Managing service performance in NoSQL distributed storage systems

Maria Chalkiadaki
Institute of Computer Science
Foundation for Research and Technology–Hellas
Heraklion GR-70013, Greece
mhalkiad@ics.forth.gr

Kostas Magoutis
Institute of Computer Science
Foundation for Research and Technology–Hellas
Heraklion GR-70013, Greece
magoutis@ics.forth.gr

ABSTRACT

In this paper we describe the architecture of a quality-of-service (QoS) infrastructure for achieving controlled application performance over NoSQL distributed storage systems. We present an implementation of our architecture as an extension to the Apache Cassandra storage system and provide results from a preliminary evaluation using the Yahoo Cloud Serving Benchmark (YCSB). Along the way we also present details of an ongoing alternative implementation of our QoS infrastructure in the context of the Apache HBase storage system. Our evaluation provides evidence that our QoS infrastructure can achieve the type of controlled performance required by data intensive performance-critical applications.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Databases

Keywords

Distributed storage systems

1. INTRODUCTION

The demand for inexpensive scalability in data-intensive analytics (web search, data warehouse analysis, etc.) has led to the adoption of *NoSQL* systems (contrasted to *SQL* systems or traditional relational databases) implementing simple interfaces to non-relational data representations. Such systems are well integrated with data programming platforms such as Map-Reduce [6], and a number of such platforms are currently implemented in Cloud infrastructures and offered to applications developers as utility services.

In this paper we focus on two open-source NoSQL distributed storage systems: Apache Cassandra and Apache HBase. Cassandra partitions data between nodes using a consistent hashing function and stores data in each node using a write-once Log Structured Merge (LSM)-tree based scheme [9]. 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. In this paper we focus on a variant of HBase (developed in-house) where we remove HDFS as a common shared store and instead provide each Region server with a local B+-tree indexed storage repository. None of these systems offer support for managed service performance to achieve application-specific targets, which is the goal of the architecture described in this paper.

Our architecture continuously monitors application performance and controls resource allocations in order to achieve application goals via two key mechanisms: *(i)* storage elasticity; and *(ii)* reserved cache allocations. We describe Cassandra and HBase prototypes that use storage elasticity to exploit I/O path parallelism and evaluate our Cassandra prototype in such a scenario. Our Cassandra prototype further supports reserved cached allocations allowing us to evaluate the tradeoffs involved in the use of both mechanisms to achieve user requirements.

The key contribution of our work is the specification of a QoS architecture that relies on mechanisms that are common across NoSQL systems. While the philosophy of the QoS architecture is not new, its focus on NoSQL technologies provides insight into a growing area of distributed storage systems in line with technology trends. In the remainder of the paper we first present background and related work in Section 2. In Section 3 we describe the design of our QoS infrastructure and in Section 4 our current implementations. In Section 5 we present an evaluation of our Cassandra prototype using the YCSB benchmark. Finally, in Section 6 we present our conclusions.

2. 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 [7], Bigtable [4], and their open-source variants Cassandra [9] and HBase [2]. 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 are beginning to offer versions of their data-centric services that support controlled performance, reliabilility, etc. Most recently (sum-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MW4NG '12 December 3-7, 2012, Montreal, Canada
Copyright 2012 ACM 978-1-4503-1607-1/12/12 ...$15.00.

mer 2012) 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 [12]) 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 caches typically used in NoSQL systems. More recently, Magoutis et al. [10] 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 [7] 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 [2] and Cassandra [9]. 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 [11]. Our HBase prototype used in this paper retains the data partititioning mechanisms of HBase but adopts a per-Region server storage subsystem (based on the Oracle Berkeley DB Java Edition) that is reminiscent of that used in Amazon's Dynamo [7].

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 [9]. Similarly, each row maps to a position on the ring by hashing its key using the same hash function. Each node is in charge of storing 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.

