Automated Workload-aware Elasticity of NoSQL Clusters in the Cloud

Evie Kassela, Christina Boumpouka, Ioannis Konstantinou and Nectarios Koziris
CSLAB, National Technical University of Athens
{*evie,christina,ikons,nkoziris*}*@cslab.ece.ntua.gr*

***Abstract*—The use of cloud computing has gained extreme popularity. Through cloud platforms that provide infrastructure as a service (IaaS), users can elastically provision resources enabling automated application throttling. Usually, scaling is either manually performed or through a service that dynamically consolidates cloud resources based on a predefined policy. However, these policies are simplistic, threshold based and may not be able to capture specific application behaviors according to configuration parameters and applied workload type. In this work, we extend TIRAMOLA, a cloud-enabled framework that allows automated resizing of NoSQL clusters, in order to identify different workload types and apply the most beneficial scaling action according to user defined policies. We perform a thorough analysis of how different query types are handled by modern NoSQL systems and evaluate the performance of a NoSQL cluster of varying size, over mixed workload types and magnitudes. We utilize this knowledge to fine tune the extended TIRAMOLA's policies in order to take accurate scaling decisions. We perform an extensive experimental evaluation of workload aware and unaware versions on an HBase cluster and our analysis confirms that the former can operate successfully in any environment, behaving accordingly to any input load.**

***Keywords*-NoSQL; elasticity; benchmarking; policy tuning;**

I. Introduction

The need for efficient storage, processing, and serving TBs or even PBs of data is now crucial not only for traditional data-oriented companies like Google, Facebook, etc but also for smaller ones [3]. SMEs or start-ups that come across Big data needs is a common phenomenon. Moreover, companies that offer Internet services to end-users (e.g., social networking or gaming sites) typically experience varying and unpredictable resource demands according to user traffic [16]. Whether this demand variation is attributed to scheduled batch processing, anticipated seasonal patterns or unanticipated load spikes due to trending services, companies require both scalable and flexible infrastructures that can easily cope with these irregular load patterns. Therefore, legacy approaches that consist of static application deployments over private data-centers are not an option anymore. Cloud computing can tackle large up-front infrastructure costs, varying resource needs and static resource allocation problems. Whether inside a private data-center, or entirely in the cloud, elastic platforms offer the dual advantage of better resource utilization and cost minimization, a win-win situation for both SMEs and larger companies.

Application elasticity, i.e., the ability to take advantage of extra added resources or to continue to function when idle resources are removed in a seamless manner so that exact demand or respective SLAs are met [8] is a required characteristic in order to setup an end-to-end elastic offering. Nevertheless, cloud technology is not the silver bullet that "automagically" provides these characteristics out of the box for any possible Big data application. Elaborate mechanisms for the accurate and timely detection of both the violation of user defined policies during the system's operation and the appropriate healing action (i.e., scaling up or down resources) are needed. Numerous research approaches have been proposed. MET [13] and the work in [10] resize a cluster using performance metrics taking into account the data locality. Starfish [15] and Autoscale [14] use predictors and take decisions to achieve the minimal costs. ShuttleDB [9] uses a two-level elasticity mechanism with a specific scaling policy and TIRAMOLA [18] offers elasticity based on any user defined policy. Finally, elastic frameworks by companies like [1], [5] have been implemented.

In this work, we build upon TIRAMOLA [18], an existing framework that offers automated elasticity using a machine learning approach that enables the system to adapt to workload variations. Although TIRAMOLA monitors applied workload and performs the appropriate elasticity actions, it is not configured neither to understand the applied workload type nor what is the best action according to any encountered mix of queries. We make the following contributions:

- We extend TIRAMOLA to take into account different workload types such as read and write queries and fine tune TIRAMOLA's decision making mechanism to adjust its behavior according to the type of the applied workload.
- We study how and why these workloads interfere considering the internal mechanisms used by any NoSQL cluster to handle read and write queries, such as caching, etc.
- We perform a thorough evaluation of the behavior of different cluster sizes of HBase, a commonly used NoSQL database, over mixed workloads that vary both their intensity and their type over time.
- Finally, we compare our findings with the original, workload un-aware TIRAMOLA, and present its improved behavior as it can better keep up with workload changes.

