# Managing Service Performance in the Cassandra Distributed Storage System

Maria Chalkiadaki
Institute of Computer Science (ICS)
Foundation for Research and Technology–Hellas (FORTH)
Heraklion, GR 70013
Email: mhalkiad(at)ics.forth.gr

Kostas Magoutis
Institute of Computer Science (ICS)
Foundation for Research and Technology–Hellas (FORTH)
Heraklion, GR 70013
Email: magoutis(at)ics.forth.gr

*Abstract*—**In this paper we describe the architecture of a quality-of-service (QoS) infrastructure for achieving controlled application performance over the Cassandra distributed storage system. We present an implementation of our architecture and provide results from an evaluation using the Yahoo Cloud Serving Benchmark (YCSB) on the Amazon EC2 Cloud. A key focus of this paper is on a QoS-aware measurement-driven provisioning methodology. Our evaluation provides evidence that the methodology is effective in estimating application resource requirements and thus in achieving the type of controlled performance required by data intensive performance-critical applications. While our architecture is implemented and evaluated in the context of the Cassandra distributed storage system, its principles are general and can be applied to a variety of NoSQL systems.**

## I. Introduction

The new breed of *NoSQL* distributed storage systems has dramatically changed the landscape of how information is represented, manipulated, and stored in large-scale infrastructures today. These systems are currently at the forefront of academic research and industrial practice, primarily due to their high scalability and availability features. Managing service performance over these systems is an area that attracts significant interest as manifested by the success of early service offerings in this space (e.g., DynamoDB [1]).

In this paper we present a quality of service (QoS) architecture and prototype that offers managed service performance over a prominent NoSQL system, Apache Cassandra. Our architecture is able to address the *storage configuration* problem, namely to appropriately provision initial storage resources for a target workload given a simple description of its characteristics. It is also able to address the *dynamic adaptation* problem by monitoring service performance at runtime and adjusting to short-term variations by either throttling the application or by expanding the set of resources assigned to it. Our focus in this paper is primarily on the first problem: our solution to it is based on a methodology using targeted measurements to produce a set of tables expressing benchmark performance over different configurations of Cassandra (number, type of servers). Using the produced tables, we can estimate the resources required to achieve the service-level objective (SLO) of an application interpolation from the baseline measurements.

In previous work we described a preliminary design of our QoS architecture and evaluated an early prototype of it [2]. In this paper we extend our earlier work in several directions:

- SLA-driven initial provisioning
- Accounting for both response time and throughput
- QoS controller is an independent component, periodically communicating with clients via RPC
- Evaluation on industry-leading Amazon EC2 Cloud
- Ability to share servers (VMs) across independently scaled workloads (e.g., different users/tables)
- Improved robustness of our prototype

Our key contributions in this paper are: A novel methodology for configuring Cloud-based Cassandra storage clusters for specific application SLOs; a dynamic QoS monitoring and adaptation mechanism to control short-term workload variations; and an extensive evaluation of our methodology in Amazon Web Services' EC2 Cloud. Our exposition proceeds as follows: in Section II we discuss related work in this space. In Sections III and IV we describe our design and implementation in the context of Apache Cassandra. In Section V we present our evaluation, and finally in Section VI we conclude.

## II. Related work

Distributed data stores (often referred to as key-value stores) that implement distributed tabular structures with configurable access semantics have recently been developed as research prototypes as well as commercial systems to support a number of rapidly-growing large-scale data-centric enterprises. Examples of such systems include Dynamo [3], Bigtable [4], and their open-source variants Cassandra [5] and HBase [6]. Cloud service offerings of these technologies are currently widely available, offering a broad range of performance and dependability characteristics.

As enterprises that have invested into Cloud computing are now raising their expectations from best effort to guaranteed levels of service, Cloud providers offer versions of their data-centric services that support controlled performance, reliability, etc. Recently Amazon Web Services (AWS) announced two new versions of existing services that offer guaranteed read/write I/O throughput on a key-value store (this service is branded *DynamoDB*) and provisioned I/O throughput over its elastic block storage (service branded *provisioned IOPS*).

Providing quality of service over distributed storage has been an active area of research for at least two decades.

Work at HP labs (a retrospective by John Wilkes provides a good overview of this work [7]) addressed a wide range of concerns, from specifications of workloads, QoS goals, and device capabilities, to mappings of workload onto underlying storage resources, and to run-time management of storage I/O flows. It is worth noting that while QoS in networking is a relatively mature field whose numerous research results have progressed in many cases into formal protocol specifications and products, storage QoS is a less mature area due to the significantly more challenging technical issues involved (such as for example non-linear behavior due to caches and the strong dependance on workload characteristics).

Work by Goyal et al. [8] in the context of the CacheCOW system contributed algorithms for dynamically adapting storage cache space allocated to different classes of service depending on observed response time, temporal locality of reference, and the arrival pattern for each class. The focus of this work was on centralized storage controllers rather than distributed servers typically used in NoSQL systems. More recently, Magoutis et al. [9] presented a self-tuning storage management architecture that allows applications and the storage environment to negotiate resource allocations without requiring human intervention. The authors of this work aim to maximize the utilization of all storage resources in a storage area network subject to fairness (rather than user-defined service-level objectives, as we do in this paper) in the allocation of resources to applications.

