodes join the cluster contributing an equal share of the work. The *Cluster Coordinator* module executes higher level add, remove and rebalance commands according to the particular NoSQL system used. Finally, the *Monitoring* module maintains up-to-date performance metrics that collects from the cluster nodes. Below we describe the modules in detail:

Command Issuing Module: This is the 'coordinator' module. In the current implementation phase, this module requests addition or removal of a number of VMs using the Cloud Management and Cluster Coordinator modules.

Monitoring Module: Our system receives data from Ganglia [14], a scalable distributed system monitoring tool that allows the user to remotely collect live or historical statistics (such as CPU load averages, network, memory or disk space utilization) through its XML API and present them through its web front-end via real-time dynamic web pages. Apart from general operating-system statistics, Ganglia may also gather NoSQL performance metrics such as the current number of open client threads, number of served operations per second, etc.

Rebalancing Module: The rebalancing module is activated after a newly arrived virtual machine from the cloud vendor has successfully started (i.e., has booted and received a valid IP). When this happens, the module executes a "global rebalance" operation, in which client requests are spread equally among the cluster nodes according to the specific NoSQL implementation and semantics.

Cloud management: Our system interacts with the cloud vendor using euca-tools, an Amazon EC2 compliant REST-based client library. The command issuing module interacts with this module when it commands for a resize in the physical cluster resources, i.e., the number of running VMs. The use of euca-tools guarantees that our system can be deployed in Amazon's EC2 or in any EC2-compliant IaaS cloud. We have created an Amazon Machine Image (AMI) that contains pre-installed versions of the supported NoSQL systems along with the Ganglia monitoring tool.

Cluster coordinator: The coordination of the remote VMs is done with the remote execution of shell scripts and the injection of on-the-fly created NoSQL specific configuration files to each VM. A higher level "start cluster", "add NoSQL node" and "remove NoSQL node" command is translated in a workflow of the aforementioned primitives. Our framework implementation makes sure that each step has succeeded before moving to the next one.

For instance, the Command Issuing module requests an "add virtual machine" command using the euca-tools API and waits until it is started and has been assigned with an IP. After this, the Cluster Coordinator creates the appropriate configuration scripts on-the-fly, transfers them to the new VM, and remotely starts the NoSQL service along with the Ganglia tool. The Rebalancer module inserts the node to the cluster and rebalances client requests among the server nodes. Our framework currently incorporates HBase, Cassandra and Riak. Yet, the system is extensible enough to include more engines that support elastic operations by implementing the system's abstract primitives in the Cluster Coordinator module and by including the system's binaries to the existing AMI virtual machine image. The precooked AMI is available for download from the project's web site.

3. EXPERIMENTAL RESULTS

Our experimental setup consists of an OpenStack private cluster of 16 worker nodes and a single cluster controller. We were allocated enough resources for a cluster of 20 client VMs (load generators) and 28 server VMs (each with 4 vCPUs and 8GB RAM). First, we identify the affected performance metrics under heavy load, second, we measure the performance gain/loss after cluster resize for various workloads and resize choices and finally we present a prototype, fully automated system setup where resources are adaptively resized according to user-defined policies.

Clients and workloads used: We utilize fixed HBase (v. 0.20.6) Cassandra (v. 0.7.0 beta) and Riak (v. 0.14.0) initial 8 node clusters with default configurations which are loaded with 20M 1KB objects (i.e., 20GB of plain raw data) by utilizing the YCSB [10] load function. We use 4 workloads (namely UNIFORM_READ, ZIPFIAN_READ, UNIFORM_UPDATE and UNIFORM_RMW) with fixed query arrival rates λ in order to better understand the behaviour of the databases for different types of load. These correspond to uniform and zipfian random reads, and uniform random updates and read-modify-writes respectively.