II. Tiramola Overview

TIRAMOLA is an elasticity framework for NoSQL systems featuring a modular architecture in agreement with most frameworks in the category [10], [13]. TIRAMOLA

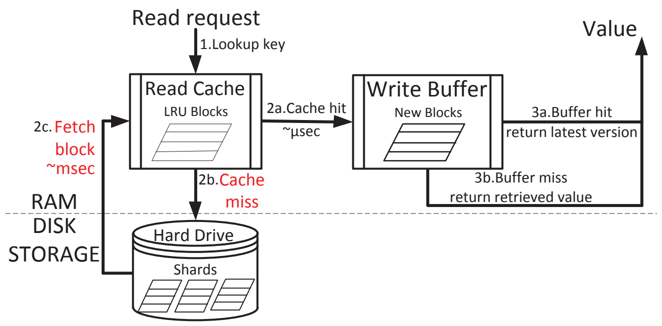

Figure 1: Read operation data flow, costly operations shown in red.

consists of different components for monitoring, making decisions about cluster resizes and interfacing with the cloud. As described in [18], decision making is modeled as a *Markov Decision Process*. The states $S = \{S_1, ..., S_M\}$, represent a cluster of $i \in [1, M]$ NoSQL nodes and the available actions in each state are to add/remove node, or do nothing. The reward function $r(s)$ serves as an indicator of the profit the system would currently have, had it been in state s. Since the user is able to define the reward function at will, the system can implement different policies depending on the metrics included in the reward function.In our experiments, we select a reward function of the form:

$$r(s) = f(throughput, latency, VMs).$$

Therefore, TIRAMOLA takes into account the performance of the cluster in terms of total throughput and total latency to decide the most beneficial state, but has no information about the type of queries executed in the cluster. The metrics that participate in the reward function form a d-dimensional dataset. For each different cluster size a 2-dimensional dataset is formed with throughput and latency measurements. TIRAMOLA assumes that the system should behave similarly in similar conditions and therefore includes only measurements relative to the current cluster load when calculating rewards. A k-means clustering of the points within a certain "slice" of throughput values around the current measurement is done and the centroid's coordinates are used as throughput and latency to compute the reward.

A. Workload Awareness

Several steps were taken to make TIRAMOLA aware of the type of load applied to the NoSQL cluster and integrate this piece of information into the decision making process.

The YCSB clients [12] producing the workload were configured to report separate metrics, i.e. throughput and latency, for each type of query. In correspondence, TIRAMOLA's Monitoring module was adjusted to collect and report the new metrics. We incorporate the separate metrics in the reward function and propose using:

$$r(s) = f(read\ thr, write\ thr, read\ lat, write\ lat, VMs).$$

In addition, TIRAMOLA's Decision Making module was modified to perform 4-dimensional clustering, when the dataset includes read/write throughput and read/write latency for each state and the centroid's coordinates are used to compute the reward. The data points fed to the clustering engine include measurements for which both read and write throughput are within the respective throughput "slices". This way the system is able to notice the different types of load and their corresponding metrics and perform actions that address the specific workload mix.

III. Basic NoSQL operations

In this section, we outline important NoSQL mechanisms that allow explaining the system's behavior in sec. IV.