With AWS being the current industry leader in guaranteed performance over distributed Cloud storage, it is worth taking a deeper look into their published and commercial work. Their SOSP paper [3] describes their (internal at the time) Dynamo key-value data store service which offered service-level agreements (SLA) on the response-time of put/get operations (e.g., service-side completion within 300ms) offered by the service measured on the $99.9^{th}$ percentile of the total number of requests, assuming the client does not exceed a peak level of load (e.g., 500 requests / sec). The recently introduced DynamoDB [1] is based on the published design of Dynamo with the introduction of new technologies such as solid-state storage (SSD) to address reliability issues.

DynamoDB departs from the original Amazon design in its SLA specification. Namely, a user specifies performance requirements on a database table in terms of *request capacity* or number of 1KB read or write operations (also known as units of read or write capacity) desired to be executed per second. DynamoDB allocates dedicated resources to tables to meet performance requirements, and automatically partitions data over a sufficient number of servers to meet request capacity. If throughput requirements change, the user can update a table's request capacity on demand. Average service-side latencies for Amazon DynamoDB are reported to be in the single-digit milliseconds range [1]. Applications whose request throughput exceeds their provisioned capacity may be throttled. DynamoDB does not seem to provide any guarantees on the response time offered nor on the distribution of requests on which their offered performance is evaluated (e.g., $99.9^{th}$ percentile over some time range).

Two of the most widely deployed NoSQL distributed storage systems are HBase [6] and Cassandra [5]. HBase tables contain rows of information indexed by primary key. The basic unit of data is the column, which consists of a key and a value. Sequences of columns (an arbitrary number) collectively form a row. A number of logically-related columns can be grouped into column families (CFs), which are kept physically close in both memory and disk. HBase partitions data using a distributed multi-level tree that splits each table into Regions and stores Region data in the HDFS distributed file system using a scheme similar to LSM trees [10].

Cassandra is an open source clone of Dynamo, combining some features (such as column families, and storage management based on LSM trees over local storage) from HBase. Each node in a Cassandra cluster maps to a specific position on a ring via a consistent hashing scheme [5]. Similarly, each row maps to a position on the ring by hashing its key using the same hash function. Each node stores all rows whose keys hash between this node's position and the position of the previous $n$ nodes on the ring when replicating $n$ times. Cassandra leverages an LSM-tree like scheme similar to that used by HBase to store data except that individual files (called SSTables) are stored in each node's local file system as opposed to a distributed file system. When reading a row stored in one or more SSTables, Cassandra uses a row-level column index (and optionally a Bloom filter) to find the necessary blocks on disk.

The idea of measurement-based performance modeling has been previously proposed by Anderson [11] in the context of storage system design and configuration of RAID arrays [7]. Complex performance evaluations of systems have been studied by Westermann et al. [12], [13], including methods for predicting application performance based on statistics over a space of measurement data. Our methodology is closely related to these approaches as we rely on a guided exploration of system configuration space under different workload assumptions. We differ from them on our focus on service-level management of scalable and elastic NoSQL technologies such as Cassandra.

Specifying SLA requirements and obligations in machine-readable form requires standards such as WS-Agreement by the Open Grid Forum and WSLA by IBM [14]. WS-Agreement defines a protocol for negotiating and creating agreements between clients and service providers and for monitoring compliance using Web services. The goal of WSLA is to allow the creation of machine-readable SLAs for services implemented using Web services technologies. Although not specifically addressed in this paper, these standards can be straightforwardly leveraged for expressing SLAs over NoSQL technologies such as Cassandra.

## III. Design

Our architecture for service-level management over the Cassandra distributed storage system is depicted in Figure 1. The QoS controller is the core component of this architecture. Its key functionalities are to *(i)* setup SLAs with application clients, requesting their CF profiles (data set size and a coarse characterization of their degree of locality, such as random, zipf-like, etc. per CF) and performance requirements (currently focusing on satisfying response-time targets at certain throughput rates); *(ii)* effect initial resource allocations for the application; *(iii)* periodically collect monitored response-time and throughput metrics from Cassandra clients and plan

