Incremental elasticity for NoSQL data stores

Antonis Papaioannou*† and Kostas Magoutis*‡
*Institute of Computer Science, Foundation for Research and Technology – Hellas, Heraklion 70013, Greece
†Computer Science Department, University of Crete, Heraklion 70013, Greece
‡Department of Computer Science and Engineering, University of Ioannina, Ioannina 45110, Greece

***Abstract*—Service *elasticity*, the ability to rapidly expand or shrink service processing capacity on demand, has become a first-class property in the domain of infrastructure services. Scalable NoSQL data stores are the de-facto choice of applications aiming for scalable, highly available data persistence. The elasticity of such data stores is still challenging, due to the complexity and performance impact of moving large amounts of data over the network to take advantage of new resources (servers). In this paper we propose *incremental elasticity*, a new mechanism that progressively increases processing capacity in a fine-grain manner during an elasticity action by making sub-sections of the transferred data available for access on the new server, prior to completing the full transfer. In addition, by scheduling data transfers during an elasticity action consecutively between each of the pre-existing servers and the new server, rather than as simultaneous transfers, incremental elasticity leads to smoother elasticity actions, reducing their overall impact on performance.**

I. Introduction

Operators of successful Internet-scale applications invariably see the growing popularity of their services translate into the need for more data capacity, higher I/O performance, and continuous operation. Data demands have been growing exponentially in recent years and distributed data stores have been gaining popularity as a means to addressing these needs. Rather than relying on centralized storage arrays, distributed data stores consolidate large numbers of commodity servers into a single storage pool, providing large capacity and high performance at low cost over an unreliable and dynamically-changing (often cloud-based) infrastructure.

The emergence of Web 2.0, social networking, and the Internet of Things increased significantly the sources of new information (applications, people, devices) and resulted in a huge need for the storage of data. IoT alone is expected to generate a staggering 400 zettabytes (ZB, each about a billion TB) of data each year by 2018 [1]. Although part of the data may be stored temporarily for subsequent analysis, storage space and throughput requirements rise dramatically during time periods where large amounts of new data is created. Data-intensive applications therefore require data engines that are *elastic* (can adapt resources to requirements dynamically), accommodating strong data growth and allowing new nodes to join (or leave) the system gracefully, with minimal performance and availability loss, at any given time.

In addition to growing data capacity needs, applications (especially those developed by modern, highly responsive Web services) demand high performance from the underlying data stores. These requirements are usually described in terms of throughput (e.g. operations per second) and/or latency (time required for each request to be served). A number of large-scale data stores in the marketplace today used (often as a service) by Internet enterprises are designed to offer performance guarantees expressed as *service-level objectives*. A premier example is DynamoDB [2], a proprietary scalable key-value store service offered by Amazon. While DynamoDB has been successful with applications that require fast and predictable performance, it will not scale automatically if the workload requirements change. Users should explicitly request more throughput. However the effect of such a change request is not instant. Amazon states that the "*increment of throughput will typically take anywhere from a few minutes to a few hours*"[1]. This means that throughput spikes cannot be rapidly remedied. As a result, applications using DynamoDB may experience periods where a portion of their requests will be dropped or blocked until the system is reconfigured. This leads application developers to opt for ad-hoc solutions [3], [4] which may violate application semantics (e.g. violate durability semantics when using intermediate buffers/queues in order to handle request throttling by DynamoDB). A likely cause of DynamoDB's inability to rapidly support a changing service-level objective is that rapid adaptation would have an adverse impact on existing guarantees to applications.

There is usually a trade-off between the duration of the adaptation actions and their performance impact during adaptation. Ideally, the system should be able to adapt to data growth or workload changes as quickly as possible to satisfy application requirements. Nevertheless, the more aggressive the adaptation action the deeper its impact on application performance. In this paper we focus on the performance impact of data elasticity actions in NoSQL data stores. In particular we target data stores that horizontally partition data (referring to data partitions as *shards*) and spread them as far as possible on the available nodes for load balancing.