A. NoSQL data storage and caching

Commonly, data is partitioned into shards, assigned to different nodes, that may be replicated to ensure availability and safety in case of failure. Shards are further divided into data blocks and data is indexed by row/column so that each block contains a number of rows. In every transaction a whole block is transferred to/from the disk. Most systems support auto-sharding meaning that the shard partitioning and reassignment during NoSQL cluster resizing are seamlessly performed. Splitting data into shards allows to equally distribute load among nodes automatically. Moreover, in-memory caching is used to improve the overall system performance. Caching blocks that were recently read may speed up read operations, whereas using an in-memory buffer for writes saves time since it allows bypassing the disk. The memory used for caching reads follows an eviction policy when it is full (e.g., LRU), and cache misses cause a latency penalty as data needs to be fetched from the disk. The memory used for buffering writes gets periodically flushed to the disk when it reaches a user imposed limit. Atomic operations are guaranteed since a row lock must be acquired before each write transaction. After a certain number of buffer flushes, stored data is reorganized and sorted automatically by a maintenance mechanism that runs periodically in the background and consolidates small files into larger ones to avoid fragmentation. During this, serving performance may be affected due to heavy I/O ops.

B. Handling Reads & Writes

Based on the modules described above the workflow of a read and a write query is depicted in Figures 1 and 2 and can be outlined as follows:

- Upon a write request, a row lock is acquired and the new data is first written to the WAL and then to the in-memory buffer, while the contents of the disk and the cache remain unchanged and valid. Then the row lock is released. At a buffer flush, its contents are written to disk blocks inserting or updating block data. If any of these blocks are loaded in the cache they now become outdated and invalid for use.
- Upon a read request, a search on the cache is performed and if no valid data is found, the requested data is fetched from the disk. In addition, the buffer is always checked for any new versions of the data not yet flushed to the

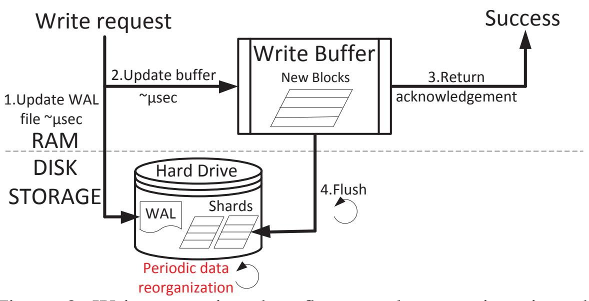

Figure 2: Write operation data flow, costly operations in red.

disk. We note that the system first searches the memory (read cache and write buffer) before it resorts to expensive disk seeks.

In other words, a write transaction doesn't include memory searching or writing to the disk and it is completed once the data is written in the WAL and in the buffer. WAL writing is usually as fast as in-memory writing as the file is kept open for sequentially appending new data avoiding expensive random disk seeks and it's replication is asynchronous. Yet, a row lock must be acquired first, so writing latency mostly increases if there is waiting time to acquire the lock. On the other hand, a read transaction requires in the best case a cache search and in the worst case fetching data from the disk and searching the buffer for updates, but no locks are required. This means that read latency is higher than write latency since searching is an expensive operation.

Read and write performance can therefore be affected under certain circumstances. Read latency can be extremely increased in heavy-write environments or when the cache size cannot fit all the concurrently read blocks. This is because in these cases cache contents become more often invalid or evicted and data must be re-fetched from the disk. Regarding write performance, the periodic flush of the buffer is not a blocking operation as a new empty buffer is instantly created, it is however resource consuming. Also, data maintenance is performed periodically after multiple buffer flushes which blocks writes on the processed data until it is completed. This negatively affects write performance, not in terms of increased latency but in terms of requests being totally blocked for a considerable amount of time. Data maintenance is performed more frequently in heavy-write environments or if the buffer size is too small for the applied write load and causes recurring disk flushes. In this case, write requests are blocked more often and write throughput drops. If multiple write requests address the same record or block, they are queued, trying to acquire the same row lock leading to an increased latency. All these factors are considered in the configuration of our system and the interpretation of its performance.

