PAX: Partition-Aware Autoscaling for the Cassandra NoSQL Database

Salvatore Dipietro
Department of Computing
Imperial College London, UK
s.dipietro@imperial.ac.uk

Rajkumar Buyya
Cloud Computing and Distributed System Lab
School of Computing and Information Systems
University of Melbourne, Australia
rbuyya@unimelb.edu.au

Giuliano Casale
Department of Computing
Imperial College London, UK
g.casale@imperial.ac.uk

***Abstract*—Apache Cassandra has emerged as one of the most widely adopted NoSQL databases. However, there is still a limited understanding on how to optimally operate Cassandra in the cloud using autoscaling methods, by which resources can be scaled up or down to reduce operational costs and meet service-level objectives (SLOs).**

To address this limitation, we present PAX, a partition-aware elastic resource management system for Apache Cassandra. PAX uses low-overhead query sampling and knowledge of the data-partitioning across the nodes to automatically adapt capacity in Cassandra clusters. Differently from existing autoscaling methods for Cassandra, which incur large acquisition times for new nodes, PAX exploits Cassandra's hinted handoff mechanism and a shared hints storage to minimize the time needed to acquire a node into the cluster.

We propose a reactive and a proactive implementation of PAX and compare their performance against different workloads with varying intensities and item popularity distributions, finding that the proactive version significantly reduces SLO violations.

I. Introduction

NoSQL databases offer the ability to store large quantities of information and retrieve them with lower latency than in traditional databases [1]. Apache Cassandra [2] is a widely adopted NoSQL database that features a decentralized architecture providing fault-tolerance, tunable consistency, selectable replication factors, and throughput scalability. In particular, the architecture of Cassandra lacks a single point of failure, making it well-suited for operation in unreliable environments such as the cloud. In spite of this, only recently researchers have started to systematically investigate autoscaling methods for Cassandra [3], [4], [5], [6].

Autoscaling methods have been investigated for several years in cloud-native web applications [7], [8]. Such methods help both cloud service providers and users to reduce operational costs and cope with workload variations [9]. Databases are also increasingly adapted to support autoscaling [10], [11], but they can face long acquisition times for the new nodes due to the wait time of data synchronization. Indeed, the time needed to add an entirely new node can take days if the dataset is very large, while at the same time affecting the performance of the existing nodes.

To illustrate the problem, we show in Figure 1 a measurement of the time required for Cassandra to add a new node to a cluster. In this example, the cluster runs on the Microsoft Azure cloud and consists of four virtual machines (VMs). Throughout the paper, all experiments are run using the YCSB benchmark [12]. Even though the cluster is small and the data stored in the nodes is not too large, around 15GB per VM, the system spends almost 30 minutes to create and transfer the data to the newly instantiated VM. Only at the conclusion of this process the new node is finally acquired into the Cassandra cluster and becomes operational. On top of this, the system requires additional time to stabilize its performance with the new node. If multiple nodes are simultaneously added to the system, the situation further degrades, with the transfer time period becoming even longer. Clearly, this problem hinders the ability for an autoscaling system to quickly adapt Cassandra to the incoming workload.

In this paper, we define a novel autoscaling method, called PAX, which relies on Cassandra's *hinted handoff* mechanism to efficiently add nodes to the cluster [13] and we introduce proactive and reactive policies that control Cassandra using workload and data partitioning information. The hinted handoff mechanism exploited for autoscaling is the mechanism that Cassandra uses for synchronizing pending writes to nodes that return online after some downtime. We argue that this mechanism can also be effectively used also to enable autoscaling for Cassandra, provided that instead of creating and booting up new VMs one keeps idle a large enough set of dormant (i.e., powered-down) VMs. Such VMs can be quickly synchronized to the cluster using the hinted handoff mechanism instead deploy an entirely new VMs.

After implementing autoscaling based on this mechanism, we show that PAX can effectively autoscale Cassandra as the rate of incoming queries varies over time. We introduce in particular a reactive and a proactive implementation of PAX, which scale resources based on CPU utilization, workload demands and arrival rate forecasting. To further optimize autoscaling, we show that PAX can leverage query sampling to decide the best dormant VMs to activate, based on the data partitions they contain. Our experiments indicate that partition-aware node acquisition can provide substantial improvements in throughput up to 69%.

We validate our approach through an extensive experimentation using time-varying arrival rates and different item

978-1-5090-0223-8/16/$31.00 © 2016 IEEE

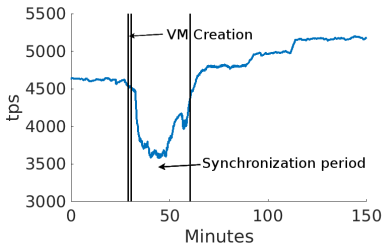

Fig. 1: Data synchronization period when a new node joins the cluster of 4 nodes with 15GB of data each with RF of 2.

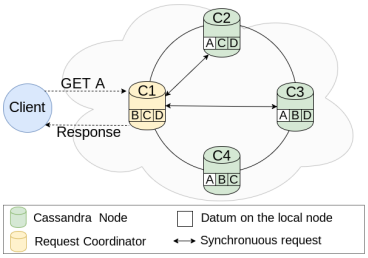

Fig. 2: Cassandra read request representation. The system is set with a RF of 3 and the query uses a CL of TWO.

popularity distributions. Among the main findings, we show that, compared to a system without autoscaling, PAX can delivery large cost savings without compromising the user QoS. In addition, thanks to the ARIMA predictive algorithm, the system seldom violates SLOs.