3. DESIGN

The goal of our QoS infrastructure is to estimate the amount of resources needed to dedicate to an application (initially, as well as dynamically over time) based on a simple description of its behavior and intended goals. Applications must provide the following information about their behavior, on a per column-family basis (also called their *CF profile*): (i) Data set size and degree of locality (a coarse characterization such as RANDOM, ZIPF-LIKE, etc.); (ii) average row size. These attributes are important for caching purposes (small datasets, strong locality, and/or small rows increase cache efficiency) and to estimate the throughput expected from the underlying devices (large rows –e.g., >1MB– result in more sequential accesses, which can be efficiently executed in standard hard-disk drives.

Applications can set a response-time target and an upper limit on offered load on the CF. If the target is not set a-priori, the system implicitly promises to keep service-side latency within tolerable limits (e.g., less than 500ms) for the offered load. The system will set aside resources (over time) to satisfy the offered load at less than the targeted response time. The key control parameters used by our system are the total amount of cache to use for the specific application profile, and the number of independent I/O paths to data servers.

Just as any adaptive QoS system, we utilize a control loop comprising the following phases: (a) monitoring, by measuring put/get operation response time and throughput; (b) comparing observed application metrics with targets agreed

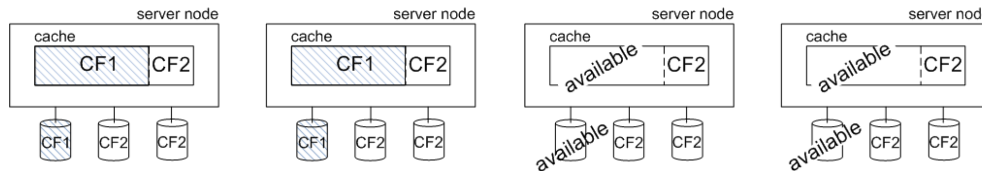

Figure 1: Example of different per-CF resource mapping strategies: Column families CF1 and CF2 target comparable levels of throughput. CF1 exhibits access locality and a small working set whereas CF2 does not. The system continuously adapts cache sizes and number of server nodes to achieve the desired performance.

upon via SLAs; and (c) analyzing how to adapt if we fall below or go above the targets. The QoS system is designed to trade off read paths for cache memory depending on which is the most efficient way to achieve its objectives (assuming reliability objectives –e.g., number of replicas used– have already been satisfied). For example, applications accessing a large CF with little locality or using large rows are not expected to benefit from a significant row cache. On the other hand applications accessing a CF over a small working set will likely benefit most from caching and will not require many independent I/O paths for the same level of throughput. An example of this principle is depicted in Figure 1.

Assuming no caching and a random workload with no locality, our QoS system initially attempts to satisfy a given load (at a reasonable response time) by assigning a CF to the right number of nodes. For example, a workload throwing 400 4KB (mostly random) reads per second to a Cassandra cluster will require –in the worst case– the CF be spread over four dedicated nodes (see section on I/O costs below for a justification of these estimates). However, a workload exhibiting strong locality in its accesses to a CF could be accomodated into fewer nodes with the assumption that cache hit rates will compensate for the reduced degree of parallelism in the I/O paths.

Based on observations and construction of the application profile over time, the QoS system decides on the best actions to take in order to increase (or decrease) available throughput: either adjust the CF cache (if the cache miss ratio is high and the workload has enough locality to take advantage of more cache) or scale data set to more machines (if the workload has no locality). The decision takes into account the overall availability of resources. For example increasing data paths may be the only option if the system is running low on memory that can be allocated for caching.

The system will throttle applications that exceed their peak load to guarantee that they are getting the level of performance requested and do not negatively interfere with other applications that may be sharing resources with them. Throttling of client traffic has also been used elsewhere (for example see [3], [10]) as an effective mechanism for short-term control of resources in a storage-area network.

Data repartitioning. The ability to change the mapping of data to storage servers on-demand and while the system is operating is a key functionality of re-configurable storage systems and underlies the elasticity properties of Cloud storage systems. In this section we describe the data repartitioning algorithm in Cassandra and a new data repartitioning algorithm we have introduced into a home-grown version of the HBase system.