C. *Comparing NoSQL implementations*

NoSQL stores adopt similar methods for solving consistency and scalability issues. In HBase [4], data shards are called HRegions and their rows contain a number of column families. Each column family has a container (Store), and each Store has a write buffer (MemStore). Every HBase node keeps a WAL (HLog) and uses a cache (BlockCache) for the Regions it is responsible for. Likewise, Cassandra [2] shards are called partitions and SSTables are the column containers. A "row cache" is stored in memory while writes are buffered in memtable and accumulated sequentially in the CommitLog per node. MongoDB [6], stores document collections, split into shards and distributed to the nodes. In an attempt to make good use of the OS cache, Extends, the data files stored in disk, are mapped to a structure in memory named "shared view". WALs are named journals and the memory buffer is the "private view". In essence, the shared and private views play the role of the read cache and write buffer respectively.

Since most systems have similar data flows for reads and writes, we expect them to experience the same implications when coping with different load types. Relying on TIRAMOLA's ability to supervise any NoSQL system, our extension is also independent of the specific NoSQL store used; it can manage any system given the appropriate reward function.

IV. Experiments

The experimental section intends to demonstrate the automatic management of VM resources by TIRAMOLA under varying incoming loads. Applied workloads follow a typical seasonal pattern found in web serving applications (Fig. 14 in [16]) and Microsoft Messengers weekly load (Fig. 5 in [11]). In particular, we evaluate loads with different percentages of read and write queries, from read-only workloads (100% read queries) to heavy-write workloads (33% write queries). In the 33% write queries case, one third of the applied workload is constantly updating the entire dataset, while the other two thirds continue to perform read queries.

In section IV-A we refer briefly to the selected configuration for the NoSQL cluster and the clients. In section IV-B we describe in detail the training process of TIRAMOLA, for which we use loads with different proportions of read and write queries and different amplitude. We analyze the cluster's performance with each load. Finally, in section IV-C based on our knowledge of the cluster's operation we select two different reward functions for TIRAMOLA, one workload-aware and one workload-unaware, and evaluate its efficiency with loads of varying amplitude and type.

A. *Experimental setup*

Our experimental setup consists of an OpenStack [7] cactus private cluster of 10 client VMs (load generators) and 10 server VMs. Each server/client VM has 2 vcores with 4GB of RAM and 200GB of storage space. We utilize an HBase [4] (v.0.92.0) NoSQL cluster with initial size of 4 VMs that can be increased up to 10 VMs. HBase is configured with a replication factor of 3 and is optimized for heavy-write

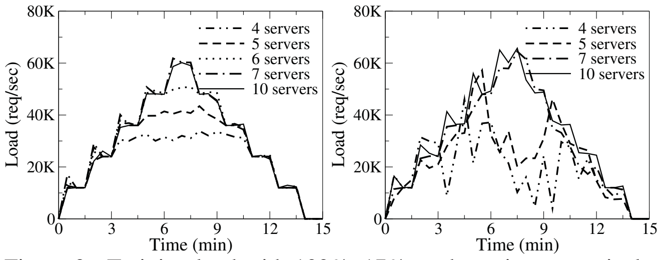

Figure 3: Training load with 100%, 17% read queries respectively

loads using the following settings: blockingStoreFiles=100 and blockingWaitTime=0. This allows more buffer flushes before the data maintenance mechanism is triggered and write requests get blocked (last paragraph of sec. III-B), and is also an attempt to minimize requests' blocking time. In this way a rare but time-consuming maintenance will be performed. Finally, we use Hadoop 1.0.1 and Ganglia 3.1.2, both in their default configuration.

The cluster is loaded with 20M records using the YCSB benchmark [12] (v.0.1.4) resulting in 90GB of data (with replication). Our workload is also generated with YCSB and comprises of simple read and write queries (UNIFORM READ and UPDATE query types). In order to maximize independently read and write performance, read queries were applied to 20K records that are loaded in the nodes' cache initially while write queries were applied to the whole dataset to avoid locks' impact. Using a bigger range of records for read queries that doesn't fit in memory, would lead to reduced performance due to constant evictions from the cache. A smaller range of write queries would also cause reduced performance in the case of an increased load of write requests, as multiple requests would queue on the same locks and very high latency would emerge. This particular client-side configuration maximizes HBase performance in both heavy-reads and heavy-writes.