The rest of the paper is organized as follows. Section II reviews related work on elastic resource management for Cassandra. Section III introduces the architecture of Cassandra, including data partitioning and the hinted handoff mechanism. Section IV presents our elastic architecture and the PAX controller. The experimental validation is given in Section VI. Section VII concludes the paper and propose future work for dealing with burstiness.

II. Related work

Over the last few years, several NoSQL databases have been adapted to support elasticity resource management [11], [14], [15], [16], [17], [18], [19]. For instance, [17] presents a controller for elastic resource provisioning of HBase clusters using Markov Decision Processes (MDPs). Similarly, [18] defines a framework to automatically reconfigure HBase nodes based on their access pattern. The work in [19] shows instead a controller based on feed-forward and feedback signals for the Voldemort database.

Prior work on Cassandra autoscaling includes in particular [3], [4] and [5]. Both [3] and [4] present a reactive autoscaling mechanism. [3] develops a Cassandra controller that gathers node and workload performance data to calculate an exponentially-weighted moving average (EWMA) of the response time currently experienced by the users. When this moving average exceeds a pre-defined threshold, the controller adds a new node to the system. The work in [4] instead implements a controller driven using an MDP to model the cluster state and take optimal autoscaling decisions. Compared to PAX, these works do not support proactive autoscaling and suffer high synchronization latency due to the effect shown in Figure 1.

Recently, [5] presents a proactive controller for Cassandra that uses regression trees to predict latency. Upon exceeding a latency threshold, resources are scaled vertically to increase capacity and avoid SLO violations. Compared to this approach, PAX reasons on CPU utilization measurements, thus it is designed to optimize the infrastructure usage and cost, as opposed to user-perceived latency. Moreover, PAX adopts horizontal scaling and ARIMA time series forecasting of arrival rates. Recent work has shown that horizontal scaling tends to be more appropriate than vertical scaling for Cassandra databases [16].

The cost of adding or removing nodes to Cassandra and other databases has been measured in [11], [14], [15], [16], [20], [21]. For all the databases that do not use a shared filesystem, measurements indicate that the time required to add a new node is sensitive to the quantity of data stored in the entire database and the transmission rate between nodes. It is observed in [16] that the usage of higher transmission rate affects the response time of the read operations.

III. A Primer on Cassandra

Cassandra features a decentralized architecture composed of multiple nodes, arranged in a ring topology as shown in Figure 2. Each node is responsible to maintain, on its local storage, part of the database. Each node can then accept and reply directly to queries issued by the users. For a given query, the node that receives the request is termed *request coordinator* and is responsible to orchestrate the local or remote operations required to respond to the query.

Differently from other NoSQL databases, Cassandra uses the notions of Replication Factor (RF) and Consistency Level (CL). The Replication Factor (RF) controls the number of identical replicas of each datum across the cluster. The RF is defined at keyspace level and ensures high-availability in the presence of failures. The Consistency Level (CL) allows to control the consistency, strong or eventual, at the granularity of each individual query. CL may be set to ONE, TWO, QUORUM, or ALL. For reads, CL controls the number of copies of the datum that need to be retrieved before replying; for writes, CL controls the number of nodes that need to acknowledge the successful write. Note that reads in Cassandra are synchronous, whereas writes happen asynchronously with respect to the issue time of the write operation.

A. Data partitioning

Cassandra partitions the database into smaller, partially overlapping, datasets that are stored locally to each node. Thus, contrary to other NoSQL databases such as HBase, Cassandra does not require a shared filesystem (e.g., HDFS). A hash function is used to distribute the record primary keys across the nodes. This is done by partitioning the hash key range

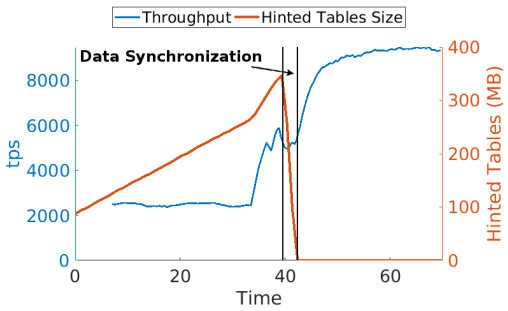

Fig. 3: Data synchronization through the hinted handoff.

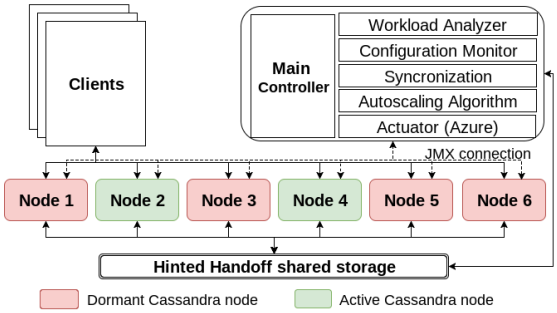

Fig. 4: PAX architecture

into sub-ranges called partitions (also called *TokenRanges*). In clusters without replication ($RF = 1$), each node i can be configured to locally store T_i unique partitions. The total number of unique partitions is thus $P = \sum_{i=1}^{N} T_i$, where N is the total number of nodes. For systems based on horizontal scaling, T_i is set to the same value on all the nodes, since the VMs are usually identical.

With replication ($RF > 1$), each node stores also some replicas. So, the total number of partitions available on a node i is $P_i = T_i \cdot RF$, where RF is the replication factor. For load balancing, the partitions are distributed randomly across the nodes with the constraint that a given partition can be stored only once on the same server.