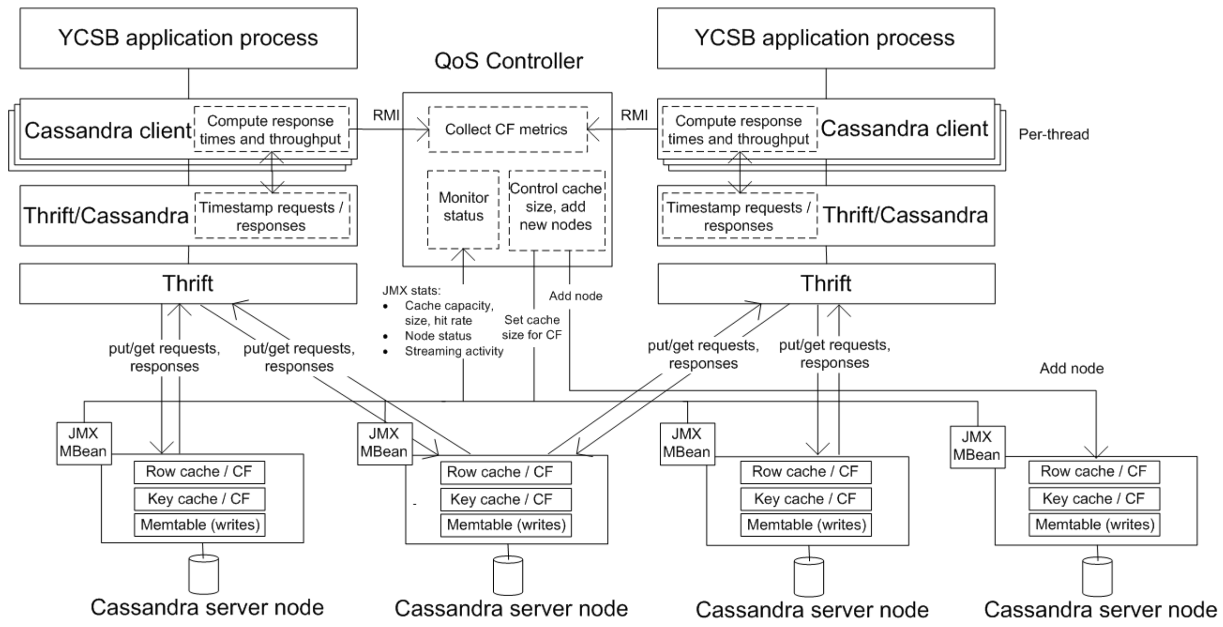


Fig. 1: The Cassandra QoS architecture

and effect changes in resource allocations to better align with requested targets; and *(iv)* perform admission control by estimating overall resource utilization and level of satisfaction of requirements for current commitments.

In this paper we primarily focus on the problem of initial resource allocation. We describe a provisioning policy based on predictions of service capacity requirements for applications. Our methodology relies on a set of performance tables (such as Table I) of measured performance results (throughput and response time) produced by a configurable load generator. We use as load generator the Yahoo Client Serving Benchmark (YCSB) [15] configured to produce a specific access pattern, I/O size, and read/write ratio (collectively termed a workload $W$) and server (VM) type ($S$). YCSB performs 1KB accesses with configurable read/write ratio using Zipf (featuring locality) or uniformly-random probability distributions.

| | Workload: $W$; Server type: $S$ | | | | |
|---|---|---|---|---|---|
| # Servers / # Clients | 1 | 2 | 3 | 4 | ... |
| clients$_1$ | $r_1$, $t_1$ | $r_2$, $t_2$ | $r_3$, $t_3$ | $r_4$, $t_4$ | $r_5$, $t_5$ |
| clients$_2$ | $r_1$, $t_1$ | $r_2$, $t_2$ | $r_3$, $t_3$ | $r_4$, $t_4$ | $r_5$, $t_5$ |
| clients$_3$ | $r_1$, $t_1$ | $r_2$, $t_2$ | $r_3$, $t_3$ | $r_4$, $t_4$ | $r_5$, $t_5$ |
| ... | $r_1$, $t_1$ | $r_2$, $t_2$ | $r_3$, $t_3$ | $r_4$, $t_4$ | $r_5$, $t_5$ |

TABLE I: Response time, throughput for variable load levels, service capacities (for given workload, server types)

Given an application's access pattern and desired load level, we can determine the number of servers of a given type required to achieve the desired throughput without exceeding a response time threshold. When the desired application characteristics or load levels do not exactly match a table entry we apply interpolation from neighboring table entries.

Each Cassandra client (usually embedded into an application) performs per-thread measurements of response time and computes exponentially-weighted moving averages (EWMAs) of response-time values using the following formula, where r*(T)* is the response time sampled at time T and $\alpha$=0.125.

$$\text{EWMA}(T) = (1-\alpha) * \text{EWMA}(T-1) + \alpha * \text{r}(T)$$

Each process computes response time EWMA and aggregate throughput across all its threads and communicates both to the QoS controller. The QoS controller combines the reported metrics across YCSB processes belonging to the same user. It uses these metrics to take control actions, such as increase/decrease I/O path parallelism, and optionally regulate cache assigned to a CF [2] or throttle an application [9], [16].

The QoS controller is able to simultaneously interface and control multiple independent users (representing different workoads). Since multiple applications executing over the same Cassandra cluster cannot normally be isolated in terms of elasticity policies (e.g., allow application A to grow the Cassandra cluster by one server while application B sees its previous configuration), our design assigns each application to its own independent Cassandra cluster (still allowing Cassandra servers to share Cloud VMs) as shown in Figure 2. Finally, for high availability we support a primary-backup scheme using a shadow QoS controller to replicate the state of the primary. Details of this approach are beyond the scope of this paper.

## IV. Implementation

Our implementation is based on Apache Cassandra version 1.0.10. YCSB is used as the canonical example of an application throughout this section; the architecture however is general and applies to any application that can run over the Cassandra client library. Our implementation extends YCSB Cassandra Client version 1.0.10 (hereafter referred to as Cassandra-Client10). The Cassandra server-side code is unmodified. For

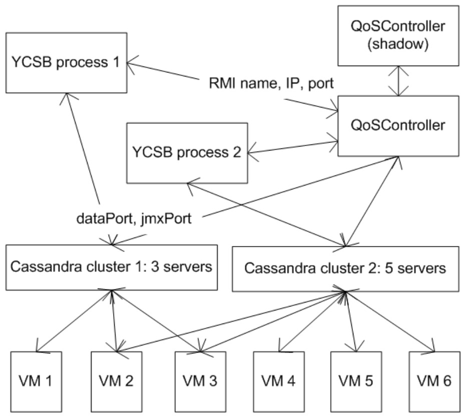


Fig. 2: Relationships between application processes, QoS controller, Cassandra clusters, and the Cloud infrastructure. The shadow QoS controller forms a primary-backup pair with the main QoS controller for high availability

simplicity we assume single datacenter and rack, one CF per cluster, single seed node (simple snitch), random partitioner, and replication factor one. In Figure 1 solid boxes denote existing components while dotted boxes denote our extensions.