For the 8-node clusters, we first identified the maximum sustained query throughput in a simple UNIFORM_READ workload. The HBase, Cassandra and Riak clusters achieved a maximum throughput of 80, 13 and 10 Kreqs/sec for a λ of 80, 20 and 10 Kreqs/sec respectively. Cluster CPU usage increased as throughput increased, reaching a 50% and 75% usage in their maximum throughput. For higher λ rates (up to 280 Kreqs/sec) query throughput remained the same while query latencies increased linearly for HBase and Cassandra, with HBase being constantly faster than Cassandra. Riak servers became unresponsive above 10 Kreqs/sec, therefore we did not include Riak in the following experiments. For the zipfian read and uniform update workloads, with a λ of 180 Kreqs/sec, Cassandra had an average CPU usage of 68% and 55% respectively, whereas HBase had 55%

[1] http://www.cslab.ntua.gr/~ikons/elastic_nosql.pdf
[2] http://tiramola.googlecode.com

Table 1: Data rebalancing effect for a 8+8 resize

Metric \ Cluster	HBase		Cassandra		Riak
	Reb	No	Reb	No	Reb
Completion time (min)	98	5	665	5	150
Data moved (GB)	22.5	-	87.7	-	44.8
Througput (Kreqs/s)	154.5	129.6	18.3	14.9	18
Avg. Latency (s)	0.7	1.1	7.1	9.3	0.2

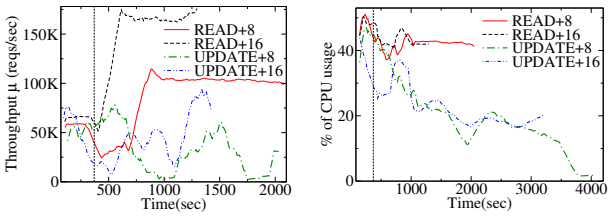

Figure 2: Query throughput and CPU usage for an HBase cluster of 8 nodes after adding 8 and 16 more nodes for the UNIFORM_READ and UNIFORM_UPDATE workload with $\lambda = 180$ Kreqs/sec

and 45%. Average query throughput was 77 and 8 Kreqs/sec respectively for the zipfian read, whereas for the update it was 30 and 12 Kreqs/sec.

Cluster resize performance measurements: Our first concern on the costs and gains of a resize operation relates to the rebalancing of the database data. Since data rebalancing is by itself a resource intensive procedure, we only perform node additions and data migration in an idle cluster for this case. This scenario is valid, since data rebalancing is usually scheduled for off-peak time periods. Nevertheless, in our experiments conducted with extra client workload during data rebalance, all systems exhibited erratic behaviour. In Table 1 we present our results. We have started an 8-node cluster in each case and added 20M tuples. We then expanded each cluster by adding 8 more nodes and applied a UNIFORM_READ workload with a $\lambda = 180$ Kreqs/sec ("No" column for every database - for Riak that rebalances data automatically as soon as a new node joins the ring, this is not applicable). After this, we manually rebalanced data in HBase and Cassandra. When data rebalancing finishes, we applied the same workload $\lambda = 180$ Kreqs/s for HBase, Cassandra and 18 Kreqs/s for Riak (Riak could not operate with a higher workload). The results for this setup are presented in the "Reb" column.

The data rebalancing costs far outweigh its benefits for the cases of HBase and Cassandra. For HBase a net gain of about 20% in throughput was achieved compared to a non-rebalanced 16 node cluster, which could be easily offset by adding two extra nodes without using data rebalancing. The results were better for Cassandra with a net gain of 22% for the average throughput. In terms of latency, similar performance benefits were achieved (33% and 23% for HBase and Cassandra respectively). The data moved during data rebalancing for HBase is 25% of the entire dataset. In Cassandra's case the whole dataset was moved. HDFS's centralized balancer is more advanced than Cassandra's decentralized balancer. Riak rebalancing is also a costly operation (43% of the dataset was transferred). Given the large time costs for data rebalancing operations (93, 660 and 150 minutes in HBase, Cassandra and Riak respectively), and the fact that simple caching can do the work (see the following analysis) we conclude that gains do not outweigh costs.

We continue our evaluation with node additions in an initial 8-node cluster without data rebalancing for HBase and Cassandra. We utilize a high load of $\lambda = 180$ Kreqs/sec for two workload types: UNIFORM_READ and UNIFORM_UPDATE (referred to as READ and UPDATE henceforth). For each workload and database combination we perform an addition of 8 and 16 nodes. The cluster resize occurs at about $t = 370$ sec.