Cassandra partitions the key space onto system nodes using consistent caching. When a new node enters the system (e.g., Node 7 in Figure 2) data movement takes place only between the neighbors of the new node. For example in the setup of Figure 2 (where replication factor is equal to two), Node 7 will receive data from Nodes 1 and 2 and Node 2 will drop all data from the key range (between Nodes 6 and 1) no longer served by it.

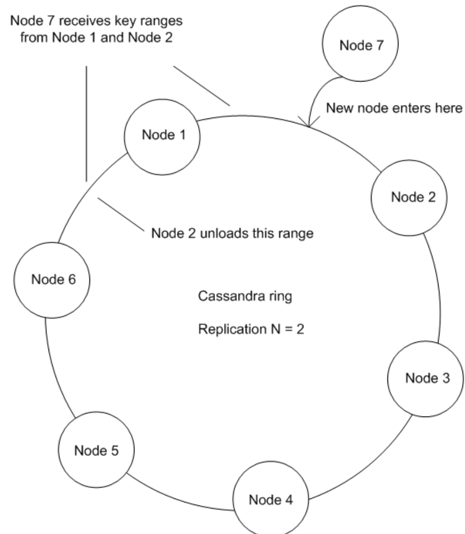

Figure 2: Data repartitioning in Cassandra.

HBase follows a different partitioning scheme where keys are mapped to Regions and Regions are mapped to Region Servers (equivalent to Cassandra nodes). Both mappings are maintained by the HBase Master, a centralized server that implements metadata interfaces to create, modify, and remove tables and to assign, move, and unassign Regions to servers. The Master operates a load-balancer component that will periodically move regions around to balance cluster load. Region splits are decided and carried out by Region servers independently based on a region-size threshold.

Costs of performing I/O. To be able to estimate performance on CF accesses, we take into account I/O path functionality, which we briefly describe below in the case of Cassandra (HBase utilizes similar concepts and therefore we omit the full details for it). The Cassandra read path starts at the client. A client can send operations to any node in the cluster, who then becomes the coordinator for the operation. The coordinator contacts a configurable number of replicas to perform the read or write operation.

A read first looks up a *row cache*, then a *key cache*, and

then (if it misses in both caches) it tries to locate the key/value pair in the node's underlying storage system. In the worst case, a number of SSTables will have to be brought in memory to find the requested key. The cost of this path involves a number of network hops (depending on whether the coordinator is also one of the replicas serving the key sought) and disk accesses (none if we have a hit in the row cache), one or more if we either hit in the key cache or have to bring in indices from SSTables on disk. The use of compactions and Bloom filters narrows down the choice among SSTables, reducing I/O operations. Disk accesses in Cassandra go to a local file system.

Write operations first record the update in a stable commit log (synchronously, or by explicit user choice, asynchronously) and then append it to a buffer (a *memtable*). When memtables reach a certain size (or at regular intervals) they are written to ordered SSTable files on disk. Write performance is normally unaffected by SSTable creation activity, unless write traffic exceeds the ability of a Cassandra node to buffer while writing to disk. Cassandra performs a number of background operations that may at times affect a node's response time, namely compactions that merge SSTables into fewer and larger files. Taking write intensity of a workload into account, one has to factor in (amortize) the periodic costs of compaction into the average cost of writes.

Finally, I/O costs depend significantly on the type of stable store –disk or SSD– used. Standard disks can perform about 100 small (<4KB) random IOPS whereas SSDs can perform several thousand random IOPS. Use of SSDs (as in DynamoDB) can significantly lower the variance in the I/O cost model and lead to more predictable behavior.

4. IMPLEMENTATION

We describe our implementation of the basic infrastructure (monitoring, elasticity, and caching control) that we developed for run-time adaptation to user-specified performance goals. Figure 3 depicts our Cassandra implementation, with solid boxes denoting existing components and dotted boxes denoting our extensions.