B. Training process

To assist TIRAMOLA in estimating each state's reward during the initial phase of the system's operation, we apply a training phase. Our training set is a simple sinusoid-like load with an average value of 30K req/sec, a peak of over 60K req/sec and a period of 15 minutes. We train the system in all sizes (4-10 servers) with 7 different load types starting from a load of 100% read queries and gradually increasing write queries' percentage to 100% write queries. A period of the training load for several cluster sizes is recorded for 2 different load types in Fig. 3, where a clear relation between the cluster's size and the achieved throughput is shown. Each worker can serve about 10K req/sec, either read or write requests, so the cluster is able to fully serve our peak load when it has at least 7 servers (6 workers, 1 master). We notice an important difference in the line shapes between graphs that correspond to different load types. Throughput develops an increasing fluctuation as writes increase especially in cluster sizes below 7 (Fig. 3), because memstores fill and flush more often and instant flushes affect negatively the already overloaded system, causing this seemingly unstable performance.

A better understanding of the system's operation and performance comes from Figures 4 and 5. In Fig. 4 we register throughput vs load type with a stable input load of 60K req/sec in all cluster sizes for read and write operations respectively. First of all, a clear difference in the lines' shape is observed in the performance of small cluster sizes, although we would expect for them to be similar (symmetrical in relation to y-axis). Noticing in Fig. 4 the left side of the left graph (heavy-read) and the right side of the right graph (heavy-write), we see that read throughput is better than write throughput. This is expected since the system is more loaded in heavy-write operation by the frequent buffer flushes, while in heavy-read operation mainly the cache is accessed. Secondly, we observe how read and write operations affect one another. In Fig. 4 the right side of the left graph (heavy-write) and the left side of the right graph (heavy-read), show that read throughput is much worse than write throughput: In a heavy-write operation, except that the system is more loaded, there is a greater chance that cache contents are invalidated. Since disk data changes, blocks have to be reloaded from the disk (section III-B), causing read performance to decrease further.

Similar conclusions emerge from Fig. 5 where we register queries' latency vs load type (read latency is in logarithmic scale). At first, it is important to notice the difference in the size of read and write latencies. Read latency is at least 10 times higher than write latency, confirming our analysis in the last two paragraphs of section III-B. Thus, total measured latency is mainly formed by read latency. As expected, latency decreases as the cluster size increases in both cases. Also, as write queries increase we can see that write latency increases only in small cluster sizes (dotted lines in right graph of Fig. 5) where the system can't handle the increasingly heavy load. On the other hand, read latency increases significantly for every cluster size, as write queries increase. This indicates that write operations not only stress the system and can indirectly affect total performance but they also directly affect read performance as data must be re-fetched from the disk. The fact that HBase exhibits a low write latency at the cost of a high read latency is also confirmed in [17] where the results match ours.

Finally, an interesting comparison is presented in Fig. 6 where we register in x axis the percentage of read queries in the incoming load and in y axis the percentage of read queries in the actually served load. We would expect that in every cluster size the percentage of served read queries would be the same as the incoming load's percentage, therefore all lines should be straight lines on the diagonal. Instead, in small cluster sizes there is a divergence from the actual expected percentage (on the diagonal). In particular, when we have 30-99% read queries in the