Figure 2 shows the HBase results. Legends refer to the workload type along with the resizing action (e.g., READ+8 represents a read workload with an 8-node resize). Regarding mean query latency (graph omitted due to lack of space), adding nodes during READ loads has a transient negative effect, attributed to the HMaster table region reassignment. Clients cache region locations which however change during cluster resizing. As a result, latency increases due to client cache misses. In the updates we notice an oscillation due to the compaction and caching mechanisms of HBase. Incoming data is cached in memory (i.e., low latency) but when the memory is full and a I/O flush occurs with a compaction, the latency is increased. Throughput μ increases in read workloads: adding 16 nodes results in about 170 Kreqs/sec in the steady state and 100 Kreqs/sec for the READ+8 case (compared to about 55K before additions). More servers are able to simultaneously handle more requests that results in a higher throughput (roughly doubling and tripling it respectively). Although items are not actually transferred, this speed is due to the caching effect of the RegionServers. Regarding updates, since they are I/O-bound operations, μ is not significantly altered.

The final graph reports the aggregate cluster CPU usage. In the read workloads we notice that the initial load of around 55% is reduced to around 42% in both cases since new nodes immediately start handling incoming queries. The addition of 16 against 8 nodes does not result in a further decrease in the average CPU, as the load is still large enough for all servers to contribute. The extra 8 nodes make a difference in terms of throughput, as shown in the second graph. In the update workloads, we notice that in both experiments the initial CPU load continues to drop until run completion since servers freeze incoming requests until the updated regions are flushed to the file system.

In Figure 3 we present the Cassandra results. The first graph presents query latency. In both READ cases, we notice that the latency drops from 22 secs to 10 secs and 8 secs respectively. New servers are assigned half of the data partitions of existing servers, they cache portions of their data and answer queries on their behalf. The larger the resize, the bigger the decrease in latency. The same hold for the update workloads. We notice here that writes are faster than reads, due to the weak consistency model followed by Cassandra. Query throughput μ shows a similar linear trend in reads and updates. Extra servers immediately join the p2p ring and take portions of the applied workload. Finally, in the third graph of Figure 3 we present the CPU usage. In the read case adding 8 nodes decreases CPU usage to around 60%, whereas adding 16 servers decreases CPU load to around 50%. The same trend holds for the updates.

Both NoSQL systems take advantage of extra nodes: HBase exhibits very fast concurrent reads compared to Cassandra. On the other hand, Cassandra is more efficient with object updates because of the weak consistency model. Finally, we notice that Cassandra does not exhibit a negative transient effect when new nodes enter the ring. Its decentral-

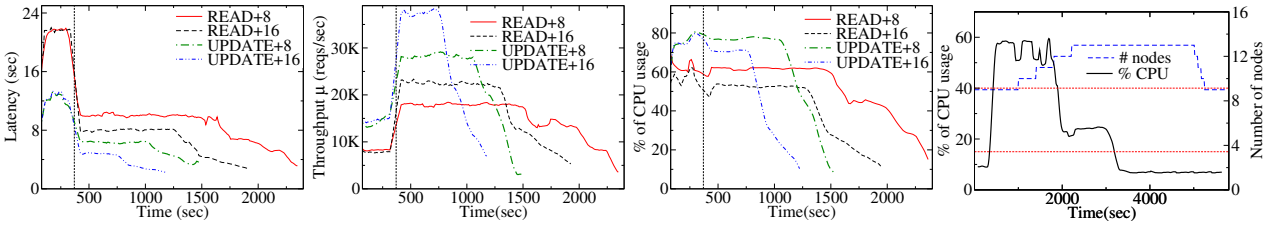

Figure 3: Query latency, query throughput and CPU usage per time for a Cassandra cluster of 8 nodes after adding 8 and 16 more nodes for the UNIFORM_READ and UNIFORM_UPDATE workload with a query rate of $\lambda = 180$ Kreqs/sec

Figure 4: CPU usage for an HBase cluster of 8 nodes under dynamic load changes.

ized nature allows for a transparent cluster resize, whereas in HBase the HMaster coordinates the whole procedure.