B. Hinted handoff mechanism

When a node is not responding for a long period of time, Cassandra assumes that the node has failed. The hinted handoff mechanism will be tasked, when the node returns online, to ensure that data remains consistent. The mechanism works as follows. When the *request coordinator* believes that a node has failed, pending writes are stored locally within a hinted handoff table, one for each failed node. Each table record is called a *hint*. When the failed node becomes active again, before starting to serve client requests, it receives from all the other active nodes the tables and applies the changes on its local copy of the data. The transfer rate of the tables can tuned by the system administrator, and the overall synchronization time depends on the amount of writes received while the node was offline as shown is Figure 3. As active nodes may need to transfer large tables, this operation can reduce system performance until termination.

IV. PAX: Partition-aware autoscaling

In this section, we present PAX, the proposed partition-aware autoscaling method. Figure 4 illustrates the PAX architectural setup, which relies on three components: i) the Cassandra cluster, consisting of a fixed set of nodes (VMs), which can be either in active or dormant (i.e., powered-down) state; ii) the controller, which analyzes the workload measurements and actuates the autoscaling decisions; iii) a hinted handoff storage area, which archives the hints to be committed to the dormant nodes upon their return to the active state.

1) Controller: The core architectural element of PAX is the elasticity controller. The main aim of the controller is to ensure that the *average node utilization* U remains within a pre-defined CPU range $[U^-, U^+]$ at all times. Since PAX relies on horizontal scaling, it can be assumed that VMs have homogeneous sizes, and thus averaging utilization across nodes is a well-defined metric. It is possible to use other target metrics with the PAX controller, such as the maximum utilization across the nodes, but due to space limitations, we focus only on discussing the implementation of PAX for the average node utilization metric U. However, we have experimentally analyzed also the maximum utilization metric and observed that it leads to make the autoscaling more aggressive than the average node utilization. This is due to the skewed distribution of partitions across the nodes. In practice, we found that U is a sufficient metric to implement effective autoscaling, as shown later in the experimental results.

PAX can operate either as a reactive controller or as a proactive one. Let a *cluster configuration* be a particular assignment of the active and dormant states to the nodes. Also, a configuration is said to be *valid* if it guarantees that all data partitions have at least CL^{max} replicas stored in the active nodes, where CL^{max} is maximum consistency level allowed for a query. In the reactive mode, when the system performance is out of target CPU range, PAX interacts with the cloud provider API to adjust the configuration by starting and stopping VMs until U returns within the range. The only exceptions is when the cluster cannot scale up or down any longer, either due to shortage of dormant nodes or because it has reached a configuration that cannot use less nodes without becoming invalid. In the proactive mode, PAX couples this control mechanism with a workload forecasting method based on ARIMA processes. To better illustrate the operation of the controller, in the next subsection we review the workload analysis and forecasting features that are supplied to the controller by the architecture.

2) Workload analyzer (WA): The workload analyzer (WA) is responsible for monitoring the system, analyzing the resource consumption data and the demand estimation. When PAX is configured as a proactive controller, WA is also responsible of workload forecasting.

WA monitors the type of requests issued to the cluster

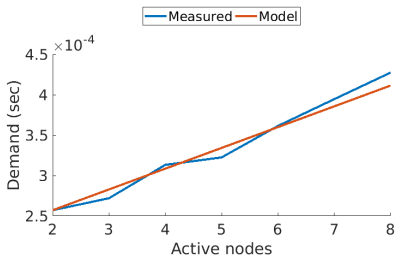

Fig. 5: Mean service demand change with the number of active nodes.

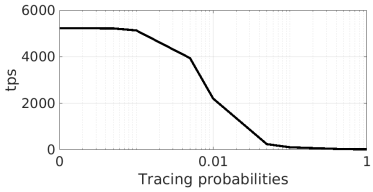

Fig. 6: Overhead of the tracing tool.

(read or write operations) and the requested primary keys. The main goal is to identify the partitions that are more frequently requested (*hot partitions*). This information is used by the PAX autoscaling algorithm to select the best dormant node to activate or, during scale down, to choose the active node to power off. While PAX does not require initial training about the hot data distribution across the nodes, it can converge faster to a good configuration if supplied with an initial estimate of the hot partitions, based on historical data. However, the controller works correctly also without this information.

At runtime, the WA can obtain the list of hot partitions either using the nodetool or the tracing utilities shipped with Cassandra. The command nodetool toppartitions samples the activity of a Cassandra cluster for a specified period of time, returning the hot partitions. However, the command can degrade the database performance and, in one of the latest Cassandra versions we used (3.0.9), the tool frequently crashed. For these two reasons, we have used in our implementation the tracing tool.

The tracing tool is a troubleshooting utility to profile the internal operations that are executed by Cassandra to complete a query. In order to prevent performance degradation, the tool allows to randomly sample queries with a given probability. Figure 6 shows the tracing overhead we observe on the same testbed used in Figure 1 for different sampling probability values. If the sampling probability is chosen small enough, the system performance is not significantly affected by the background execution of the tracing tool. Thus, we choose the maximum acceptable sampling probability for PAX to be 0.001, for which the system throughput is degraded by 2.9% on average, however PAX can also work smoothly with lower sampling probabilities, although it can take a longer time to converge to an optimal configuration. At runtime, every minute WA retrieves and clears from the database the tracing information provided by the tracing tool. The list of hot partitions is then calculated and made available to the controller.

