Automatic Scaling of Selective SPARQL Joins Using the TIRAMOLA System

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

ABSTRACT

Modern cloud infrastructures based on virtual hardware provide new opportunities and challenges for developers and system administrators alike. Most notable is the promise of resource elasticity, whereby the infrastructure can increase or decrease in size based on demand. Utilizing elastic resources, applications can provide better quality of service and reduce cost by only paying for the required amount of resources. In this work, we extensively study the performance of some popular NoSQL databases over an elastic cloud infrastructure. NoSQL databases focus on analytical processing of large scale datasets, offering increased scalability over commodity hardware. We then proceed to describe *TIRAMOLA*, a cloud-enabled framework for automatic provisioning of elastic resources on *any* NoSQL platform. Our system administers cluster resources (VMs) according to user- or application-specified constraints through an expandable monitoring and command-issuing module. Users can easily modify resizing policies, based on application-specific metrics and thus fully utilize the elasticity of the underlying infrastructure. As a realistic use-case, we apply this framework on top of a fully distributed RDF store backed by an elastic NoSQL database. Letting *TIRAMOLA* manage the number of committed resources results in automated cluster resize actions and throughput maximization, while application experts need only provide simple elasticity rules.

1. INTRODUCTION

Cloud Computing has been receiving an increasing amount of attention from both the industry and academia. On-demand and pay-as-you-go access to computational and storage resources that reside in distant data centers is a very attractive business model, especially for small to medium-sized enterprises (SMEs) and start-ups that require straightforward access to scalable hardware and software infrastructures without the administrative costs. This model drives the proliferation of cloud platforms that provide infrastructure as a service (IaaS) such as Amazon's elastic compute cloud (EC2) or its open-source alternatives such as Eucalyptus and OpenStack. The use of cloud APIs allows administrators to resize virtual infrastructures, in terms of the number of virtual machines, the number of virtual cores, memory and storage space etc. so as to satisfy their applications' needs.

The computational and storage requirements of applications such as web analytics, business intelligence and social networking over tera- (or even peta-) byte datasets have pushed on premise infrastructures to their limits and drive the adoption of the Infrastructure as a Service (IaaS) paradigm over the Cloud[26].

These needs have led to the development of horizontally scalable, distributed non-relational data stores, called NoSQL databases. Google's Bigtable [12] and its open-source implementation HBase [5], Amazon's Dynamo [15], Facebook's Cassandra [24], and LinkedIn's Voldemort [7] are a representative set of such systems. NoSQL systems exhibit the ability to store and index arbitrarily big data sets while enabling a large amount of concurrent user requests.

Data generated through the integration and interlinking of diverse Web sources exhibit such features. These datasets are stored and queried using the RDF [8] framework and the SPARQL [32] language respectively in several Semantic Web applications. The ultimate goal is to utilize and combine these datastores in order to extract knowledge through Linked Data processing. It is clear that this extraction can be achieved by utilizing scalable data management techniques and infrastructures. A number of horizontally scalable RDF data-stores (e.g., [11, 14, 20]) have been recently introduced towards this direction.

NoSQL datastores' ability to scale horizontally makes them perfect candidates for cloud platforms that provide infrastructure as a service (IaaS) and applications based on them will exploit their characteristics: Scalability in processing big data is possible through *elasticity* and *sharding*. The former refers to the ability to expand or contract dedicated resources in order to meet the exact demand. The latter refers to the horizontal partitioning over a shared-nothing architecture that enables scalable load processing. 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, 15, 24, 7, 9]) 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

SWIM 2012, May 20, 2012, Scottsdale, Arizona, USA.

Foursquare outage [19]). Adaptive frameworks are offered by major cloud vendors as a service through their infrastructure: Amazon's SimpleDB [2], Google's AppEngine [4] and Microsoft's SQL Azure [10] are proprietary systems provided through a simple REST interface offering (virtually) unlimited processing power and storage. However, these services run on dedicated servers (i.e., no elasticity from the vendor's point of view), their internal design and architecture is not publicly documented, their cost is sometimes prohibitive and their performance is questionable [23].

Recent works have provided interesting insights over the performance and processing characteristics of various analytics platforms (e.g., [23, 30, 13]), without dealing with elasticity in virtualized resources, which is the typical case in cloud environments. The studies presented in [16, 27, 35] deal with this feature but do not address NoSQL databases, while [25] is file-system specific. Finally, proprietary frameworks such as Amazon's CloudWatch [1] or AzureWatch [3] do not provide a rich set of metrics and require a lot of manual labor to be applicable for NoSQL systems.

Thus, although both NoSQL and cloud infrastructures are inherently elastic, there exist no detailed and extended studies to report how effective this is in practice, at least over architecturally different engines. Our work aims to bridge this gap between individual implementations and practice. Besides the inherent elasticity constraints NoSQL engines exhibit [21], our task is to deploy a distributed framework that allows (in a customizable and automated manner) NoSQL platforms to expand or contract their resources by using a cloud management platform. In this work, we make the following tangible contributions:

- We document the costs and gains after a cluster resize and a data rebalance operation. Using both client and cluster-based metrics, we register the performance gains when increasing the size of the cluster under varying generic workloads (with and without rebalancing operations).
- We present *TIRAMOLA*, an open-source prototype implementation of our framework that performs distributed monitoring at various granularities of a NoSQL platform and automates the cluster resize process according to user-defined policies.
- As a case study, we apply *TIRAMOLA* on a distributed RDF datastore [29] which supports the storage and indexing of RDF data along with the efficient resolution of join queries. Through the application of a varying workload and the use of a simple policy, we demonstrate *TIRAMOLA*'s ability to perform correct cluster resize operations as the cluster increases and decreases its size in order to maximize the reported throughput.