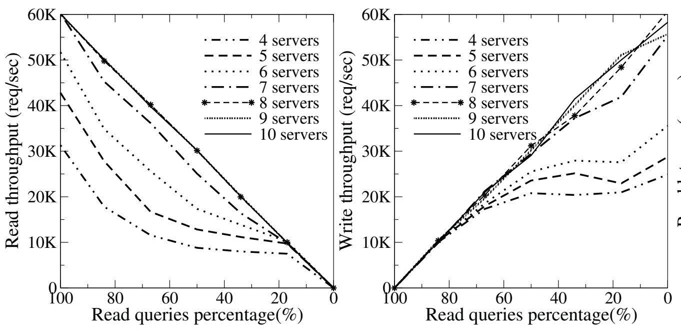

Figure 4: Read/Write Throughput vs load type for every cluster size

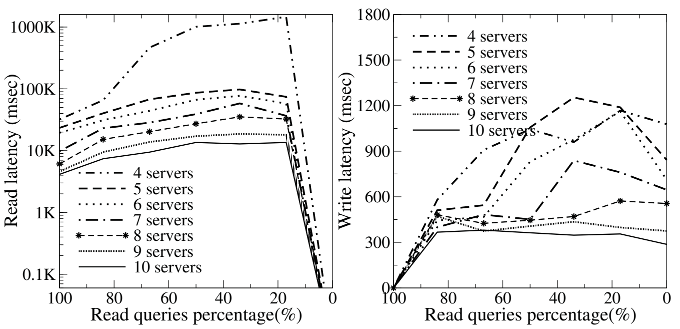

Figure 5: Read/Write Latency vs load type for every cluster size

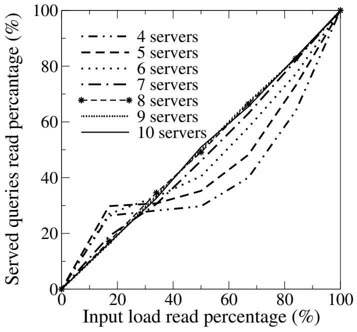

Figure 6: Output vs input load type

incoming load the system serves less read than write requests while in 0-30% read percentages the system serves more read than write requests. These two opposite behaviors confirm that read requests are negatively affected by write ones except in heavy-write operation where the system struggles with writes but serves a few reads.

C. Load-unaware vs load-aware

To evaluate the load-unaware and load-aware operation of TIRAMOLA we devise two different reward functions:

$r_1(s) = A \cdot thr - B \cdot lat - C \cdot |VMs|$

$r_2(s) = A_r \cdot thr_r + A_w \cdot thr_w - B_r \cdot lat_r - B_w \cdot lat_w - C_t \cdot |VMs|$

Function $r_1(s)$ considers total throughput thr, total latency lat and the number of VMs, hence it is load unaware. Function $r_2(s)$ uses the same metrics divided by query type: read throughput thr_r, read latency lat_r, write throughput thr_w, write latency lat_w and VMs and is therefore load aware. We use a plus sign for the metrics considered profitable and a minus sign for the costly ones.

In order to calculate parameters $A, B, C, A_r, A_w, B_r, B_w, C_t$, a good understanding of the system's operation is required, which can be obtained by the training phase. As a result, the analysis of the training phase is very important in grasping this knowledge and applying it as we do in the following procedure. Our chosen values for function r_1 are $A = .001$, $B = .002$, $C = 1.4$. We apply double weight in latency in relation to throughput, based on the throughput and latency values that appear in Figures 4, 5 for 100% read load (30-60K throughput, 4-40K latency). For the number of VMs (values 4-10) a greater parameter was required so that TIRAMOLA doesn't add/remove nodes too greedily or too modestly. With function r_1 load unaware TIRAMOLA is expected to work flawlessly with a read-only load and its efficiency with other types of loads remains to be seen. For function r_2 we chose $A_r = .0005$, $A_w = .0005$, $B_r = .0003$, $B_w = .0007$, $C_t = 1.2$. We split the value of A into A_r and A_w, as read and write throughput are of the same order of magnitude. A different method was used for latency parameters since read and write latencies have a proportion of at least 10 units (sec. IV-B). Based on this fact, we would have chosen a write latency parameter 10 times higher than the read latency one. Instead, we chose a 2 times greater parameter, thus relying more on read and less on write latency, as read latency shows steeper changes when the type of load changes. It is therefore considered a more representative metric for identifying the type of load (Fig. 5, sec. IV-B). To select B_r, B_w and C_t a more extensive experimentation was required. We note here that our intent is not to find perfect parameter values but to demonstrate the procedure that is followed in setting up a new reward function and the system's efficiency with different reward functions when the type and size of the input load change.