A. Workload forecasting

The WA component also retrieves performance metrics for each node and predicts changes to the arrival rates of queries. In our implementation, performance metrics such as CPU utilization and the total number of operations executed are retrieved from each active Cassandra node using JMX. However, other monitoring systems may be used.

To predict the future workload, WA maintains an Autoregressive Integrated Moving Average (ARIMA) model. ARIMA is a method for non-stationary time series prediction that combines an autoregressive and a moving average model. It is commonly used in autoscaling mechanisms [22], [23], [24], [25], [26]. The ARIMA prediction is calculated taking in consideration the mean time required to boot up a dormant VM. In our experiments on Microsoft Azure based on VMs of class *A2*, we have measured this time to be around 3 minutes on average. ARIMA predictions are thus set at 3 minutes in the future.

WA predicts the global arrival rate λ to the cluster, measured in transactions-per-second (tps). Since we are concerned with the mean utilization across identical nodes, the value yields the predicted CPU utilization as [27] $U^{pred} = \lambda D$, where D is the mean service demand of a node and it is estimated by linear regression of the current measurements of U and λ. We have observed that the demand D changes almost linearly with the number of active nodes, as shown in Figure 5. This is mainly due to the increasing number of read requests issued by the request coordinator [28]. In our tests, each additional active node increases D on average by $\alpha = 10\%$. This correction is included upon forecasting the utilization after an autoscaling decision. In general, the value of α is sensitive to the consistency level CL of the queries. In other setups it may requires runtime estimation by fitting to a line the obtained measurements of U and λ as the cluster configuration changes.

B. Autoscaling algorithms

Central to the PAX autoscaling algorithm are the decisions on: i) which nodes to scale; ii) how many nodes to scale; iii) when to trigger the autoscaling action. We discuss these aspects separately in the next subsections.

1) Data-aware node acquisition: When the PAX controller decides to take an action, so to increase or decrease the number of active nodes, it is necessary to identify the set of nodes that are going to be involved in the action. This is decided based on the information generated by the WA component by prioritizing the acquisition of nodes including the hot data partitions. PAX associates to each Cassandra node a score (V_n). The algorithm receives from the WA components the primary keys (K) and the key access rate λ_k (request per second) for the queries randomly sampled in the last control period. To identify the location of the data accessed by the sampled query, the primary key is hashed and the partition ($p \in P$) that contains the data is identified. For each of the node that stores that partition,

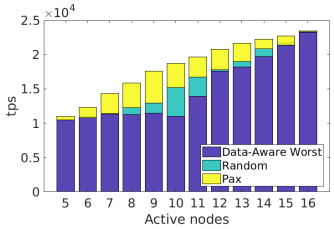

Fig. 7: Gains due to data-aware node acquisition.

the associate node value V_n is then incremented by λ_k. Thus, across the entire cluster

$$V_n = \sum_{p \in P_n} \sum_{k \in K_p} \lambda_k$$

where P_n is the set of local partitions on the node n and K_p is the set of primary keys contained in the partition p. The values V_n are used to decide the order in which dormant VMs are activated during scale-up or scale-down.

To assess the effectiveness of this data-aware scale-up approach, referred to as PAX (or data-aware *best*) selection , we show in Figure 7 a comparison against a method that selects the node with the *worst* V_n score and with a method that picks the node at *random*. The experiment is conducted on a 16 nodes Cassandra cluster where the minimum number of active nodes is 5 and the workload used a Zipfian popularity distribution for the data. To understand the best achievable performance, we use a static workload characterization. The remaining nodes are activated one by one until all the nodes are active. The performance gap between the two algorithms is quite evident and reaches a maximum with 10 active nodes. We attribute the fact that the gap is maximal around the middle to the fact that the number of possible combinations for choosing the active nodes is maximal at this point, subject to the configuration validity constraint, increasing the probability of errors for the random and worst schemes. Here the PAX selection reaches a throughput of 19070 tps, while the worst selection has only 11220 tps. Progressively, the gap is reduced since the offline nodes are fewer and thus the worst and random methods eventually pick up also the nodes with the hot partitions. The random algorithm shows as expected a performance in-between the two other methods, but it still fairly worse than the best one. Overall, this experiment confirms that data-awareness can produce visible gains during Cassandra autoscaling. We present in the next sections more advanced scenarios with dynamic workload characterization and varying experimental setups.

2) Number of nodes to scale: PAX offers three strategies to control the number of nodes involved in a scaling action. These strategies are called: conservative, average and aggressive.

During scale up, with the conservative strategy the smallest possible number of nodes that PAX predicts to bring back U within the target range is used. The aggressive strategy instead activates the largest possible number of nodes such that the predicted U remains above U^-. The average strategy targets instead the center of the target range. Viceversa, during scale down, the conservative policy switches off the maximum

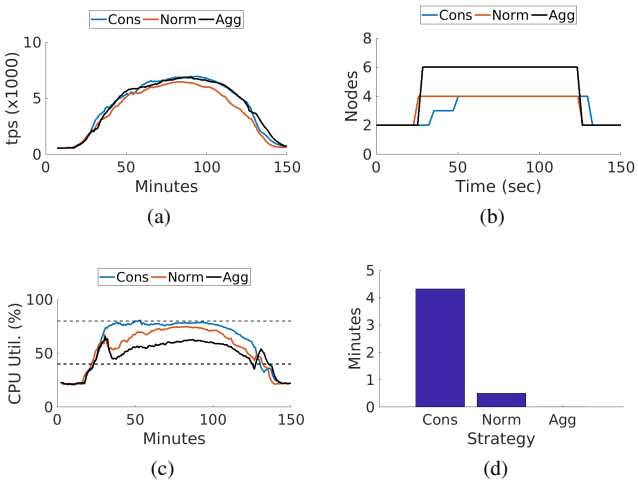

Fig. 8: Comparison between the three aggressive strategies: Conservative, Normal (average), Aggressive. The four images represent: a) throughput; b) number of nodes used over time; c) CPU utilization; d) T_u

number of node, while the aggressive ones powers down the minimum number of nodes.