The rest of the paper is organized as follows: Section 2 presents TIRAMOLA's architecture; Section 3 outlines our results; Section 4 presents related work and Section 5 concludes our work by briefly highlighting our findings.

2. ARCHITECTURE

TIRAMOLA, our elasticity-testing framework is an open-source project [1] with over 2K lines of Python code. Its architecture is depicted in Figure 1. It consists of a number of configuration, management and monitoring modules which interact with both the IaaS and PaaS layers of a shared-nothing distributed application sitting on top of a

[1]http://tiramola.googlecode.com

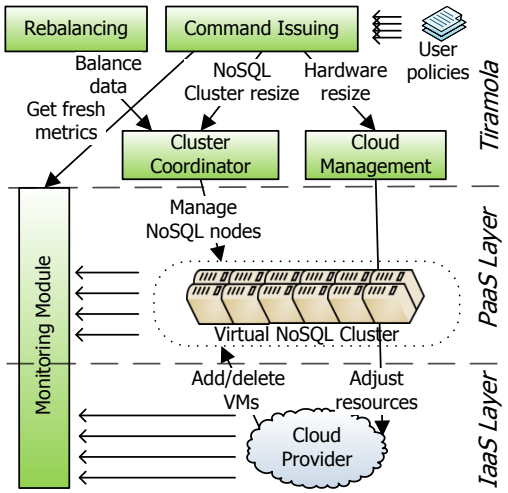

Figure 1: The *TIRAMOLA* NoSQL elasticity-testing framework.

cloud platform. More specifically, it comprises of the higher-level Command Issuing and Rebalancing modules which are responsible for initiating cluster resize operations and ensuring equal load distribution respectively. The lower level components of *TIRAMOLA* are the Cloud Management, the Cluster Coordinator and the Monitoring module. The *Cloud Management* module interacts with the cloud vendor's IaaS level to adjust the cluster's physical resources by releasing or acquiring more virtual machines. The *Cluster Coordinator* executes higher level add, remove and rebalance commands according to the particular NoSQL system used on the PaaS cloud level. Finally, the *Monitoring* module collects fresh and up to date metrics both from the application (PaaS) and the infrastructure (IaaS) layer. In the following we describe in detail the aforementioned modules.

The *Command Issuing* module coordinates the entire resource adjustment process. It currently requests addition or removal of a number of VMs using the *Cloud Management* and the *Cluster Coordinator* modules. The user can define a set of policies which can trigger different cluster resize events. The policies are based on metrics retrieved by the *Monitoring* module. Their complexity can vary from simple threshold based policies (e.g., trigger node removals when the cluster's CPU usage is very low for a long time) to more sophisticated application specific policies (e.g., trigger node additions when there is a sudden increase in the mean query response time). Currently, this module requests addition or removal of a number of VMs using the Cloud Management and Cluster Coordinator modules.

TIRAMOLA's *Monitoring* module is designed and deployed in order to gather both generic IaaS related and specific PaaS related metrics. Currently, it receives data from Ganglia [28], a scalable distributed system monitoring tool that allows the user to remotely collect live or historical IaaS 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 PaaS related metrics such as the current number of open client threads, number of served operations per second, etc.

The *Rebalancing* module is activated after successful *TI-*

RAMOLA Cloud Issuing commands (i.e., a newly requested VM 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.

TIRAMOLA's *Cloud management* module interacts with the cloud vendor using the well known 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. Our cloud management platform is a private OpenStack [6] installation. 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.

The *Cluster coordinator* module issues higher level PaaS management commands after the requested infrastructure has been successfully reserved and initialized by the *Cloud Management* module. This coordination is done with the remote execution of shell scripts and the injection of on-the-fly created NoSQL specific configuration files to each VM. Higher level "start cluster", "add NoSQL node" and "remove NoSQL node" commands are translated in a workflow of primitive remote commands and configuration file transfers. Our framework implementation executes health checks and makes sure that each step has succeeded before moving to the next.

The following scenario depicts *TIRAMOLA*'s module collaboration during a cluster resize: the *Command Issuing* module identifies an overloaded situation by utilizing info from the *Monitoring* module and 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. Alternatively, when the decision dictates for node deletion, the *Cluster Coordinator* is instructed to remove it from the cluster calling the "terminate instance" command of the *Cloud Management*. Data loss during node removal is avoided with the use of NoSQL data replication. In this case, NoSQL systems transparently replicate the removed data in order to maintain the replication factor.

Our framework currently incorporates two popular NoSQL systems that implement rebalancing operations: HBase and Cassandra. 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 virtual machine image is available for download from the project's web site.