Several popular data stores [5], [6], [7], [8] require a new (joining) node to take responsibility for sections of data spread across the existing nodes to achieve a more efficient load balancing, receiving the corresponding data via network transfers. One common option to carry out these network transfers [5], [7], [8] is to perform them in parallel towards the new node, raising a number of challenges: First, a large number of nodes are simultaneously reducing their processing capacity while engaged in data transfer, resulting in an overall performance

[1]https://aws.amazon.com/dynamodb/faqs/ (retrieved April 2017)

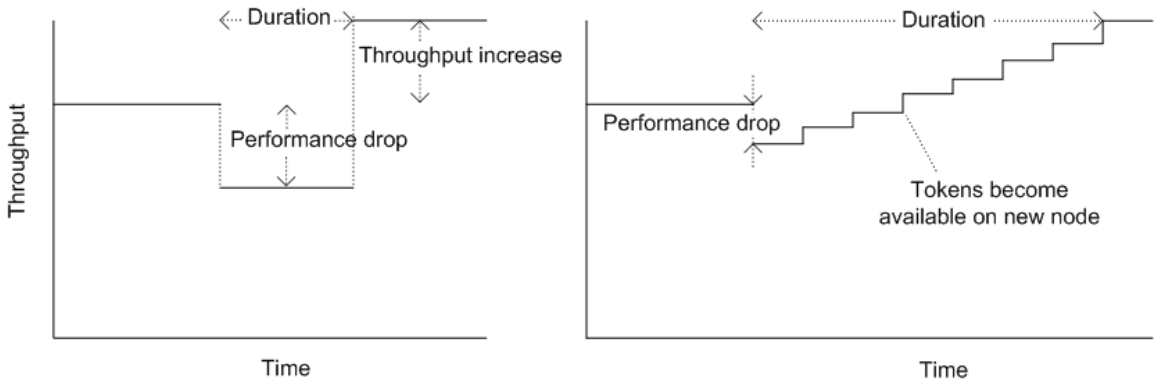


Fig. 1: Performance characteristics of parallel network transfers (left) vs. incremental elasticity (right)

impact. Second, the new (joining) node is solely engaged in data transfer (at the speed of its network link) and cannot contribute processing capacity until the time all data transfers are over. Third, with all data transfers completing at about the same time, associated activities such as data compactions are likely to also overlap, resulting in significant I/O activity at the new node during the early stages of its normal operation [9]. Fourth, the many-to-all communication pattern in this phase is known to under certain circumstances be a cause of throughput collapse in large-scale data centers [10]. Our *incremental elasticity* technique addresses all four challenges. Our evaluation in this paper focuses primarily on the first two.

Incremental elasticity replaces parallel network transfers with a sequential communication schedule where senders take turns sending data to the new node. As soon as a transfer is over, the associated data are becoming available for access on the new node, while a subsequent transfer of data takes place. The expected performance benefits of incremental elasticity (visualized in Figure 1) are summarized as follows: Fewer nodes are involved in network transfer at any time, reducing the overall performance drop during elasticity. With data becoming available on the new node as soon as data transfers complete, processing capacity increases in a stepwise incremental fashion, providing early benefits of the newly available capacity while the elasticity action is still on. With only a single sending server active at a time, the delay due to its higher load may be masked by other replicas of the data it owns.

In this paper[2], we demonstrate the benefits of incremental elasticity using Cassandra [5], a popular open-source key-value store inspired of Dynamo [6]. Our contributions are:

- A new mechanism for gradually increasing the processing capacity of scalable data stores called incremental elasticity
- An implementation of incremental elasticity in the context of the Cassandra column-oriented key-value store
- An experimental evaluation of the benefits of incremental elasticity vs. simultaneous parallel network transfers under Yahoo! Cloud Serving Benchmark (YCSB) workloads

[2]A preliminary two-page version and a poster appears in ICDCS'17 [11]

The remainder of this paper is structured as follows: Section II provides background on elasticity mechanisms in data stores, including specifics on Cassandra. Section III describes our design and implementation of incremental elasticity. Section IV describes the performance results and comparison of incremental elasticity vs. parallel transfers in Cassandra under different YCSB workloads. Section V discusses related work and finally we summarize our conclusions in Section VI.

II. Background

Elasticity is the ability of a system to dynamically (in an online fashion) adjust its processing capacity by adding or removing resources to meet workload changes. Changes to system capacity while a workload runs typically impact its performance. Ideally, data store elasticity should result in an adjustment of processing capacity proportional to the resources being added or removed, completes at the shortest possible time, and impacts running workloads as little as possible. To achieve a proportional increase in overall performance, the data migrated should be chosen appropriately so as the elasticity action results in a well-balanced system. Elasticity may either expand capacity in a *horizontal* manner, adding or removing nodes from a cluster, or in a *vertical* manner increasing or decreasing the amount of resources (CPU, memory, I/O) of existing nodes of a cluster. In this paper we focus on horizontal elasticity in data stores whose nodes privately own and manage their storage resources (local or network disks or SSDs). Thus, a new (joining) server must receive the data it manages to its own storage upon joining the cluster, via network transfers from other data-store nodes. In such a model, elasticity actions apply to both compute and storage resources.