In our first experiment we apply three different types of loads with 100%, 84% and 67% read requests, all with the same sinusoid-like shape with a period of 1.5 hour (10x slower than training load) and an average of 30K req/sec. We didn't increase write percentage any further because 67% read and 33% write requests is already a heavy-write workload and the cluster's performance drops drastically after one hour due to intensive flushing and data maintenance (sec. III). We register the observed system's output as well as the decisions taken using $r_1(s)$ over time in Fig. 7. The area that is filled with the lightest color depicts the served read workload in req/sec, whereas the darkest area depicts the write workload. The dotted lines represent the cluster size in VMs and their steps depict TIRAMOLA's decisions.

TIRAMOLA manages to behave according to the system's load for the three load types, as shown in Fig. 7. With 100% read requests, increase and decrease size actions follow the load trend. However, with 84% and 67% read requests TIRAMOLA doesn't behave similarly as the cluster's size increases to 7 nodes too soon and then exhibits frequent small fluctuations. This indicates that when TIRAMOLA uses reward function r_1, it can't operate with any type of load as efficiently as it does with 100% read load. Reward function r_1 doesn't take into account the type of load, it only considers total throughput and total latency. Thus, the data points included in the throughput slice and used by the clustering engine come from every type of load that displays

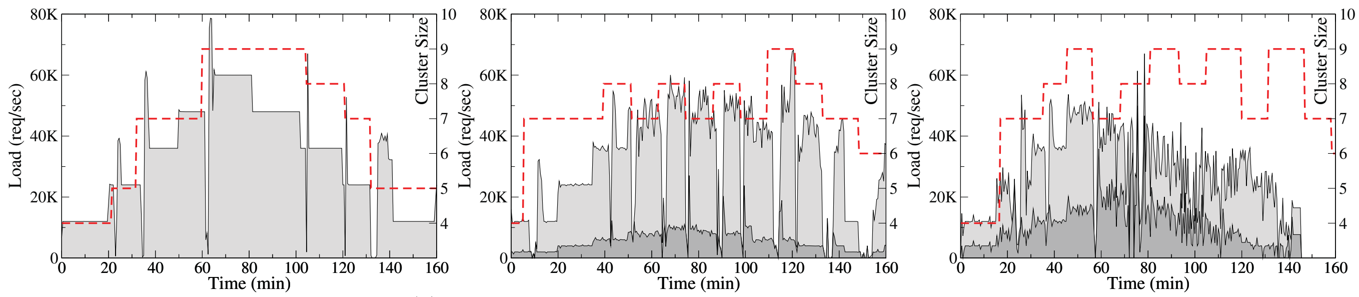

Figure 7: System behavior with $r_1(s)$ when applying three similar loads with 100%, 84% and 67% read queries respectively

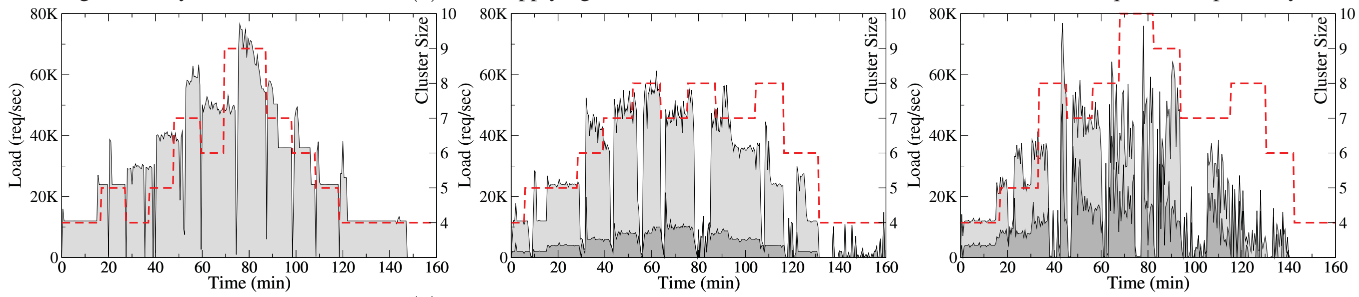

Figure 8: System behavior with $r_2(s)$ when applying three similar loads with 100%, 84% and 67% read queries respectively