3. EXPERIMENTAL RESULTS

Our experimental setup consists of an OpenStack private cluster of 9 VM containers with one machine also serving as the Cloud endpoint, scheduler, storage and networking cluster controller. Each VM container has 2 6-core Intel Xeon®CPUs with Hyperthreading at 2.67GHz, 48 GB of RAM (with disabled page swapping) and two 2TB disks setup with RAID 0 for maximum disk I/O. Thus the cluster can support a total of 108 small VMs.

Each of our server VMs has a 4 virtual core processor with 8GB of RAM and 300GB of storage space. It therefore corresponds to a type medium Amazon VM. Cluster peers store their data into their root file system, i.e., no external Elastic Block Storage (EBS) is used. The versions of Hadoop and Ganglia used are 1.0.1 and 3.1.2 respectively, both in their default configuration.

Clients and workloads used: We utilize fixed HBase (v. 0.20.6) and Cassandra (v. 0.7.0 beta) initial cluster sizes of 8 nodes which are loaded with 20 million objects (i.e., 20GB of plain raw data, since each item takes up 1KB) by utilizing the YCSB [13] load function. Other standard benchmarks such as TPC-W focus mainly on relational databases, and we have opted for analytical rather than transactional database load, given the scope of NoSQL databases. Every database is configured with a replication factor of 3, which in the case of Cassandra and HBase results in a total database size of about 60GB and 90GB respectively (HBase uses more metadata per record). Since HBase and Cassandra are implemented in Java and they were setup for production mode, a generous heap space of 6GB was supplied. For the most part, all database systems were setup using their default settings, as presented in their online manual pages. The only deviation from this rule is Cassandra's `auto_bootstrap` parameter, which was set to `false`, as it effectively prevents adding more nodes to the cluster ring than the number of already participating nodes.

The default workload for the YCSB tool is random uniform read, with varying λ, where λ is the number of active threads of the YCSB tool. YCSB uses two parameters, the number of threads per client and the target number of operations per second they will try to achieve. Consequently, λ defines the number of concurrent pending operations at every time point. In our experiments, we pose a sufficient number (2M to 10M) of simple get queries that collectively cover a significant part of the dataset (10% to 50%).

YCSB comes with a set of workloads that simulate real-world conditions. However we use 4 consistent workloads (namely UNIFORM_READ, ZIPFIAN_READ, UNIFORM-_UPDATE and UNIFORM_RMW) in order to better understand the databases' behaviour for different types of load. These are simple (i.e., not composite) workloads, where all operations are of the defined type, that is uniform random reads, zipfian random reads, uniform random updates and uniform random read-modify-writes respectively.

Our evaluation conforms to the following methodology: First, we identify the cost and performance gain/loss cost after cluster resize for various workloads and resize choices. Then, given our findings, we present a prototype, fully automated system where the command issuing module monitors the appropriate performance metrics and adaptively adds/removes cluster resources in order to keep the cluster within some user-defined limits. Finally, we present a prototype system based on a SPARQL application backed by HBase, and evaluate our elasticity-providing system's implementation.

3.1 Cluster resize performance measurements

Data rebalancing costs: Our primary concern on the costs and gains of a cluster resize operation relates to the rebalancing of the database data. As data rebalancing is a resource intensive procedure, we perform node additions and

Table 1: Completion time, total moved data, final average query throughput and latency for a 8+8 node cluster resize operation in HBase and Cassandra with and without data rebalancing

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

data migration only on an idle cluster, that is without applying extra client load. This is a valid scenario because data rebalancing is normally scheduled during off-peak time periods. Moreover, in experiments conducted with extra client workload during data rebalancing, both systems exhibited erratic behaviour, and never achieved the throughput or latency performance of the original cluster before the resize operation. Even worse, DB clients received disconnects and inconsistencies, resulting in a significant number of exceptions.

In Table 1 we present our results for a primary examination of the costs and gains of data rebalancing for HBase and Cassandra. Beggining with an 8-node cluster in each case, we insert 20M tuples. Following data insertion, we expand each cluster with 8 more nodes, let the systems stabilize without performing data rebalancing and apply a UNIFORM_READ workload with a $\lambda = 180$ Kreqs/sec load. These results are presented in the "No" column for every database. Afterwards, a manual data rebalancing operation is performed in HBase (through the HDFS balancer[2]) and Cassandra (through `loadbalance` commands[3]). Once DB data are balanced, we apply a UNIFORM_READ workload with a $\lambda = 180$ Kreqs/s for HBase and Cassandra. The results for this setup are depicted in the "Reb" column for each database.

From Table 1, we can deduce that the data rebalancing costs far outweigh its benefits for both HBase and Cassandra. For HBase, a net gain of about 20% in throughput (154.5 Kreqs/sec balanced vs 129.6 Kreqs/sec not balanced) was achieved compared to a non-rebalanced 16 node cluster. In Cassandra's case, the results were better with a net gain of 22% for the average throughput (18.3 Kreqs/sec vs 14.9 Kreqs/sec for a non balanced 16 node cluster). 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 are roughly 22 GB or 25% of the entire dataset. In Cassandra's case, much more data are moved (87.7 GB in total), which translates to the whole dataset, as HDFS's centralized balancer is more advanced than Cassandra's when deciding how many data blocks to move to the new nodes, as it takes into account replica locations to minimize transferred data. Such a behavior leaves the cluster unbalanced in terms of disk usage, but it significantly reduces rebalancing time compared to Cassandra. In Cassandra's case a decentralized but more naive and less efficient in terms of data movement approach is used to balance the cluster, which divides the ID space of a pre-existing node and assignes one part to a newly introduced node.