To support our enhanced functionality (QoS attributes of multiple users accessing different CFs and clusters), we have added the following YCSB command-line arguments:

- -rt : desired response time for read/write requests.
- -throughput : desired throughput for read/write requests.
- -QoS{Port, Name, IP} : RMI connection port, name, address between YCSB user and QoSController.
- -dataPort : RMI port for data connection between YCSB user and Cassandra cluster (each user is mapped to a separate cluster).
- -cf : column family name.

Another key YCSB parameter is the statistical distribution of requests, which can be uniformly random or Zipf (locality). A YCSB application accesses a single Cassandra CF and involves multiple concurrently executing load-producing threads. Each thread uses a unique CassandraClient10 object.

**Monitoring**. We modified the Cassandra Thrift implementation depicted in Figure 1 to timestamp read and write operations. CassandraClient10 computes read/write latency (ms) and read/write throughput (MB/s) (using the Cassandra Thrift timestamps) and stores them into a (per-thread) CassandraQoS object. CassandraQoS computes the EWMA of response times and estimates throughput by dividing the bytes transferred for completed operations over a given time period. We have added a *StatGathering* thread to YCSB to periodically (every 30 seconds) collect from all CassandraQoS objects their response-time EWMAs and throughput values. *StatGathering* computes the average of EWMAs and aggregate throughput across YCSB threads. QoSController collects those numbers from each YCSB process via periodic RMI calls.

**QoS controller**. The QoSController is a separate process executing on a dedicated node. To enable the QoS controller to simultaneously control independent YCSB workloads we:

- Allow each YCSB process to have access to a different Cassandra server cluster. To allow sharing of VMs across clusters, network ports (data and JMX) used by a cluster are set per YCSB process.
- Give each YCSB process a separate RMI connection to the QoSController.

Each YCSB process communicates with the QoSController over its assigned RMI name, IP, and port, and passes to it configuration information followed by the user's SLA.

Configuration information includes parameters of its Cassandra cluster: Name of cluster; initial and max number of Cassandra servers; IP addresses of servers and seeds; data port (for read/write operations) and JMX port for monitoring and managing Cassandra servers. A Cassandra cluster initialized for a specific YCSB process takes as parameters the ports all servers should listen to, e.g. data port 9160, JMX port 7199 for *Cluster0*; data port 9161, JMX port 7198 for *Cluster1*; etc.

The QoSController starts JMX connections to each Cassandra server in a cluster and periodically (every 30 seconds) collects statistics on row cache capacity, current size, and hit ratio. When row caches fill up (past a ramp up phase after an elasticity action), the QoSController goes over a 10 minute period during which it checks I/O response times and throughput (20 times) for compliance with the user-specified SLO. If the SLO is violated, it decides to start a new server to further distribute the load in the cluster.

**Elasticity**. When the QoSController starts a new server in a cluster, it initiates a JMX connection to that server and uses it to set specific attributes for that server such as row cache capacity and to get information about the server it will stream data from to balance the ring. When data streaming is complete, the new server transitions into normal mode in the cluster and is ready to receive client requests. At that point, a cleanup thread in the QoSController deletes keys from offloaded nodes using the JMX `cleanup` API.

A challenge we faced early on was that the standard version of YCSB (as of version 0.1.4) does not take elasticity into account: it statically binds to an initial set of Cassandra servers and cannot dynamically redistribute load to an expanding cluster. In our extended version, we check in YCSB whether a new server has been inserted in the cluster (ring) before each read or write operation (by checking the *host* attribute in the list of properties) and if so we include the new server. Each CassandraClient10 then re-selects the Cassandra server to which it binds to and sends its requests. To spread the client load uniformly over the servers we map each client to a server by taking the modulo of the client identifier over the total number of servers. This dynamic reassignment of clients to servers is performed each time a new server enters the cluster.

**Caching**. At the Cassandra server side, we use fixed-size row caches per CF, set using the JMX `setCapacity` method exported by storage servers. In our earlier versions of our implementation using JVM heap for cache memory we were careful regulating these caches to avoid exceeding a certain fraction of the total heap size. Our experience indicates that exceeding that limit triggers frequent garbage collection (GC) activity and leads to automatic cache-size reduction by Cassandra. Our current implementation configures Cassandra servers to use off-JVM heap memory (and thus not GC'ed) for its caches (row cache, key cache, and memtable). We thus avoid some of the cache-related memory pressure effects that impact Cassandra performance in unpredictable ways. We do not however fully prevent such activity, which is inherent in Java implementations of data-intensive distributed systems.