Figure 8 presents experimental results for the three strategies. Figure 8(a) shows the throughput changes over time. The conservative reacts slowly to these changing, performing many more scaling up actions than the other policies, as seen in 8(b). As visible from Figure 8(c), this means that the conservative strategy remains close to the upper bound of the utilization band. Conversely, the aggressive strategy performs only 2 actions. However, it activates more nodes than the other strategies but it never violates the CPU upper bound, as shown in Figure 8(d). For this reason, we have decided to adopt in PAX by default the aggressive strategy.

3) Triggering a scaling action: The data-aware acquisition and the strategies to select the number of nodes allow PAX to implement a scaling decision. This involves CPU utilization prediction for all the possible node configurations, retaining only the valid ones within the target range. Among these, PAX selects the one with the desired number of nodes, based on the strategy selected, and with the best nodes, based on data-awareness.

As mentioned, PAX implements the decision in either reactive or proactive approach. In both cases, PAX requires the CPU trend needs to violate for at least 3 consecutive times, spaced by 1-minute intervals, the CPU bounds before the controller takes any action. This avoid unnecessary or inaccurate actions triggered by errors in the predictive model. Moreover, we set a stabilization period of time of 5 minutes in-between any two actions to reduces the likelihood of instabilities due to fluctuations in the measurements.

Figure 9 compares the proactive and reactive implementations of PAX and also illustrates the impact of data-awareness on these by considering the best data-aware node acquisition

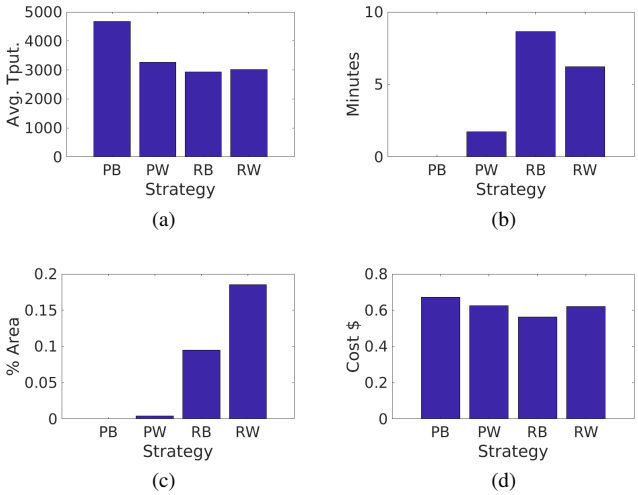

Fig. 9: Comparison between Proactive PAX (PB), Proactive Data-Aware Worst (PW), Reactive PAX (RB), Reactive Data-Aware Worst (RW). The figures represent: a) the average throughput; b) T_U; c) A_U; d) the cloud costs for the experiment.

of PAX against the worst-case acquisition method. The experiments run on a 8 nodes cluster with a 2 hours workload peaking at a maximum of 80 clients.

The metrics we collect are similar to those presented in [29]. We define the under-provisioning time T_U as the time that the system spends above U^+. We condition this to the system not using the maximum number of nodes, since we regard this situation as an error in static provisioning of the cluster size, rather than a shortcoming of the autoscaler. We also consider the under-provisioning area A_U between the cluster utilization and the upper bound of the target CPU range. This is computed only in periods when the utilization is above U^+.

The results show that the reactive controller introduces some under provisioning time due to the lack of utilization prediction. On the other hand, the proactive controller exhibits negligible under-provisioning time and area. However the proactive system anticipate some actions consuming more resources and with a higher experimental cost. The throughput results convincingly argue that the proactive data-aware controller performs much better than all the other methods.

Based on this analysis, we consider the reactive controller better suited in situations where the user wants to reduce the operational cost, when the user is renting the VM. On the other hand, the proactive controller limits considerably the under-provisioning time of the system making it more appropriate in that situations where the user wants to reduce CPU utilization, as in the case of providers exposing Cassandra services from within their own infrastructure. Given its increased ability to control the system, in the next sections, we will focus on evaluating the proactive version of PAX.

TABLE I: Performance comparison results.

	$T_i = 2$	$T_i = 256$
$N = 8$	8683 ops/sec	9067 ops/sec
$N = 16$	10663 ops/sec	10684 ops/sec

TABLE II: Minimum configuration size M under different T_i values. Experiments executed with a Cassandra cluster with $N = 8$ nodes, assuming a CL^{max} of ONE

	$RF = 2$		$RF = 4$	
T_i	P_i	M	P_i	M
1	2	4	4	2
2	4	4	8	2
4	8	5	16	3
8	16	6	32	3
32	64	7	128	4
128	256	7	512	5
256	512	7	1024	5

V. Tuning the PAX Architecture

In this section, we provide additional details concerning the PAX architecture, such as the number of nodes to be used in the cluster and the setup of the hinted handoff storage mechanism.