Elasticity-provisioning prototype: We now present some initial results achieved using our framework to deploy a fully automatic cluster resize system. We utilize a long running read-heavy load, for a total load of 9 Kreqs/sec on an 8-node HBase cluster. After 7 min, an additional high load of 180 Kreqs/sec is created for a short period of time. Utilizing the experience from our previous experiments, we trigger node additions when CPU usage is over 40%, for *any* one server on the cluster and we remove nodes when the CPU usage is 15% or lower for *all* nodes in the cluster. To avoid oscillations, we utilize a 5 minute running average of the load. In Figure 4 we show our results. The low initial load stresses the servers enough to avoid a node removal but not enough for a node addition. When load increases, CPU increases dramatically and 10 minutes are needed for the new node to be fully operational and monitored. The high load requests complete before the addition of the fourth extra node. At this point the total load decreases significantly, but not substantially for a new node addition. The system continues its normal operation, since the medium load is just enough to prevent node removal. Once the load drops under the lower threshold, node removal proceeds. Node removal is much faster than node addition, thus the system returns to its original size in about 5 minutes. CPU usage does not drop during single node addition, as there are not enough nodes added to accommodate the high load.

4. DISCUSSION AND CONCLUSIONS

In the following section, we argue on the design choices, we offer recommendations based on our experience in setting up this system and we conclude our work.

Concerning monitoring, the selected metrics are important: We showed that we can accurately distinguish the system's critical state using passive, general purpose metrics. Finally, Ganglia's rich set of available metrics and ease of use enable the automatic decision making module to take accurate and prompt decisions.

Regarding region rebalancing, in HBase it is performed automatically when new nodes are added or removed from the cluster. HBase is extremely elastic, as all new nodes can quickly assume load, increasing the cluster's performance in very short time, as reported. Conversely, Cassandra does not split data into regions of equal size. It reassigns regions on a per node basis, i.e., region rebalancing is performed in node pairs between newly arrived and previously existing ones by splitting the key-space in half. Riak divides data space into partitions and as new nodes are added, they claim an equal slice of partitions. However, the fact that the cluster turns unresponsive for a throughput higher than what its nodes can handle, prevents us from examining its elasticity under high load. Concerning data rebalancing, it should only be performed when the system administrator can accurately predict a constant load for a large amount of time, that is in the order of days. In this case, even if the performance is affected for a while, the overall gain justifies this transient rebalancing cost. On the contrary, variable and unpredictable short-term load does not justify such an expensive of operation, and should be avoided.

In this work we quantified and analyzed the costs and gains of various NoSQL cluster resize operations, utilizing three popular NoSQL implementations. HBase is the fastest and scales with node additions (only for reads though); Cassandra performs fast writes and scales also, without any transitional phase during node additions; Riak was unresponsive in high request rates, could scale only at lower rates but rebalances automatically; all three achieve small gains from a data rebalance, provided they are under minimal load. Based on our findings, we offered a prototype implementation of our automatic cluster resize module, that matches the number of provisioned resources against the total demand and the application expert's rules of required operation. Our open-source implementation can provide a good basis on which numerous applications can test their adaptivity at very-high scale.

5. REFERENCES

[1] Amazon EC2. http://aws.amazon.com/ec2/.
[2] Amazon SimpleDB. http://aws.amazon.com/simpledb/.
[3] Foursquare Outage Post Mortem. http://bit.ly/gjA2IK.
[4] Google AppEngine. code.google.com/appengine.
[5] HBase Homepage. http://hbase.apache.org.
[6] OpenStack. http://openstack.org.
[7] Riak Homepage. http://wiki.basho.com/.
[8] D. J. Abadi. Data Management in the Cloud: Limitations and Opportunities. *IEEE Data Eng. Bull*, 32(1):3–12, 2009.
[9] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.
[10] B. F. Cooper et al. Benchmarking Cloud Serving Systems with YCSB. In *ACM SOCC*, 2010.
[11] D. Kossmann et al. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. In *SIGMOD*, 2010.
[12] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS*, 2010.
[13] P. Marshall, K. Keahey, and T. Freeman. Elastic Site: Using Clouds to Elastically Extend Site Resources. In *CCGRID*, 2010.
[14] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7):817–840, 2004.
[15] P. Xiong et al. Intelligent Management of Virtualized Resources for Database Systems in Cloud Environment. In *ICDE*, 2011.