## V. Evaluation

We evaluated our system on the Amazon Web Services (AWS) EC2 Cloud using two different Cassandra clusters. The first one (referred to as SMALL) consists of 7 servers of type AWS *m1.small* featuring 1 virtual core with Intel Xeon processors, 1.7GB DRAM, 160GB local (instance) storage. The second one (MEDIUM) consists of up to 5 servers of type AWS *m1.medium* featuring 1 virtual core with Intel Xeon processors, 3.75GB DRAM, 410GB local storage. The server operating system is Linux Ubuntu 10.4.1 LTS, 64 bits. The Cassandra software version (baseline) is 1.0.10 using the OpenJDK 1.6.0-24 Java runtime environment with heap size of 1GB. Our evaluation workload is the Yahoo Cloud Serving Benchmark (YCSB) version 0.1.4. The YCSB workload executes on an EC2 instance of type AWS *m1.large* (2 virtual CPUs, 7.5 GB DRAM, 840GB local storage). The QoSController process executes on a dedicated AWS *m1.small* EC2 instance.

To exhibit our QoS-aware provisioning methodology we focus on two distinct types of applications: those that exhibit locality in table accesses and those that do not. We emulate both by configuring YCSB to produce accesses based on (a) a Zipf probability distribution; and (b) a uniformly-random probability distribution. According to the Zipf distribution, some records are extremely popular while most records are unpopular. In addition, we have disabled the key cache to focus on the characteristics of the row cache alone.

In the first part of our evaluation we produce instances of Table I by progressively expanding I/O path parallelism between the application and storage servers (via elasticity actions) as much as needed to match the selected workloads (no SLO is set in this phase). Our evaluation considers AWS *m1.small* and *m1.medium* types; our methodology however extends to coverage of other VM types. We use our extended version of the YCSB benchmark configured for 128, 256, and 512 concurrent client threads to produce read-only uniformly-random or Zipf workloads. In a real setting, the QoS controller would build a larger set of tables for better coverage. However the current set of points are sufficient for demonstrating our approach. YCSB is initially setup over a Cassandra cluster of two servers. Progressively, the QoS controller grows the cluster to five (MEDIUM) or seven (SMALL) servers[1].

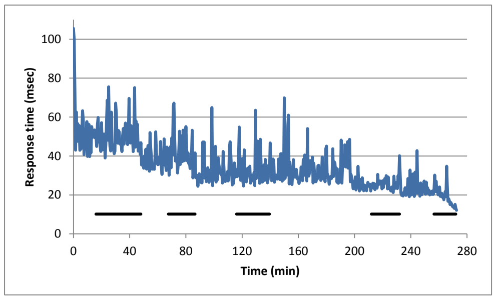
**Zipf distribution.** In the first set of experiments, we configure YCSB to produce a workload of ZIPF-distributed reads to 15 million 1KB records (a 15GB dataset). The QoS controller sets the row cache size to 500MB per Cassandra server on the initial cluster. The server row cache capacity is periodically checked (every 30 secs). The QoS controller maintains the cluster size until all server row caches have filled up and then on for about 10 min. At that point the QoS controller triggers an elasticity action. In this phase, the QoS controller is configured to scale the system continuously as long as there is performance benefit from doing so.

Figures 3 and 4 depict EWMA response time (a) and throughput (b) in the SMALL and MEDIUM clusters respectively with a load of 256 concurrent client threads. The horizontal bars designate periods of data streaming during which a new (bootstrapping) node receives data from offloaded nodes. Tables II and III summarize our results for the SMALL and MEDIUM clusters (in steady state) for 128, 256, and 512 concurrent client threads. As cluster size grows, performance benefits come from increased I/O path parallelism as well as from the larger aggregate cache capacity available (each new server adds 500MB of cache to the cluster). Bootstrapping has a performance hit, but this is typically small due to throttling on streaming throughput applied by Cassandra.

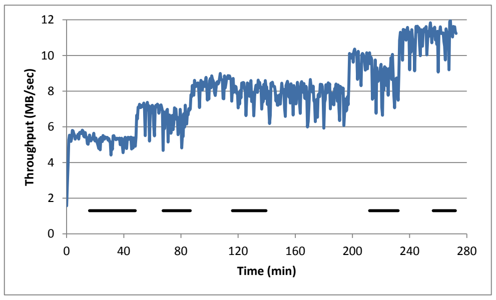
As anticipated, the MEDIUM cluster can achieve a given level of performance with fewer servers compared to SMALL. For example, a response time of 34ms for 512 client threads is achievable with either 7 *m1.small* VMs or with 4 *m1.medium* VMs. Similarly, for the same load and service capacity level the MEDIUM cluster achieves higher throughput (up to 45%) compared to SMALL. Figures 5(a) and 5(b) depict the response time vs. offered-load relationship in the two clusters with growing service capacity. We also observe that results with the SMALL cluster exhibit higher variation compared to the MEDIUM cluster. This can be attributed to the fact that average CPU utilization is lower (and thus more CPU available to absorb spurious activity) in the MEDIUM vs. the SMALL cluster.