A. Cluster size

Upon instantiating the cluster, the administrator should decide several parameters such as the maximum number of nodes N, the RF, the maximum CL allowed for a query (CL^{max}), and the data partitioning setup. These decisions affect the flexibility of the autoscaling mechanism, in particular the minimum number of machines M that need to remain online at all times to ensure that the configuration remains valid.

1) Choosing the total number of nodes: The maximum number of nodes N should be such that the cluster can achieve the maximum target throughput when all nodes are online. Benchmarking may be used to estimate the N parameter experimentally against a reference workload. N can also be changed at runtime where needed, but as observed in Figure 1 the synchronization time will be much longer than with the hinted-handoff mechanism, which requires only a few minutes.

2) Choosing the replication factor: Normally, RF is chosen in production environments to be between 2 and 4. RF of 1 is usually not advised since, in case of failure, the system may not be able to recover the data. Probabilistic methods have been devised to analyze the influence on RF on the system availability and resilience to malicious users and identify an appropriate assignment [30], [31], [32].

3) Choosing the data partitioning: The RF and the number of partitions per node T_i are static properties of the cluster decided upon its creation and the initial loading of the dataset. They determine the set of valid configurations for a cluster and thus the flexibility of the autoscaling mechanism. Although T_i does not significantly affect performance, as shown in Table I for a 8-node cluster, it influences the placement of data on the nodes, and thus how many nodes can be turned off by the autoscaling controller while remaining in a valid configuration.

TABLE III: Testbed configuration used for the controller evaluation.

	Node	YCSB client	Controller
Number of VMs	8	4	1
VM type	General purpose		
O.S.	Ubuntu 16.04.2 LTS		
vCPUs	2	2	4
Memory	3.5GB	3.5GB	7GB
O.S. Disk	30GB Read/Write Host Caching		
Data Disk	80GB (No Caching)	none	none

TABLE IV: YCSB Workload characteristic used for the system evaluation

Workload	Read	Write	Distribution
A	50%	50%	Zipfian
B	95%	5%	Zipfian
C	100%	0%	Zipfian
G	100%	0%	Latest

The dependence is illustrated in Table II. Here, we repeat a set of 14 experiments, changing the T_i and RF values. For example, the first row considers the case where each node contains $T_i = 1$ unique partitions. In this case, out of the $N = 8$ nodes, 4 nodes should always remain online if $RF = 2$, but this reduces to 2 if $RF = 4$. However, larger RF values increase costs, since more storage capacity and nodes will be needed to replicate the data.

B. Hinted handoff storage

To ensure that each node can retrieve all hinted handoff tables anytime, we recommend to setup a shared storage area among the cluster nodes. By default, these tables are stored locally on each Cassandra node disk. However, it may happen that some tables are stored on nodes that become dormant, causing the risk that a newly activated node does not find all hinted handoff tables required for its synchronization.

A shared storage area avoids the above issue. Within this area, each node can store its tables in a specific folder, so that they are always available for retrieval during scale up operations. When a dormant node is powered up, it receives the data from all the active nodes and after it applies also the tables found in the shared area.

Although we did not experience similar cases, it is conceivable that if some nodes remain dormant long enough, the size of the hinted handoff tables may grow large enough to become an issue for both performance and cost. Several strategies are possible to mitigate this risk. One possible solution could be to adopt a hybrid architecture, where most hints are stored locally to the nodes and only the ones of the dormant nodes are in the storage area. A simpler alternative consists in periodically activating the dormant nodes to allow them to synchronize the pending hints. Platform such as Azure allows to pay only for the minutes effectively used to perform the synchronization.

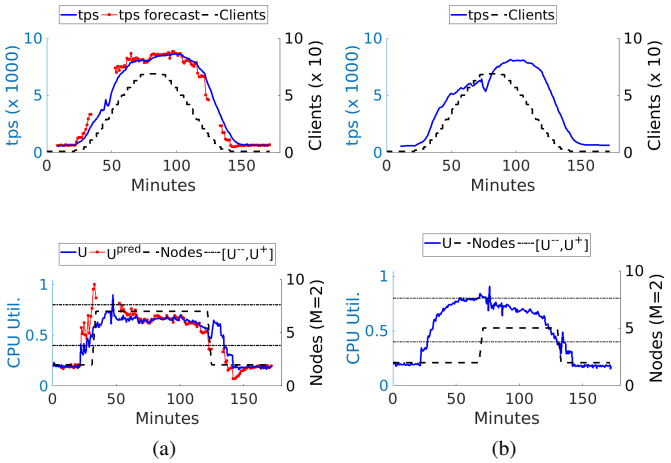

Fig. 10: Controller benchmark using PAX algorithm. The experiments represent the controller behavior when a peak behavior with maximum 80 clients using workload B using the approach: a) proactive; b) reactive.

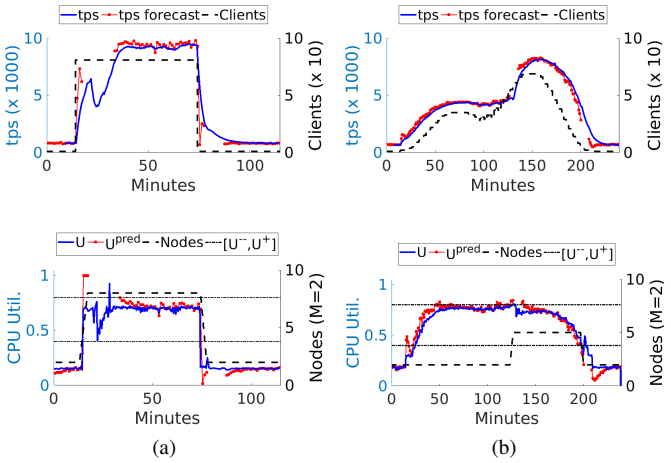

Fig. 11: PAX controller response to a) a step of 80 clients starts issuing YCSB workload A; b) two overlapped peaks and workload C.