Several popular data stores [5], [6], [7], [8] achieve data partitioning using consistent hashing [12] for mapping keys to nodes. The specific variation of consistent hashing used in this paper (implemented by the Cassandra [5] column-oriented data store) is to associate each physical node with a number of non-contiguous key ranges, *tokens*, hashed to a ring (Figure 2 left). Each token identifies a *key range*, starting from the previous token (in counterclockwise order) up to it. Tokens are also referred to as "virtual" nodes or *vnodes*. Keys are mapped first to tokens and then to nodes. This allows a finer granularity in

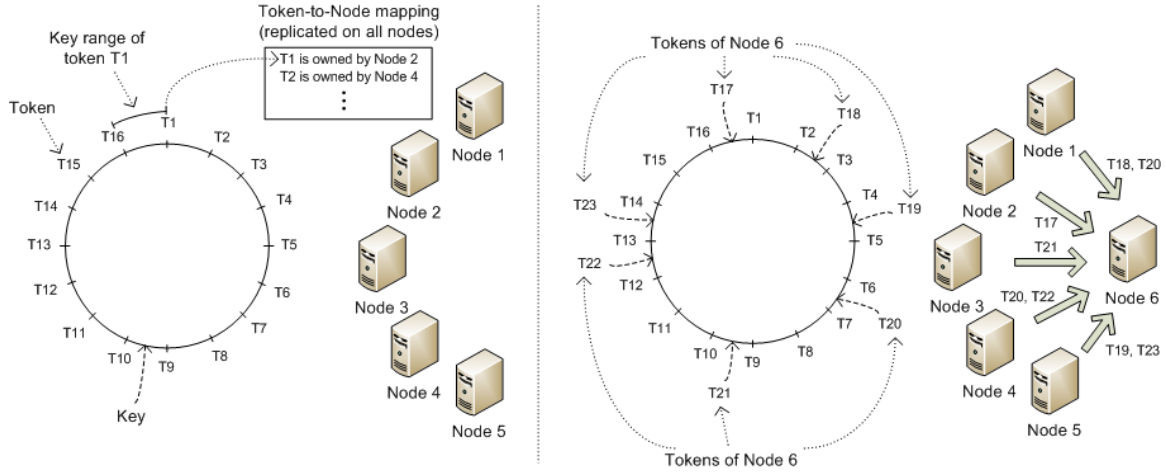


Fig. 2: Creating shards through tokens (left); cluster expansion via parallel network transfers (right)

data distribution among servers [13]. Cluster expansion via the addition of a new node introduces a number of new tokens to the ring (e.g. tokens of Node 6 in Figure 2 right). The new node will take responsibility for key ranges identified by its tokens, and will receive the corresponding data via network transfers from previous data owners (Figure 2 right).

To durably store data, Cassandra implements a version of Log-Structured Merge (LSM) trees [14]. Cassandra nodes apply each incoming update to a write-ahead log for durability and then to an ordered memory buffer (the memtable), which is periodically flushed to an indexed ordered file (the SSTable). A number of SSTables may be consulted to retrieve a requested key/value. Periodic compactions (merge sort runs) of SSTables aim to maintain a small number of large SSTables, improving read performance.

Elasticity via parallel data transfers. The Cassandra column-oriented data store performs elasticity actions by transferring data to a joining node via parallel network transfers. The associated configuration (state) changes are communicated via a gossip protocol [5]. At startup, a new (joining) node uses the gossip protocol to announce itself to the cluster and to receive information about the node topology as well as the tokens that each existing node owns in the cluster. The startup phase of a joining node is also referred to as *bootstraping*. An important task during this phase is to select the tokens for which the new node will take responsibility, and to initiate the data transfer corresponding to these tokens from existing cluster nodes.

Cassandra uses a *streaming* protocol to handle the data exchange among nodes in the cluster. Bootstrap of a joining node involves several stream sessions, each involving the joining node (*receiver*) and one of the existing nodes in the cluster (*follower*), transferring data between each pair of nodes in four steps: (1) The receiver initiates a stream session with a follower; (2) After the initial handshake, the receiver sends a list of tokens (key ranges) it needs from the follower. Upon receiving the preparation message, the follower records which sections of SSTable files it has to send and enters the streaming phase; (3) During the streaming phase the follower sends the data to the new node; (4) Finally after all the data transfer tasks complete, the nodes agree to close the session.