**Uniformly random distribution.** Similarly to the case of the Zipf distribution, we configure YCSB to produce uniformly-random reads over the same 15GB dataset with 128, 256, and 512 client threads. Figure 6 depicts performance results for the MEDIUM cluster with 256 client threads (a similar figure for SMALL is omitted due to space constraints). The throughput drop at 83 min is due to a brief freeze of all Cassandra VMs (which we believe is Cloud related). Tables IV and V summarize our results from these experiments in steady state. Similar to the Zipf distribution, we observe that throughput increases and response time decreases with growing cluster size, although less so (up to 15%) due to smaller benefit from caching in this case (18% hit ratio vs. 60% for Zipf). The MEDIUM cluster can achieve a given level of performance with fewer servers compared to SMALL: a response time of about 45ms for 512 client threads is achievable with either

[1]Our goal in sizing the two clusters was to provide comparable performance at their maximum capacity. Since *m1.medium* VMs are more powerful than *m1.small*, five *m1.medium* VMs are sufficient to match and exceed the maximum performance possible with seven *m1.small* VMs.

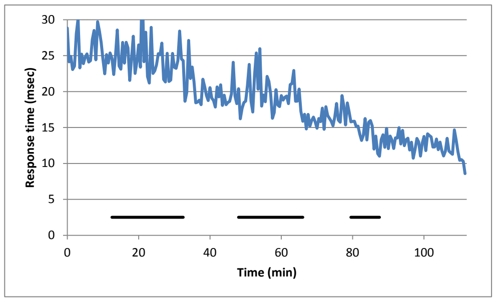


(a) Response time

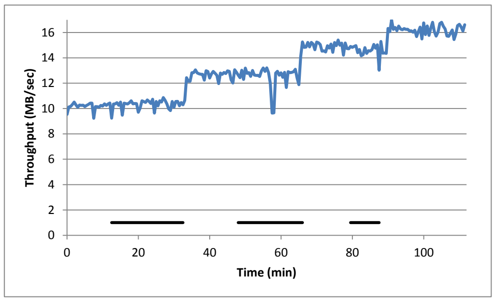


(b) Throughput

Fig. 3: Amazon's M1.SMALL, Zipf distribution, 1 client with 256 threads



(a) Response time



(b) Throughput

Fig. 4: Amazon's M1.MEDIUM, Zipf distribution, 1 client with 256 threads

| | ZIPF-100% READS: AMAZON M1.SMALL | | | | | |
|---|---|---|---|---|---|---|
| # Servers / # Clients | 2 | 3 | 4 | 5 | 6 | 7 |
| 128 | 23.37, 5.55 | 19.7, 6.66 | 15.62, 7.72 | 13.98, 8.81 | 12.18, 10.17 | 10.75, 11.6 |
| 256 | 49.51, 5.47 | 37.92, 6.87 | 32.3, 8.5 | 25, 9.65 | 22.4, 11.1 | 18.01, 11.14 |
| 512 | 102.23, 5.21 | 76.15, 6.59 | 61.01, 8 | 51.45, 9.8 | 44.01, 10.9 | 34.6, 12.2 |

TABLE II: Response time (ms), throughput (MB/sec) for ZIPF access pattern on Amazon's M1.SMALL

| | ZIPF-100% READS: AMAZON M1.MEDIUM | | | |
|---|---|---|---|---|
| # Servers / # Clients | 2 | 3 | 4 | 5 |
| 128 | 13.53, 9.27 | 11.73, 11.85 | 8.93, 14.7 | 5.89, 15.9 |
| 256 | 25.73, 10 | 19.64, 11.67 | 16.57, 14.88 | 12.02, 16.20 |
| 512 | 58.44, 9.33 | 44.63, 11.95 | 34.36, 15 | 25.77, 17.95 |

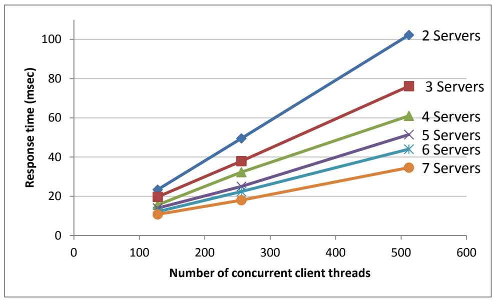
TABLE III: Response time (ms), throughput (MB/sec) for ZIPF access pattern on Amazon's M1.MEDIUM

| | UNIFORM-100% READS: AMAZON M1.SMALL | | | | | |
|---|---|---|---|---|---|---|
| # Servers / # Clients | 2 | 3 | 4 | 5 | 6 | 7 |
| 128 | 25.4, 4.8 | 22.17, 6.14 | 17.61, 7.06 | 15.76, 8.17 | 12.78, 9.55 | 12.24, 10.4 |
| 256 | 51.28, 5.16 | 51.12, 4.88 | 40.94, 6.46 | 33.24, 7.8 | 26.6, 9.4 | 22.7, 10.7 |
| 512 | 116.9, 4.42 | 83.14, 5.58 | 70.27, 7.70 | 54.73, 9.25 | 44.24, 10.6 | 44.46, 10.6 |

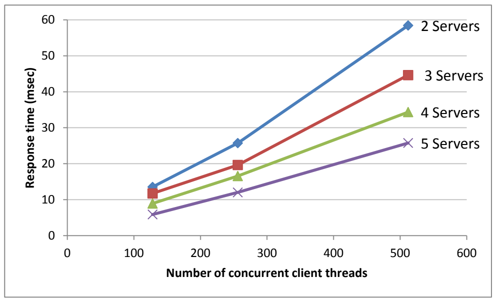
TABLE IV: Response time (ms), throughput (MB/sec) for UNIFORM access pattern on Amazon's M1.SMALL

| | UNIFORM-100% READS: AMAZON M1.MEDIUM | | | |
|---|---|---|---|---|
| # Servers / # Clients | 2 | 3 | 4 | 5 |
| 128 | 15.69, 8.78 | 11.8, 11.18 | 10.16, 13.06 | 7.52, 14.75 |
| 256 | 28.79, 9.05 | 24.64, 11.24 | 19.39, 13.51 | 14.77, 15.22 |
| 512 | 60.5, 8.64 | 48.8, 11.08 | 45.99, 12.77 | 28.44, 14.8 |