similar total throughput. This way, the estimated gains are totally different from the current gain as they are based on irrelevant data from different load types and TIRAMOLA is driven to a wrong state. Latency plays an important role, since it changes significantly when the load type changes (sec. IV-B) turning the centroids' y-coordinates irrelevant.

Fig. 8 displays TIRAMOLA's behavior when we use reward function r_2 with the same input load. With r_2 TIRAMOLA is taking better decisions for all 3 load types, adding and removing VMs as load increases and decreases, yet its behavior is not perfect. Small fluctuations appear which can be handled by fine tuning the parameters of r_2, or by configuring TIRAMOLA to make decisions based on the average of the three latest (instead of one) measurements to avoid unstable throughput measurements that appear when write requests increase, as observed in Fig. 3. Reward function r_2 allows TIRAMOLA to handle more types of loads quite efficiently now that the type of load is considered and the decision is taken performing a clustering of the data corresponding to a certain type of load. This was anticipated as a system's performance may alter significantly when the load type changes and TIRAMOLA needs to be aware of that fact. Finally, we note that the different frequency in the training load and the applied load does not affect TIRAMOLA, showing its ability to adapt to any applied load and operate normally if correctly configured.

In conclusion, our analysis proves that the load-aware version of TIRAMOLA can perform more informed scaling decisions compared to the unaware one, enabling it to adapt to workloads of different types, magnitudes and frequencies.

Acknowledgments

This work has been funded by EU and GR Resources under the Hellenic (GSRT) "COOPERATION 2009" National Action "09SYN-72-881" MoDisSense Project.

References

[1] AWS Elastic Beanstalk. http://aws.amazon.com/elasticbeanstalk/.
[2] Cassandra. http://cassandra.apache.org/.
[3] EMCVoice. http://onforb.es/1rnYzdp.
[4] HBase. http://hbase.apache.org/.
[5] Jelastic. http://jelastic.com/.
[6] MongoDB. http://www.mongodb.org/.
[7] OpenStack. https://www.openstack.org/.
[8] M. Armbrust et al. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.
[9] S. Barker et al. ShuttleDB: Database-Aware Elasticity in the Cloud. In *ICAC*, 2014.
[10] M. Chalkiadaki and K. Magoutis. Managing service performance in the cassandra distributed storage system. In *CloudCom*. IEEE, 2013.
[11] G. Chen et al. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *NSDI*, volume 8, pages 337–350, 2008.
[12] B. Cooper et al. Benchmarking cloud serving systems with YCSB. In *SOCC*, pages 143–154, 2010.
[13] F. Cruz, , F. Maia, M. Matos, R. Oliveira, J. Paulo, J. Pereira, and R. Vilaa. MET: Workload aware elasticity for NoSQL. In *ACM ECCS*, 2013.
[14] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. Autoscale: Dynamic, Robust Capacity Management for Multi-tier Data Centers. *ACM (TOCS)*, 30(4):14, 2012.
[15] H. Herodotou et al. Starfish: A Self-tuning System for Big Data Analytics. In *CIDR*, 2011.
[16] A. Qureshi, R. Weber, H. Balakrishnan, J. Guttag, and B. Maggs. Cutting the electric bill for internet-scale systems. *SIGCOMM*, 39(4):123–134, 2009.
[17] T. Rabl, S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, and S. Mankovskii. Solving Big Data Challenges for Enterprise Application Performance Management. In *VLDB*, 2012.
[18] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris. Automated, Elastic Resource Provisioning for NoSQL Clusters Using TIRAMOLA. In *CCGrid*, 2013.