[2]`http://bit.ly/iCNsbF`
[3]`http://bit.ly/kEZZFI`

As time costs for data rebalancing operations are large (93 and 660 minutes in HBase and Cassandra respectively), and since the benefits can be outweighed by adding more nodes without data migration (which incurrs a fixed time cost of 5 minutes for each DB as shown in our results), we conclude the costs outweigh the gains when aiming for the highest degree of elasticity. Therefore, subsequent tests were run without performing data rebalancing operations.

Resizing gains: Our evaluation continues by identifying the gains in performance when adding nodes under varying workloads without data rebalancing for HBase and Cassandra. We study the behaviour exhibited by an 8-node cluster under varying workloads after the addition of a variable number of nodes. We utilize a YCSB-generated load of $\lambda = 180$ Kreqs/sec, which is well over the maximum load that both Cassandra and HBase can efficiently handle deployed on an 8-node cluster. Two types of workloads are used, UNIFORM_READ and UNIFORM_UPDATE (referred to as READ and UPDATE henceforth). For each combination of workload and database, we perform an addition of 8 (i.e., double the size) and 16 (i.e., triple the size) nodes. The cluster resize occurs at about $t = 370$ sec (shown in Figures 2 and 3 as a vertical line). The client-side query latency and throughput μ are measured as well as the total aggregate cluster CPU usage reported by the Ganglia tool.

In Figure 2 we present our results for the HBase cluster. Legends refer to the workload type along with the resizing action (e.g., READ+8 represents a read workload with an 8-node resize). As read operations are faster than writes, we achieve comparable run times by adjusting the amount of objects requested in each workload, i.e., 10M in READ and 4M in UPDATE. The first graph presents the mean query latency. Adding nodes during READ loads (READ+8 and READ+16) has a transient negative effect on query latency, as HMaster reassigns regions to the new nodes when the addition is performed. Although data is not moved, the reassignment poses an extra burden to the already overloaded cluster. Clients cache region locations which however change during cluster resizing. Consequently latency increases due to client cache misses. This effect lasts for around 4-5 minutes when adding 8 nodes, and just a couple of minutes when adding 16 nodes (with query latency less affected during the latter transition). This is because more servers quickly join the cluster, taking a large portion of the applied load. The 24 nodes in total can now handle the load. Consequently, the transient period takes less time and clients are affected less. In the update workloads, we notice an oscillation in both cases: This happens because of the compaction and caching mechanisms of HBase. Incoming data is cached in memory (resulting in low update latencies) but when the memory is full and a I/O flush occurs with a compaction, the latency increases until the new blocks are written to the file system.

The final graph reports the aggregate cluster CPU usage as registered by the Ganglia tool. In the read workloads we notice that the initial load of around 55% is reduced to about 42% in both cases. Evidently, the newly arrived nodes immediately start handling incoming queries and alleviate the initially overloaded cluster. 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. Contrarily, a

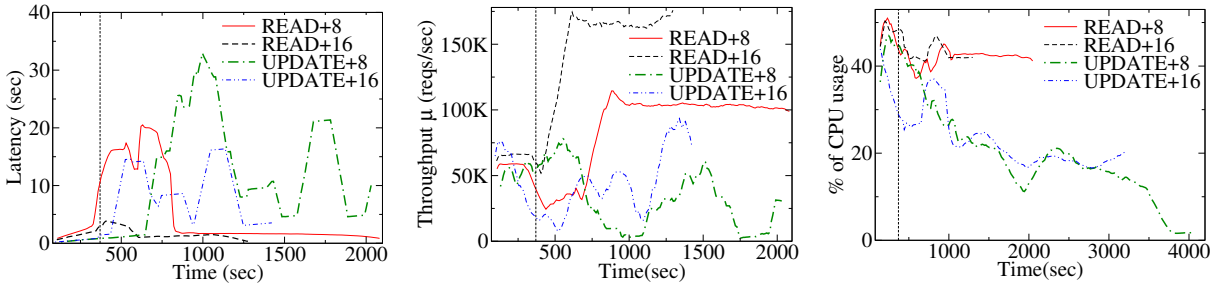

Figure 2: Query latency, query throughput and CPU usage per time for an HBase cluster of 8 nodes after adding 8 and 16 more nodes for the UNIFORM_READ and UNIFORM_UPDATE workload with a query rate of $\lambda = 180$ Kreqs/sec