TABLE V: Response time (ms), throughput (MB/sec) for UNIFORM access pattern on Amazon's M1.MEDIUM

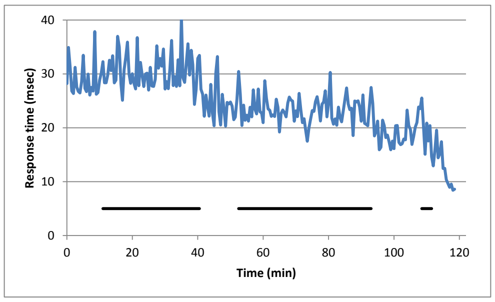


(a) Amazon's M1.SMALL

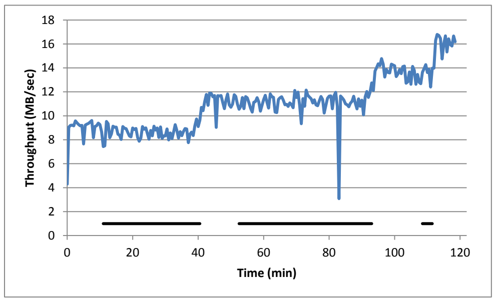


(b) Amazon's M1.MEDIUM

Fig. 5: Response time (ms) vs. offered load for different cluster capacities in the ZIPF workload



(a) Response time



(b) Throughput

Fig. 6: Amazon's M1.MEDIUM, Uniformly random distribution, 1 client with 256 threads

| # Servers / # Clients | ZIPF-100% READS: AMAZON M1.SMALL | | | | | |
|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | 6 | 7 |
| 256 | 49.51, 5.47 | 37.92, 6.87 | 32.3, 8.5 | 25, 9.65 | 22.4, 11.1 | 18.01, 11.14 |
| *384* | *75.87, 5.34* | *57.03, 6.73* | *46.65, 8.25* | *38.22, 9.72* | *33.20, 11.00* | *26.3, 11.67* |
| 512 | 102.23, 5.21 | 76.15, 6.59 | 61.01, 8 | 51.45, 9.8 | 44.01, 10.9 | 34.6, 12.2 |

TABLE VI: The row on 384 clients is a weighted average of the rows on 256 and 512 clients on Amazon's M1.SMALL

7 *m1.small* VMs or with 4 *m1.medium* VMs. Similarly, the MEDIUM cluster achieves a higher level of throughput compared to SMALL for the same load level.

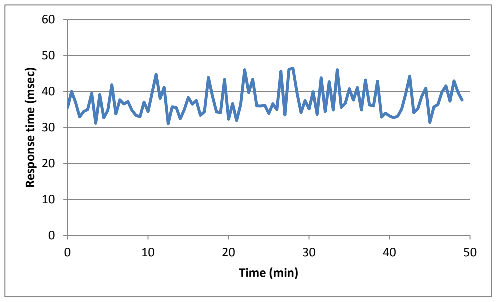
**Validation of the methodology.** At the initial stage of the YCSB benchmark the user sets up an SLA for the CF created and accessed by YCSB. In the SLA the user specifies the dataset size (15GB), degree of locality (ZIPF), the requested maximum average response time for read operations (40ms), an upper limit on throughput (384 threads), and row size (1KB). The QoS controller uses Table II to estimate the capacity to achieve the requested SLA. It uses weighted-average interpolation to produce the new row for 384 threads shown in Table VI. Other approaches to estimation have been explored in the past [11], [13]; a more thorough exploration of such techniques however is outside the scope of this paper.

Using the predictions of Table VI, the QoS controller provisions a 5-node Cassandra cluster of EC2 *m1.small* type VMs, creates a CF on it and periodically monitors the achieved response time and throughput. Figure 7 shows that response time and throughput closely approximate the levels predicted by Table VI. Although average response time is below 40ms, throughput is slightly higher than expected (10.55 vs 9.72 MB/s). Since the user-requested SLA is achieved, the QoS controller does not trigger any further elasticity actions.

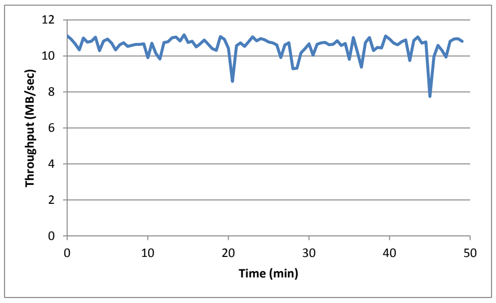
Although (due to space constraints) we have focused on read-only workloads in this paper, we provide some insight to the characteristics of write workloads. Figure 8 depicts YCSB response time and throughput in an identical setup to the one used in Figure 7 (384 threads, 5 servers). We observe that response time is higher (61ms vs. 37.3ms) while throughput is lower (4.9MB/s vs. 10.6MB/s) with both metrics exhibiting more noise compared to the read-only workload. Cassandra's default write policy (unstable writes to commit log and memtable with periodic syncs to disk) largely decouples write performance from the disk device. However interference with frequent memtable/SSTable compaction activity (especially intensive in a 100% write workload) hurts performance due to increased I/O activity as well as increased CPU needs.