VI. Performance Evaluation

A. Methodology

The evaluation of our elastic system is conducted on Microsoft Azure cloud with a Cassandra cluster composed of 8 nodes. The hardware characteristics of the virtual machines are presented in Table III. On each Cassandra node is installed Sun Java 1.8 and Cassandra 3.0.9. In the default Cassandra configuration file, we modify the *num_tokens* to 2 to allow us to have a larger elastic range (from 2 to 8 VMs) with a CL of ONE and the hints folder is redirected to the shared disk managed by Azure. We also install and enable Jolokia to expose the JMX metrics over HTTP. The Cassandra database is loaded with a total of 180 GBs of data using a replication factor of 4.

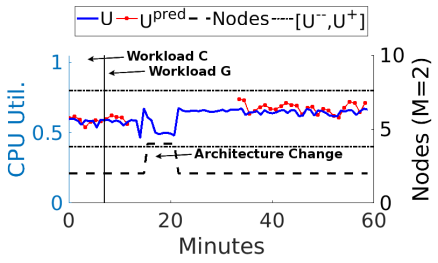

Fig. 12: PAX response to changes in the hot partitions.

TABLE V: Evaluation results for PAX.

	max N	T_U	A_U	$/min	$ saving
Fig. 10a	7	30s (0.41%)	0.07%	0.0059	33.56%
Fig. 10b	5	505s (3.54%)	0.20%	0.0038	56.36%
Fig. 11a	8	21s (0.3%)	0.12%	0.0074	11.53%
Fig. 11b	5	184s (1.28%)	0.007%	0.0037	61.42%

Compared to the state-of-the-art in Cassandra autoscaling presented in Section II, our setup has a larger database with a higher replication factor. Using the default architecture, these factors significantly impact on the data synchronization time necessary to the system to change the configuration since the amount of data that each node needs to transfer is bigger.

The workload generation machines run the YCSB benchmark [12]. The YCSB workload characteristics are summarized in Table IV. We deploy in total 4 workload generator virtual machines and one VM for the PAX controller.

B. Step response

Figure 11a presents the performance of the PAX controller under a step increase of 80 clients, using the workload A. The top figure shows the increase in arrival rate (in tps) and clients. The bottom figure shows the response of the PAX controller in terms of number of nodes, actual utilization U of the testbed and predicted utilization values by the ARIMA forecasting. In these and the following graphs, the predicted throughput and CPU are represented by the dot marker. In-between any two markers, *no* prediction is performed since the controller awaits that the system stabilizes to retrain the ARIMA process and resume the forecasting.

In the experiment, the system does not have any information about the future, the controller is, as expected, not able to anticipate the sudden step increase but, immediately after it reacts correctly to it by starting new VMs that avoid the utilization to step out of the target range, except for a negligible period as reported in Table V.

C. Comparing proactive and reactive approaches

Figure 10a and Figure 10b compares the proactive and reactive controller using a workload B, lasting two hours and exhibiting a peak in the number of active clients.

It is possible to notice that the reactive controller consume less resources, saving around 0.126$ per hour. However, the reactive implementation has lower throughput than the proactive one. In addition, by using less resources, the average CPU utilization of the reactive method is higher than in the proactive case. The proactive controller reduces significantly the under-provisioning of the system and it supports better the time-varying workload.

D. Response to overlapped peaks

Figure 11b presents an experiment with two successive peaks in the number of clients and based on workload C. As the clients grow, the CPU utilization sharply reaches the upper bound. As this is a rather slow growth pace, the ARIMA predictor suggests that this peak can be handled with the current testbed and this is indeed the case. However, as the second larger and growing faster peak arrives, the ARIMA controller is able to anticipate it in the initial stages, shortly after 100 minutes, activating other 3 VMs to handle the peak workload effectively. Even if the system presents a higher T_u time compared to the other proactive experiments (Table V), the A_u is very low meaning that the system is really closed to the U^+ bound. In addition, this experiments presents the higher cost saving of 61.42% respect a traditional Cassandra implementation where all the nodes are always active.

E. Architecture change

As discussed in Section IV-B, the configuration can significantly influence performance. Here we show the ability of the WA query sampling to trigger actions in response to a change of the query mix issued by the clients.

The results are presented in Figure 12. During the experiment execution, the workload generated by YCSB changes from workload C to G, modifying the primary key access rates. Using the WA information, the PAX controller determines a better configuration and starts and stops nodes to change the set of active partitions.

When a new configuration is detected, the controller activates it and during the stabilization period the two configurations are active both at the same time. Then the VMs of the old configuration become dormant. The configuration changes significantly benefits throughput, which increases from 1190 to 2500 tps. This illustrates the benefits of data-aware autoscaling in Cassandra.

VII. Conclusions and Future Work

In this paper, we have proposed PAX, a new auto-scaling system for Cassandra. PAX leverages the hinted-handoff mechanism of Cassandra to reduce the data synchronization period when a new node is added to the cluster. Based on this, we have defined reactive and proactive controllers that profile the current workload and use this information to activate the Cassandra nodes that store hot data partitions. We have found that both reactive and proactive implementations are useful in practice, with the reactive method using less VMs but incurring more frequently under-provisioning, while the proactive allowing negligible violations of the target utilization.