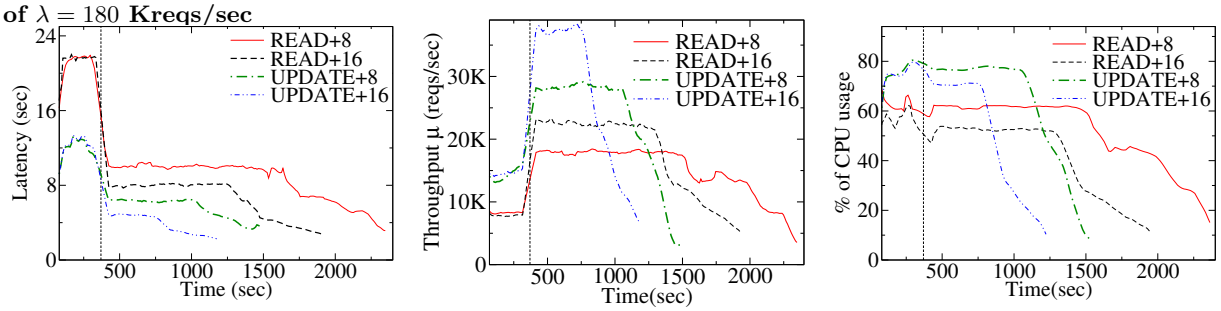

Figure 3: Query latency, query throughput and CPU usage per time for a Cassandra cluster of 8 nodes after adding 8 and 16 more nodes for the UNIFORM_READ and UNIFORM_UPDATE workload with a query rate of $\lambda = 180$ Kreqs/sec

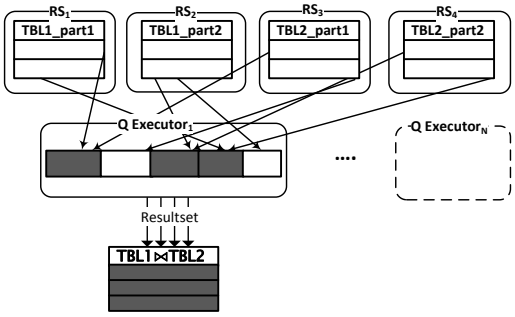

Figure 4: SPARQL querying architecture

drop in CPU usage is a good indication for adding servers against the maximum load. In the update workloads, we notice that in both experiments the initial CPU load continues to drop until run completion. This drop is due to the slow writes that occur during updates: The server freezes incoming requests until the updated regions are flushed to the file system.

In Figure 3 we present the respective results for the Cassandra cluster. In each test, clients request 2M each out of the total 20M objects stored in the database. During the experiment, clients are aware of the list of server addresses and each request is directed to a random one. As Cassandra does not have a mechanism to inform clients about new servers, a custom script was developed to propagate these changes back to the clients, whenever a cluster resize occurred. The first graph presents query latency: In both READ cases, we notice that the latency almost immediately drops from an initial value of around 22 secs to 10 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: Adding more nodes reduces the query latency from 11 sec to around 7 in the UPDATE+8 case and to around 5 in the UPDATE+16 case. Again, we notice here that writes are in general faster than reads, due to the weak consistency model followed by Cassandra, where writes do not have to be propagated to every replica holder for the operation to succeed.

Query throughput μ shows a similar trend. Both in read and update workloads we notice the almost linear effect of adding new nodes. Read throughput is increased from an initial value of 9 Kreqs/sec to 18 Kreqs/sec when 8 nodes are added and to 22 Kreqs/sec when 16 nodes are added. Update throughput is increased from around 15 Kreqs/sec to 29 Kreqs/sec in the +8 case, and to 35 Kreqs/sec in the +16 case. This behaviour is expected, since extra servers immediately join the p2p ring and take portions of the applied workload. Moreover, the asynchronous nature of eventual consistency enables Cassandra to maintain a stable throughput rate even in a write-heavy workload: Updates are successful upon transmitting the object to a single server, which replicates it later on in a lazy manner.

Finally, in the third graph of Figure 3 we present the variation of the total cluster's CPU usage during time. In the read case, we notice that adding 8 nodes slightly decreases the initial CPU usage to around 60%, whereas adding 16 extra servers decreases the average CPU load to around 50%. Same as HBase, the 50% load of the 24-node cluster shows that the applied load is big enough for every server to contribute, since new servers are not idle. The same hold for the update workloads: 8 more nodes bring the load to around 80% and the addition of 16 nodes drops the load to around 70%. Update workloads are more computationally heavy than simple reads, as in the update case there is an extra disk access cost that is avoidable by caching fetched results.

In this kind of setting, we note how both NoSQL systems take advantage of the addition of extra nodes. HBase per-

forms faster concurrent reads compared to Cassandra: In the READ+16 case it can handle 160 Kreqs/sec with a latency of about 2–3 secs and an aggregate CPU usage of 40%, whereas Cassandra's throughput is 40 Kreqs/sec, a latency of around 8 secs and a higher CPU usage of 50%. On the other hand, Cassandra is more efficient with object updates: It maintains stable throughput and latency curves avoiding oscillations that occur with HBase. Finally, we notice that Cassandra does not exhibit a negative transient effect when new nodes enter the ring. Its decentralized architecture allows for a transparent cluster resize, whereas in HBase the HMaster needs to coordinate the whole procedure.

3.2 Elasticity-provisioning prototype

In this section we present some initial results achieved using our framework to deploy a fully automatic cluster resize system. Our goal is to demonstrate the effectiveness and modularity of our design to allow for adaptive cluster resize without any human involvement. We argue that Tiramola's elastic HBase implementation can be used by higher level applications making them elastic and robust to load variances. To showcase Tiramola's elasticity provisioning we utilize it on a SPARQL querying application prototype [29] that depends on HBase to access data.