## VI. CONCLUSIONS AND FUTURE WORK

In this paper we presented a methodology for QoS-aware provisioning of Cassandra clusters based on application SLAs. Our evaluation demonstrates that the methodology is effective

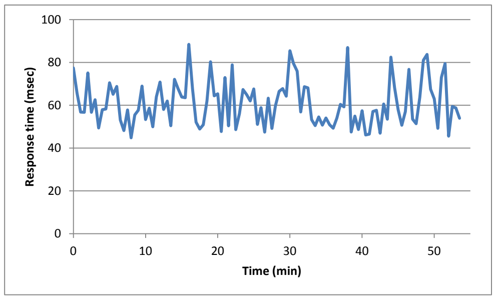


(a) Response time

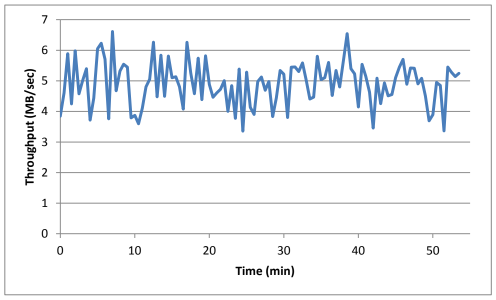


(b) Throughput

Fig. 7: Amazon's M1.SMALL, Zipf read-only distribution, 1 client with 384 threads, 5 servers



(a) Response time



(b) Throughput

Fig. 8: Amazon's M1.SMALL, Zipf write-only distribution, 1 client with 384 threads, 5 servers

in predicting server capacity requirements given simple application workload descriptions. Part of the simplicity of our approach stems from the scalability and elasticity mechanisms built into NoSQL systems such as Cassandra; we believe that our work is more broadly applicable to such systems.

A number of QoS management aspects were omitted from our evaluation due to space constraints. A full exposition of write-intensive workloads and the handling of interference between workloads feature prominently on this list. Based on our experience with write workloads we believe that our methodology can straightforwardly extend to them. Workload interference has been deemed to be an important parameter in the past [7], especially over disk drives. We believe that with the proliferation of flash drive (SSD) technology, the importance of interference at the disk-drive level is less critical today than it was ten years ago. However, contention for other resources (CPU, memory) still needs to be taken into account in provisioning concurrent workloads over Cassandra clusters. We plan to evaluate the above QoS aspects in our future work.

## ACKNOWLEDGMENT

We thankfully acknowledge the support of the PaaSage (FP7-317715) and CoherentPaaS (FP7-611068) EU projects.

## REFERENCES

[1] Amazon Web Services, "DynamoDB," http://aws.amazon.com/dynamodb/, Aug. 2012.

[2] M. Chalkiadaki and K. Magoutis, "Managing Service Performance in NoSQL Distributed Storage Systems," in *Proc. of the 7th Middleware for Next-Generation Internet Computing (MW4NG) Workshop*, Montreal, Canada, Dec. 2012.

[3] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," in *Proc. of 21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, Oct. 2007.

[4] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[5] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," in *Proc. of 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, Big Sky, MT, Oct. 2009.

[6] Apache Software Foundation, "HBase," http://hbase.apache.org/, 2012.

[7] J. Wilkes, "Traveling to Rome: A retrospective on the journey," *Operating Systems Review (OSR)*, vol. 43, no. 1, pp. 10–15, Jan. 2009.

[8] P. Goyal *et al.*, "CacheCOW: QoS for Storage System Caches," in *Proceedings of 11th International Workshop on Quality of Service (IWQoS 03)*, Monterey, CA, Jun. 2003.

[9] K. Magoutis, P. Sarkar, and G. Shah, "OASIS: Self-Tuning Storage for Applications," in *Proc. of 23rd IEEE Conference on Mass Storage Systems and Technologies (MSST)*, College Park, MD, 2006.

[10] P. O'Neil *et al.*, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[11] E. Anderson, "Simple table-based modeling of storage devices," HP Laboratories, Tech. Rep. HPLSSP20014, Jul. 2001.

[12] Westermann *et al.*, "The Performance Cockpit Approach: A Framework For Systematic Performance Evaluations," in *Proc. of 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, Lille, France, Sept. 2010.

[13] D. Westermann *et al.*, "Automated Inference of Goal-Oriented Performance Prediction Functions," in *Proc. of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, Essen, Germany, Sept. 2012.

[14] P. Bianco, G. Lewis, and P. Merson, "Service level agreements in service-oriented architecture environments," Software Engineering Institute, Tech. Rep. CMU/SEI-2008-TN-021, Sept. 2008.

[15] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with YCSB," in *Proc. of the 1st ACM Symposium on Cloud computing (SoCC '10)*, Indianapolis, IN, Jun. 2010.

[16] D. Chambliss *et al.*, "Performance virtualization for large-scale storage systems," in *Proc. of the Symposium on Reliable Distributed Systems (SRDS)*, Florence, Italy, 2003.