During bootstrap, the joining node does not serve client requests. To maintain availability during elasticity actions, Cassandra does not suspend updates on tokens (key ranges) involved in streaming and about to change ownership. To ensure that the joining node will receive such updates, Cassandra operates as follows: Prior to initiating streaming, the bootstrapping node announces the tokens that it will be responsible for. Existing nodes in the cluster mark these tokens as *pending* and forward a copy of any write request on the pending tokens to the joining node (in the following we refer to these writes as *shadow writes*). In this way, when the new node finally joins the cluster, it will already have received any updates accepted by the system after streaming started, avoiding consistency gaps that may lead to repair [15] operations in the future. When all streaming sessions complete and all data are available to the new node, it enters *normal* state and announces (through gossip) its availability to serve client requests. Note that token transfers typically materialize as many small SSTables in the new node, triggering several compactions to consolidate them.

The aggregate ability of all sending nodes to transfer data may exceed the available network throughput at the receiving side. TCP will thus appropriately throttle each sender. Additionally, Cassandra offers a configuration option (*stream_throughput_outbound_megabits_per_sec*) to limit each sender's streaming throughput. This addresses an empirical observation (e.g.[16]) that high streaming throughput often leads to significant performance variability. Throttling sources reduces the impact of streaming on performance, however it would also increase the duration of elasticity without any incremental benefit in the meantime, as all tokens are made available only at the end of the streaming sessions.

III. Design and implementation

Design. At its core, incremental elasticity involves a commu-

nication mechanism that transfers data to a joining node via a sequential schedule where senders take turns sending data to the new node. As soon as a transfer is over, the associated data are becoming available for access on the new node, while a subsequent transfer of data is in progress. To enable incremental elasticity, senders should either cooperatively decide the schedule of data transfers or assign the task of coordinating those transfers to the joining node, aiming for sequential transfers between a pair of nodes (existing node, new node), allowing only one active such session at a time. Assuming the simpler solution that the joining node is responsible for coordinating data transfers in a pair-wise manner, the choice of which node to stream from can be simple, such as using a round-robin policy or random. Other policies however, such as starting from nodes that are better prepared for the transfer, are possible. The data transfer protocol used should support pair-wise transfers and orchestration by the new (receiving) node, as can (with appropriate modifications) the streaming protocol described in Section II.

At startup the new (joining) node should build a current view of the system by contacting its membership service and receiving information about the node topology and data distribution in the cluster (Figure 2 left). At this point it must be decided which parts of the data the joining node will eventually serve. A key design point for incremental elasticity is to ensure announcements of data ownership (e.g., that a certain region of data is now served by the new node) are now performed incrementally and in a piecemeal fashion. This affects both read and write operations: As soon as a section of data is now owned by the joining node, reads are now directed to it, and writes need no longer be performed twice (cf. *shadow writes* described in Section II).

Under parallel data transfers, a joining node is fully dedicated to processing stream sessions as fast as its CPU resources are able to cope with network traffic (ideally at link speed). As the aggregate network bandwidth of all senders may far exceed the network bandwidth of the receiver, senders are expected to be throttled by TCP, if not by data-store-level streaming limits. Replacing parallel stream sessions by sequential ones should not necessarily increase the overall duration of the elasticity action, if data transfer still happens at the speed of the receiver's network link. In incremental elasticity, the same streaming throughput could be achievable even with a single sender, if the sender is able to send as fast as its network link allows it. While we initially aimed to achieve the same duration of elasticity actions between parallel and incremental elasticity, in our implementation we found that senders cannot achieve full network speed, even with the addition of multithreading techniques. However, reducing the speed at which the joining node processes incoming traffic means that the node has more CPU resources to devote to processing requests after the first batch of tokens become available. Any delay in completing the elasticity action is thus compensated by additional processing capacity available during the action. Even if we are able to (through a more aggressive implementation) achieve full link speed at the receiver, acquiring additional processing capacity there can be achieved by dynamically changing the allocation of memory and virtual CPU resources at runtime through most hypervisors, for the duration of elasticity.

We consider the above design principles as straightforward and believe that they can easily be supported by many distributed data stores. In what follows we describe our implementation in the Cassandra column-oriented data store.