3.2.1 SPARQL querying architecture

In this section we focus on achieving scalable and efficient concurrent execution of selective SPARQL queries. Figure 4 depicts the functional overview of our RDF store prototype. A number of HBase servers (Region servers) maintains the triples indexed using 3 permutations (spo, osp, pos). Moreover, a number of query processing servers (Query executors) receive user requests, run the task of collecting data from the various Region servers, and locally perform the join task. Queries can be formed in the well known SPARQL standard. Different query executor instances can process queries independently and provide concurrent query throughput. To coordinate the several executor instances we utilize a zookeeper quorum. The zookeeper allows scalable and fault tolerant synchronization between large number of machines. To handle large query throughput we implement a distributed query queue using the zookeeper. Clients connect to the zookeeper and send their queries to the queue. On the other side, executors also connect to the zookeeper and constantly check the queue for input queries. The query execution servers reside in different nodes of the cluster and can be added or removed at will. Therefore, our architecture is elastic in both HBase and query execution layer, which means that we can independently change both the number of HBase and query execution servers.

There are two main factors that affect query throughput. First, SPARQL query throughput highly relies on HBase's read throughput. Using a large number of query execution servers can result in load that overcomes HBase's max read throughput. In that case executors will stall and HBase will become the throughput's bottleneck. Another factor that limits the number of servers is the number of CPUs available in each cluster's node. If a node executes more query server instances than its CPU capacity, server threads will be stalled and their execution will be serialized.

In order to achieve full elasticity we must ensure that adding more cluster nodes will result in increased query throughput. Each node has 2 virtual CPU cores. To avoid CPU over-utilization we launch 2 server instances on each node, one for every virtual CPU available on the node. From our HBase elasticity experiments we know that adding more HBase region servers results in increased read throughput, which will be able to withstand the increased amount of query execution servers. Thus, we choose to add one more HBase region server for every node in the cluster. Using two query execution servers and one HBase region server on each separate cluster's VM, provides a fully scalable and elastic SPARQL querying architecture.

3.2.2 Elastic SPARQL querying experiments

In order to provide a realistic, web-scale evaluation, we utilize the LUBM dataset generator [17]. LUBM creates RDF datasets with information taken from the academic domain, enabling a variable number of generated triples by controlling the number of *university* entities. A LUBM dataset of 5 thousand universities that contains approximately 7 million triples (125GB of raw data) is loaded using a MapReduce bulk import job that creates three distributed HBase indices [29]. We test our system's elasticity in concurrent query throughput with a load consisting of queries with the form:

```
SELECT  ?x
WHERE {?x rdf:type ub:GraduateStudent .
?x ub:takesCourse <...>}
```

This query retrieves all graduate students that take a certain course. It consists of 2 triple pattern queries that are joined by a single variable. To generate random load we use for each query a different randomly selected course and put it in the second triple query pattern.

In order to demonstrate the elasticity properties of our implementation, we utilize a long running sinusoidal load using a multi-threaded client, that queries our application using an average λ of 40 SPARQL queries per second, a peak load of 70 queries per second and a period of 1 hour. SPARQL queries are translated into multiple HBase index accesses, which means that a load of 70 SPARQL queries per second can result in a load of thousand HBase reads per second. This simulates a very fast variable load, that is common for most cloud applications.

We expect that our command-issuing module will autonomously decide to add nodes as the load increases beyond the server's capability and then return to the original cluster size once the extra load has faded. Using our knowledge of the database's operating limits, outlined in the previous experiments, we have configured the command-issuing module accordingly. The parameter to trigger node addition is increasing CPU usage of over 60%, measured as an average over all servers that compose the cluster. This measure takes into account the extra processing required for our application's joins, and is therefore set higher than our previous NoSQL examination would suggest is necessary. On the other hand, to remove a node, our threshold is average CPU usage of 50% or lower. To avoid oscillations as CPU usage could vary greatly during normal operation, the command-issuing module monitors a running average of CPU usage with a time window of 5 minutes. Large variations that might occur during this period are considered transient and are taken into account as an average, since the system would not have time to act in this small time period.

In Figure 5 we present our experimental results for a managed 8-node cluster. At the beginning of the experiment, the high load forces the CPU usage to high levels and, given that the load is extreme for the size of the cluster, the queue size increases rapidly. With the addition of one node, the queue size returns to normal, but given the minimal decrease in