Monitoring. Our monitoring infrastructure continuously samples response times and I/O throughput of get and put operations and periodically computes exponentially weighted moving averages (EWMAs). The primary client of these metrics is the QoS component, which uses the estimates to take control actions (regulate cache assigned to a CF, or increase/decrease I/O path parallelism).

The EWMA of successive response times for read operations is calculated based on the following formula, where r*(T)* is the response time sampled at time T and α=0.125.

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

The Thrift/Cassandra component depicted in Figure 3 measures individual operations by timestamping request and response messages. Cassandra clients compute EWMAs for response time and throughput and make them available to the QoS controller through a simple interface. This functionality is also built in and supported by the HBase client.

Elasticity. We use default support for elasticity in Cassandra, namely we increase cluster capacity for an application's CF by bootstraping a new node into the targeted ring (either at an explicit token or by automatically choosing a token that gives it half the tokens from the most space-constrained node) and use the `move` API to load-balance the ring. We delete moved keys from offloaded nodes lazily using the `cleanup` API.

We implemented elasticity in our version of HBase by (i) splitting a Region (that maps to a Berkeley DB database) into two (approximate) halves; (ii) serializing and shipping a Region (Berkeley DB database) to a remote node; and (iii) starting a Region at a remote node. The data transfer is performed in the background prior to the Region being moved, limiting the amount of time writes are blocked.

Caching. To control cache allocations in Cassandra we use the `setCapacity` method of the row/key-cache JMX MBeans exported by storage servers, determining how many keys and rows to cache per CF on each node holding replicas of those keys. Each key-cache hit saves one seek and each row-cache hit saves two (or more) seeks. Since keys tend to be much smaller than entire rows, the efficiency of the key cache is expected to be higher than that of the row cache.

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 version solves this problem by using off-JVM heap memory as described later in this section.

In contrast to Cassandra, HBase offers coarse-grain control of its cache memory (termed the *block cache*), namely one can control only the total amount of memory allocated to HFile blocks (including index blocks). Therefore HBase does not readily support per-CF control of cache memory. Extending HBase to support per-CF read caching is a goal of ongoing work.

QoS controller. The key functionalities of the QoS controller are to *(i)* setup SLAs with application clients, requesting their CF profiles (Section 3) and performance requirements (we are currently focusing on satisfying response-time targets at certain throughput rates); *(ii)* effect initial resource allocations based on estimates for cost of I/O to be performed by the application; *(iii)* periodically (once a minute) collect monitored response-time and throughput metrics from Cassandra clients and plan 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. Since an application comprises several Cassandra clients and each client monitors performance independently, the QoS controller must combine (currently averages) the reported metrics for the entire application.

Our 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. Our QoS controller is able to draw certain statistics (such as cache hit rates, cache capacity, etc.) at runtime through Cassandra's JMX interface. The controller has also the

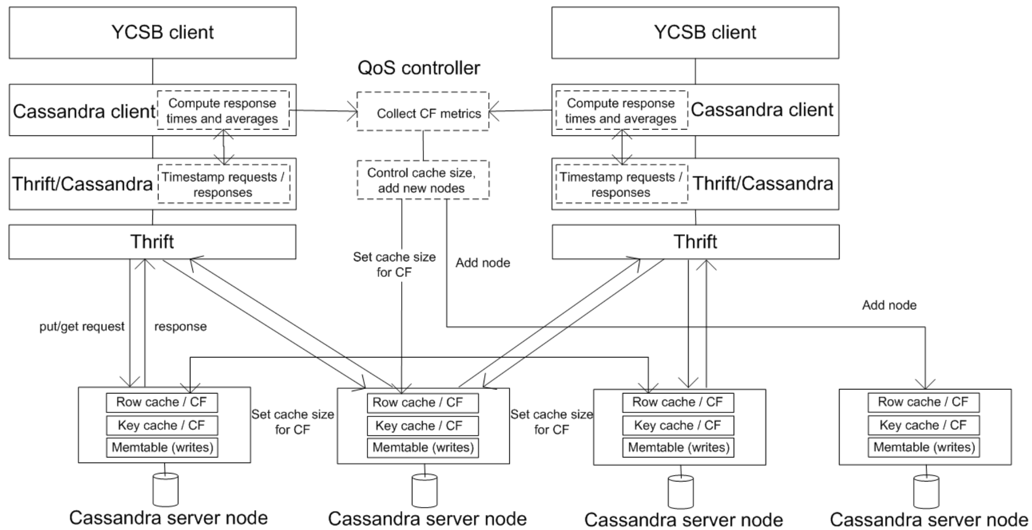

Figure 3: The Cassandra QoS monitoring and control system.