Implementation. Our implementation extends Apache Cassandra version 3.7, especially its gossip and streaming components. Each Cassandra node has a *StreamManager* module responsible for all streaming operations. When a new node bootstraps it creates a *StreamPlan* object to associate token requests with the existing nodes. Internally it builds *StreamSession*s to handle network communication between nodes. The *StreamPlan* is associated with a *StreamCoordinator* component that manages the *StreamSession*s. *StreamCoordinator* initiates the stream sessions sequentially ensuring that only one is active at a time. Generally speaking, we could schedule transfers between arbitrarily-sized subgroups of nodes and the joining node but we consider only subgroups of size one in this paper. As soon as a stream session is over, it selects the next session to start from the inactive stream sessions queue.

With incremental streaming we expect that the aggregate processing capacity of the cluster should increase each time a batch of tokens are made available to the new node. At startup when the joining node announces itself to the cluster, it enters the *joining* state. In this state, it does not receive any client requests (either directly from clients or re-directed from other nodes). State transitions are announced using Cassandra's gossip protocol. As soon as the first streaming session is over, the bootstrapping node enters in state *partial_join* (introduced in our implementation) in which it remains until the end of the streaming process. This state indicates that the node has available data and is able to serve client request on a subset of tokens but is not yet fully integrated to the cluster. To support the *partial_join* state, we introduced a new type of gossip message sent by the new node when a stream session completes, announcing the state and informing the cluster of the tokens the new node is responsible for. Nodes receiving this message update their tokens-to-node mapping, and start to appropriately redirect client requests to the new node.

Modifying the scheduling of token transfers means that we should modify the scheduling of *shadow writes* as well. In incremental elasticity, shadow writes only concern tokens that are being streamed at a specific time period. In standard Cassandra, nodes mark as pending those tokens that the joining node advertises as tokens it will eventually be responsible for. In our implementation, only tokens currently being streamed are shadow-writed until the end of the current streaming session. Tokens that correspond to future streaming sessions are fully served by their current owning nodes and not shadow written. This is possible since the new node announces the pending tokens during the preparation step of each stream session and tokens become available when the session is

complete. Overall this reduces the load that shadow writes place on the joining node during streaming. As soon as all data transfers are complete, the new node announces its state as normal to the cluster.

IV. Evaluation

A. Experimental testbed and methodology

Our experimental testbed is a cluster of 9 servers, each equipped with a dual-core AMD Opteron 275 processor at 2.2GHz with 12GB of main memory. All servers run Ubuntu 14.04 64-bit with a 3.14.1 Linux kernel and are interconnected via a 1Gb/s Ethernet switch. Servers store data on a dedicated 300GB 15,500 RPM SAS drive that delivers 120MB/s in sequential reads and 250 IOPS in random reads with a 4K block size. Hard drives are formatted with the ext4 file system.

We use Cassandra version 3.7 with the OpenJDK 1.8.0-91 Java runtime environment. Our evaluation workload is the Yahoo Cloud Serving Benchmark (YCSB) [17] version 0.11 executing on a dedicated server. The benchmark is configured to produce two different mixes of reads vs. updates/writes: 95%-5% (Workload B) and 50%-50% (Workload A) respectively, with requested keys selected randomly with the Zipf distribution. The YCSB evaluation dataset consists of 80 million unique records resulting to 34GB of data per node when loaded. Our initial Cassandra cluster consists of 7 servers. During elasticity actions an 8^{th} server joins the cluster. The replication factor is set to 3.

We experiment with Cassandra data-consistency levels QUORUM and ALL. QUORUM requires responses from a majority (2 out of 3 in this case) of replicas to complete a read or write operation, whereas ALL involves all replicas. In QUORUM reads, 2 of the 3 replicas return only a checksum of the data, a Cassandra optimization to reduce network traffic.

The number of YCSB client threads (number of parallel connections between database client and servers) is set to 30 and 25 for the QUORUM and ALL consistency levels respectively, empirically determined to stress the cluster while keeping average response time under 50ms (considered a reasonable threshold). We used the default settings for node caches, namely key cache enabled and row cache disabled (however nodes benefit from caching at the OS buffer cache). Unless stated otherwise, we do not limit each node's streaming throughput (stream_throughput_outbound_megabits_per_sec set to 0). For repeatability, we modified the default token partitioner to ensure that each Cassandra node will be responsible for serving the same key ranges across experiments. The dataset is loaded fresh onto Cassandra nodes before each experiment.