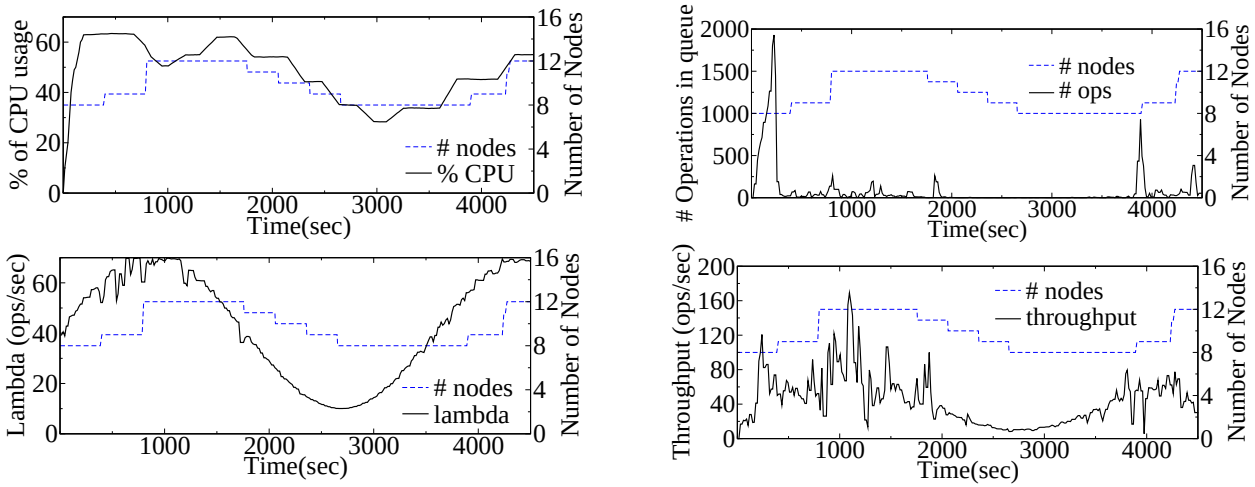

Figure 5: CPU, queue length, Input load (λ) and throughput (μ) for our elasticity prototype under sinusoidal load changes.

cpu usage over the previous time period, the system adds 3 more nodes to the cluster. The increased throughput is achieved and once the peak load has subsided, CPU usage drops and the queue length approaches zero, so our framework removes nodes from the cluster one by one, in order to retain the cluster data which are stored in HBase. The experiment then proceeds in a similar fashion, which validates the behavior of the system under similar conditions.

Although node addition and removal incurs a cost thus causing the cluster to perform erratically during the procedure, manifesting in the throughput and queue length graphs, the gains are evident. First, the cluster uses the minimum number of nodes required for good operation, in our example 8 nodes for most of the time, with a maximum of 12 needed for the brief high load periods, which translates to reduced operational costs. Second, our approach guarantees quality of service for users, even under heavy load, as requests are answered in a timely manner and the service cluster is always online.

4. RELATED WORK

A literature study about the challenges of scaling whole applications in cloud environments is presented in [34], along with an overview of state of the art efforts towards that goal. In [16], the authors propose a service specification language for cloud computing platforms that handles automatic deployment and flexible scaling of services. Similarly, the work in [27] designs a model implementing an elastic site resource manager that dynamically consolidates remote cloud resources on demand based on predefined policies and a prototype for extending Torque clusters. The works in [35, 31] solve the problem of optimizing VM resources (CPU, memory, etc) to achieve maximum performance using relational DBs. In comparison, we use the standard VM model that large cloud providers currently offer. As a general observation, these works do not address NoSQL systems and their performance under (dynamic) cluster resizes.

Herodotou et al [18] present the Elastizer, a system for automatic sizing of a MapReduce cluster according to workload profiling. When the user defines the job's profile through a set of parameters, Elastizer determines the amount of cluster resources by utilizing training sets of previous runs, in order to optimize the job's execution. Their approach can be applied only to batch-oriented data analytics tasks, whereas Tiramola can be applied to any application.

The work in [25] presents policies for elastically scaling the Hadoop-based storage-tier of multi-tier Web services based on automated control. Oppositely, we studied NoSQL systems that store structured data and are easier to manipulate in the application level. We converge in adopting the same metric, CPU Utilization, and evaluating rebalancing costs. The main difference is that this approach is HDFS-specific since monitoring was integrated in HDFS nodes, while data rebalance is mandatory for new nodes. However, our framework uses an external monitoring tool, Ganglia, that is generic for any NoSQL system and rebalancing need not be forced, as it relates to the NoSQL system used.

A thorough analysis of various proprietary NoSQL-like cloud services, namely Google's AppEngine, Microsoft's SQL Azure and Amazon's SimpleDB, is presented in [23]. The authors test system aspects such as scalability, cost and performance, using the TCP-W benchmark. All systems are treated as "black boxes", since no information about their design or implementation is assumed to be known. In contrast, our system is fully aware of the different engines' mechanisms. Moreover, we utilize our measurements to present a modular framework that can be used to realize automatic cluster-resizing.

Cloudy [22] is a cloud-enabled framework which supports auto-scaling features according to demand, providing simple key/value put/get primitives. Nevertheless, compared to our system, it is not designed to support numerous NoSQL databases. In NEFELI [33], cloud users inform the cloud management framework with hints about the application nature and the framework modifies its scheduling policies to improve application performance. The difference is the need for a middleware to be installed inside the cloud management layer, whereas our framework utilizes only the cloud client tools, being completely agnostic about the internals of the cloud management platform, enabling it to be installed over various platforms.

CloudWatch [1] is Amazon's commercial product for monitoring and managing cloud resources. CloudWatch offers a set of metrics for every VM instance and a policy framework to trigger balancing actions when some conditions are met. Its metric support is limited, offering only hypervisor-related information such as CPU usage and network traffic. Memory usage or application-specific metrics are not supported

out of the box, in contrast to Ganglia which, apart from its own rich set of probes, has inherent support from many applications, including some NoSQL systems. CloudWatch, as a general purpose tool, does not have inherent support for NoSQL, thus the extra coding to reconfigure the application cluster after a hardware resize is not avoided. Other frameworks for commercial cloud platforms like AzureWatch [3] feature similar characteristics, resulting in vendor lock-ins.