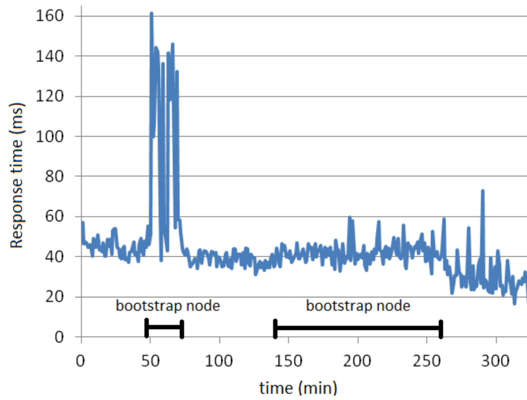

Figure 4: Zipf-distributed reads.

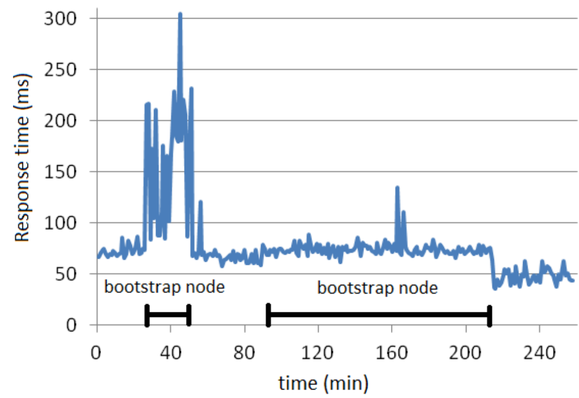

Figure 5: Uniformly random reads.

ability to regulate certain background activities (such as frequency and scope of compactions) via ColumnFamily-StoreMBean methods such as `setMinimumCompactionThreshold`. Such controls are critical in order to avoid the unpredictable impact of background activities on quality of service. A full investigation of these effects and mechanisms is the focus of future work.

5. EVALUATION

Our experimental setup consists of four servers with dual-core Intel Core2-Duo CPUs at 3.16GHz equipped with 3GB of DRAM and a single direct-attached SATA disk capable of about 100 small (1KB) random IOPS. The server operating system is Linux Ubuntu 10.4.1 LTS. Cassandra is version 1.0.10 running over the OpenJDK 1.6.0-24 Java runtime environment with a heap size of 1GB in each server. Our evaluation workload is the Yahoo Cloud Serving Benchmark (YCSB) [5] version 0.1.4, configured to use 8 concurrent load-generating threads. In our experiments one server is dedicated to executing the YCSB workload while the remaining three are dedicated to the Cassandra service. We have used the Random Partitioner (the default partitioning strategy using consistent hashing) for mapping rows to Cassandra servers and set the replication factor to two in a read-any, write-both setting.

To exhibit the dynamic decisions taken by the QoS controller we present results with 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; or (b) a uniformly-random probability distribution. According to the Zipf distribution, some records are extremely popular while most records are unpopular. We have disabled the key cache to focus on the characteristics of the row cache alone.

Zipf distribution. In the first set of experiments, we configure YCSB to produce a workload of Zipf-distributed reads to 10 million 1KB records (a 12GB dataset). At the initial stage of the YCSB benchmark the user sets up an SLA for the single CF created and accessed by YCSB. In the SLA the user specifies the dataset size (12GB), degree of locality (ZIPF), the requested maximum response time for read operations (30ms), an upper limit on throughput (1000 rows/sec), and row size (1KB). The QoS controller creates

a CF on a single Cassandra server and periodically (once a minute) monitors the achieved response time.