In future work, we would like to extend our controller to support also the response time metric in order to evaluate response to burstiness in workloads.

REFERENCES

[1] J. Pokorny, "Nosql databases: a step to database scalability in web environment," *International Journal of Web Information Systems*, vol. 9, no. 1, pp. 69–82, 2013.

[2] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[3] M. Chalkiadaki and K. Magoutis, "Managing service performance in the cassandra distributed storage system," in *Proceedings of the 2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 1, Dec 2013, pp. 64–71.

[4] A. Naskos, E. Stachtiari, A. Gounaris, P. Katsaros, D. Tsoumakos, I. Konstantinou, and S. Sioutas, "Harmon," in *Proceedings of the 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2015, pp. 31–40.

[5] Y. Kishore, N. H. V. Datta, K. V. Subramaniam, and D. Sitaram, "Qos aware resource management for apache cassandra," in *Proceedings of the 2016 IEEE 23rd International Conference on High Performance Computing Workshops (HiPCW)*, Dec 2016, pp. 3–10.

[6] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Computing Surveys*, 2016.

[7] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[8] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *CoRR*, vol. abs/1609.09224, 2016.

[9] D. Agrawal, A. El Abbadi, S. Das, and A. J. Elmore, "Database scalability, elasticity, and autonomy in the cloud," *DASFAA (1)*, vol. 6587, pp. 2–15, 2011.

[10] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Information Sciences*, vol. 275, pp. 314–347, 2014.

[11] D. Seybold, N. Wagner, B. Erb, and J. Domaschka, "Is elasticity of scalable databases a myth?" in *Proceedings of the 2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016, pp. 2827–2836.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154.

[13] Datastax. (2017, sep) Hinted handoff: repair during write path. [Online]. Available: https://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsRepairNodesHintedHandoff.html

[14] I. Konstantinou, E. Angelou, C. Boumpouka, D. Tsoumakos, and N. Koziris, "On the elasticity of nosql databases over cloud management platforms," in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, ser. CIKM '11. New York, NY, USA: ACM, 2011, pp. 2385–2388.

[15] T. Dory, B. Mejías, P. Van Roy, and N.-L. Tran, "Comparative elasticity and scalability measurements of cloud databases," 2011.

[16] J. Kuhlenkamp, M. Klems, and O. Röss, "Benchmarking scalability and elasticity of distributed database systems," *Proceedings of the VLDB Endow.*, vol. 7, no. 12, pp. 1219–1230, Aug. 2014.

[17] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris, "Automated, elastic resource provisioning for nosql clusters using tiramola," in *Proceedings of the 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, May 2013, pp. 34–41.

[18] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. a. Paulo, J. Pereira, and R. Vilaça, "Met: Workload aware elasticity for nosql," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 183–196.

[19] A. Al-Shishtawy and V. Vlassov, "Elastman: Elasticity manager for elastic key-value stores in the cloud," in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, ser. CAC '13. New York, NY, USA: ACM, 2013, pp. 7:1–7:10.

[20] 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," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1724–1735, Aug. 2012.

[21] I. Konstantinou, E. Angelou, D. Tsoumakos, C. Boumpouka, N. Koziris, and S. Sioutas, "Tiramola: elastic nosql provisioning through a cloud management platform," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 725–728.

[22] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload prediction using arima model and its impact on cloud applications's qos," *IEEE Transactions on Cloud Computing*, vol. 3, no. 4, pp. 449–458, Oct 2015.

[23] J. M. Tirado, D. Higuero, F. Isaila, and J. Carretero, "Predictive data grouping and placement for cloud-based elastic server infrastructures," in *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Computer Society, 2011, pp. 285–294.

[24] S. Barker, Y. Chi, H. Hacigümüs, P. Shenoy, and E. Cecchet, "Shuttledb: Database-aware elasticity in the cloud," in *Proceedings of the 11th International Conference on Autonomic Computing (ICAC 14)*. Philadelphia, PA: USENIX Association, 2014, pp. 33–43.

[25] W. Fang, Z. Lu, J. Wu, and Z. Cao, "Rpps: A novel resource prediction and provisioning scheme in cloud data center," in *Proceedings of the 2012 IEEE Ninth International Conference on Services Computing*, June 2012, pp. 609–616.

[26] Q. Zhang, M. F. Zhani, R. Boutaba, and J. L. Hellerstein, "Harmony: Dynamic heterogeneity-aware resource provisioning in the cloud," in *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems (ICDCS),*. IEEE, 2013, pp. 510–519.

[27] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[28] S. Dipietro, G. Casale, and G. Serazzi, "A queueing network model for performance prediction of apache cassandra." ACM, 5 2017.

[29] N. R. Herbst, S. Kounev, and R. H. Reussner, "Elasticity in cloud computing: What it is, and what it is not." in *ICAC*, vol. 13, 2013, pp. 23–27.

[30] H. Yu and P. B. Gibbons, "Optimal inter-object correlation when replicating for availability," in *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '07. New York, NY, USA: ACM, 2007, pp. 254–263.

[31] P. Li, D. Gao, and M. K. Reiter, "Replica placement for availability in the worst case," in *Proceedings of the 2015 IEEE 35th International Conference on Distributed Computing Systems*, June 2015, pp. 599–608.

[32] H. Yu, P. B. Gibbons, and S. Nath, "Availability of multi-object operations," in *Proceedings of the 3 USENIX Symposium on Networked Systems Design and Implementation*, 2006, pp. 211–224.