5. CONCLUSIONS

This work presents an extensive study that quantifies and analyzes the costs and gains of various NoSQL cluster resize operations, utilizing two popular NoSQL implementations. To perform our study, we designed and implemented a fully modular, cloud-enabled framework that allows efficient resource monitoring and direct interaction with the cloud vendor and the cluster manager. We have also evluated the performance gains that a distributed application, utilizing a NoSQL database backend, can achieve by exploiting the elasticity offered by modern cloud infrastructures.

The ease of setup and the performance under elastic operations weigh on choosing a particular NoSQL. Both engines we tested can be setup with relative ease. There are few configuration files that need to be injected during launch and most of the operational parameters can be adjusted by altering the appropriate settings. There is no need to provide new configuration files or commands during normal operation, especially during the rebalancing phases. Nevertheless, their quite distinctive behavior and performance under different scenarios make the decision quite application-specific: HBase is the fastest and scales with node additions (only for reads though), while Cassandra performs fast writes and scales also, without any transitional phase during node additions. Both achieve small gains from a data rebalance, provided they are under minimal load.

We believe that this work has demonstrated the feasibility of our ultimate goal. 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. We have also demonstrated the gains of a SPARQL query server application, using HBase as an RDF store, under variable query load and elastic operations. Our open-source implementation can provide a good basis on which numerous applications can test their adaptivity at very-high scale.

6. REFERENCES

[1] Amazon CloudWatch. http://aws.amazon.com/cloudwatch/.
[2] Amazon SimpleDB. http://aws.amazon.com/simpledb/.
[3] AzureWatch. http://www.paraleap.com/azurewatch.
[4] Google AppEngine. code.google.com/appengine.
[5] HBase Homepage. http://hbase.apache.org.
[6] OpenStack: The Open Source, Open Standards Cloud. http://openstack.org.
[7] Project Voldemort. http://project-voldemort.com.
[8] Resource Description Framework(RDF). http://www.w3.org/RDF/.
[9] Riak Homepage. https://wiki.basho.com/display/RIAK/Riak.
[10] Windows Azure Platform. http://www.microsoft.com/windowsazure/.
[11] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. SW-Store: a Vertically Partitioned DBMS for Semantic Web Data Management. *The VLDB Journal*, 18:385–406, April 2009.
[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.
[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM SOCC*, pages 143–154, 2010.
[14] O. Erling and I. Mikhailov. RDF Support in the Virtuoso DBMS. *Networked Knowledge-Networked Media*, pages 7–24, 2009.
[15] G. DeCandia et al. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, page 220, 2007.
[16] F. Galán, A. Sampaio, L. Rodero-Merino, I. Loy, V. Gil, and L. M. Vaquero. Service Specification in Cloud Environments Based on Extensions to Open Standards. In *COMSWARE*, 2009.
[17] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158 – 182, 2005.
[18] H. Herodotou, F. Dong, and S. Babu. No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-Intensive Analytics. In *ACM SOCC*, page 18. ACM, 2011.
[19] E. Horowitz. Foursquare Outage Post Mortem. http://bit.ly/gjA2IK.
[20] A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM–a Pragmatic Semantic Repository for OWL. In *WISE*, pages 182–192. Springer, 2005.
[21] I. Konstantinou, E. Angelou, C. Boumpouka, D. Tsoumakos, and N. Koziris. On the Elasticity of NoSQL Databases over Cloud Management Platforms. In *CIKM*, pages 2385–2388, 2011.
[22] D. Kossmann et al. Cloudy: a Modular Cloud Storage System. In *VLDB*, 2010.
[23] D. Kossmann, T. Kraska, and S. Loesing. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. In *SIGMOD*, pages 579–590, 2010.
[24] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.
[25] H. C. Lim, S. Babu, and J. S. Chase. Automated Control for Elastic Storage. In *ICAC*, 2010.
[26] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. Byers. Big data: The Next Frontier for Innovation, Competition and Productivity. *McKinsey Global Institute, May*, (June), 2011.
[27] P. Marshall, K. Keahey, and T. Freeman. Elastic Site: Using Clouds to Elastically Extend Site Resources. In *CCGRID*, 2010.
[28] 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.
[29] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H2RDF: Adaptive Query Processing on RDF Data in the Cloud. In *WWW*, 2012.
[30] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD*, pages 165–178, 2009.
[31] A. Soror, U. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic Virtual Machine Configuration for Database Workloads. In *SIGMOD*, 2008.
[32] SPARQL Query Language for RDF. http://www.w3.org/tr/rdf-sparql-query/.
[33] K. Tsakalozos, M. Roussopoulos, V. Floros, and A. Delis. Nefeli: Hint-Based Execution of Workloads in Clouds. In *ICDCS*, pages 74–85, 2010.
[34] L. M. Vaquero, L. Rodero-Merino, and R. Buyya. Dynamically Scaling Applications in the Cloud. *SIGCOMM Comput. Commun. Rev.*, 41:45–52, 2011.
[35] P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. Hacigümüs. Intelligent Management of Virtualized Resources for Database Systems in Cloud Environment. In *ICDE*, 2011.