Figure 4 depicts results from a typical run of the system, which is initially above the response-time target. The QoS controller increases the cache size for the CF from 0 to 200MB in steps of 50MB. Detecting that the benefits from increasing cache size diminish, it decides (at 47′) to spread the CF over two nodes. Bootstrapping the second node (which takes over 50% of the ring) lasts 26′ transferring 12GB (our network has a peak speed of 10MB/s). During that time response time increases since the first node is busy preparing and streaming SSTables to the new node while the latter is unavailable for reads during that process. After the elasticity operation is complete, each node's cache is halved to 100MB to conserve the total cache size of 200MB. The controller further increases each node's cache size to 300MB in increments of 50MB (to a total of 600MB) achieving an average response time of about 38ms with no measurable gain from further cache increases. It then decides (at 137′) to increase the number of nodes to three (reducing each node's cache to 200MB). Bootstrapping the third node (which offloads 25% of the ring from the second node) lasts 125′ with small performance hit due to throttling applied by Cassandra. This brings response time within user-specified levels.

Uniformly random distribution. In the second set of experiments we configure YCSB to produce a workload of uniformly-random reads over the same 12GB dataset. The user sets up an SLA similar to the previous with the difference that the degree of locality is now RANDOM and the requested maximum response time is 50ms. The QoS controller creates a CF on a single Cassandra server and periodically (once a minute) monitors response time.

Figure 5 depicts results from a typical run of the system. Increasing cache size from 0 to 100MB yields low cache hit rates, prompting the controller to quickly (at 27′) increase the number of nodes to two while reducing each node's cache to 50MB. Bootstrapping lasts 29′ with significant impact on performance. The controller then increases each node's cache to 150MB (for a total of 300MB), stopping there as it sees no measurable benefit from caching. It then decides (at 90′) to further scale the system to three nodes, a bootstrapping process that (similar to the previous experiment) lasts 122′ with little performance impact, eventually bringing response time within the user-specified target.

Note that we originated the system at a single node although knowledge about the workload provided in the SLA would suggest a better starting point is to spread the dataset initially on more than one nodes. Our choice aimed to illustrate that the QoS controller can still navigate the system towards user-specified goals even when starting from a sub-optimal point, at the expense of more elasticity actions. While spreading a dataset initially over many nodes may seem to always be a good idea, a downside is that it increases interference between applications at storage nodes.

6. CONCLUSIONS

In this paper we describe a QoS infrastructure geared towards scalable NoSQL storage systems and current implementations of the infrastructure within the Apache Cassandra and HBase systems. Our evaluation of the Cassandra-based implementation under YCSB workloads highlights control of server-side caching as an effective solution to regulating application response time when the application exhibits strong data-access locality. Control over I/O path parallelism via elasticity mechanisms is a complementary and effective solution for matching user performance requirements. The impact of elasticity actions on performance varies depending on their intensity and the hardware characteristics of the underlying platform, warranting further investigation. Our work in this paper highlights the viability of the basic mechanisms underlying our QoS architecture. Future work will focus on the dynamics under variable, competing, and write-intensive workloads.

7. ACKNOWLEDGMENTS

We thankfully acknowledge the support of the CumuloNimbo (FP7-257993) and PaaSage (FP7-317715) EU projects.

8. REFERENCES

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

[2] Apache Software Foundation. HBase. http://hbase.apache.org/, August 2012.

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

[4] F. Chang et al. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[5] B. F. Cooper et al. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing (SoCC '10)*, Indianapolis, IN, June 2010.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

[8] P. Goyal, D. Jadav, D. S. Modha, and R. Tewari. CacheCOW: QoS for Storage System Caches. In *Proceedings of 11th International Workshop on Quality of Service (IWQoS 03)*, Monterey, CA, June 2003.

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

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

[11] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[12] J. Wilkes. Traveling to Rome: A retrospective on the journey. *Operating Systems Review (OSR)*, 43(1):10–15, January 2009.