Impact of compactions. To isolate the performance impact of SSTable compactions on the joining node from the impact of the streaming process itself we chose to disable compaction activities on the joining node (node 8) during its bootstrapping process. This choice was based on the observation that high compaction activity is significantly penalizing the joining node right after the elasticity action under parallel streaming. It also reflects standard practice in field use of Cassandra [9]. We observed that under incremental streaming, the impact of compactions is spread over time, however for fairness we used the same delayed compaction policy in that case as well. Other solutions to reducing the impact of compactions on the joining node are to limit compaction throughput and the number of active compaction tasks at any time, however an in-depth investigation of this aspect is beyond the scope of this paper.

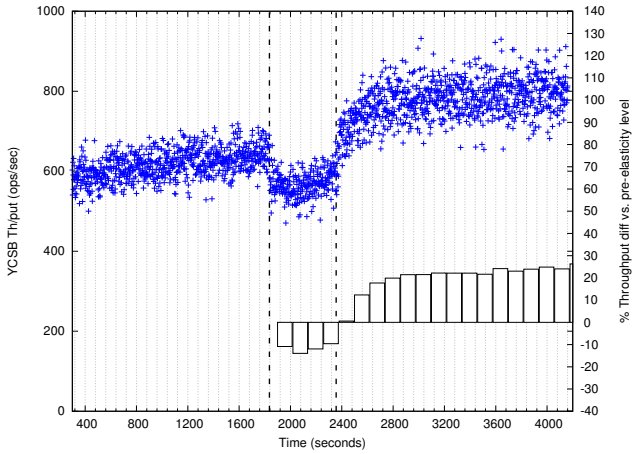
B. Analysis of experimental results

Figure 3 depicts YCSB throughput (YCSB ops/sec) with the workload mix for 95% reads 5% writes and QUORUM consistency. The elasticity action is triggered 30 minutes into the experiment when the system is deemed to have reached a steady state. The duration of streaming activities is highlighted by dashed vertical lines. In the inset we depict a bar chart indicating the relative performance change during the elasticity action vs. pre-elasticity performance. The latter is the average throughput during the most recent 5-minute time window before the elasticity action starts. Each histogram bar corresponds to an average over a 2-minute interval window.

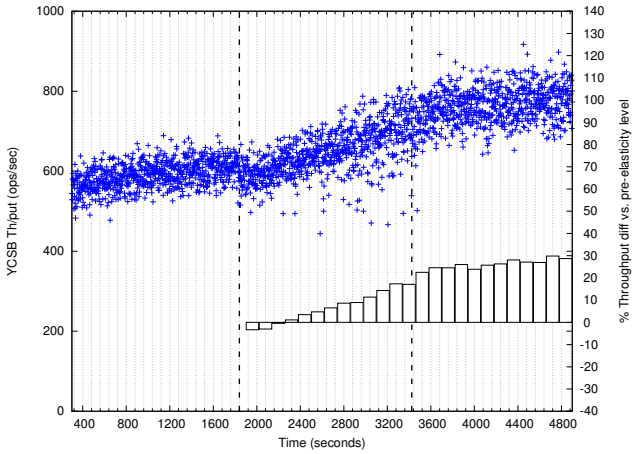
Figure 3a (parallel streaming) exhibits a clear performance hit of about -14% during data streaming, followed by a performance increase (vs. pre-elasticity levels) due to the expansion of the cluster after the elasticity action is over (640 vs. 820 ops/sec). Performance under incremental streaming (Figure 3b) exhibits very little degradation into the early stages of elasticity (-2.5%) and a performance increase via the growing processing capacity of the joining node. This result highlights the benefit of decreased performance impact during elasticity actions with incremental streaming. Streaming takes 26 minutes for incremental vs. 9 minutes for parallel, with both systems taking additional time to reach their full post-elasticity throughput. Taking this additional time into account, we observe that Cassandra with incremental streaming takes about twice the time to reach its full post-elasticity throughput compared to parallel streaming.

To statistically validate the results depicted in Figures 3a and 3b, we calculated the average performance change during streaming over ten runs. The maximum performance hit observed during parallel streaming over those runs was on average -12.96% (standard deviation 1.17%) with a maximum of -15.08%. During incremental streaming, the maximum performance hit over the ten runs was on average -2.32% at the beginning of streaming, with a near-linear decrease as the new node progressively joins the cluster.

One (existing) way to limit the performance impact of streaming is to actively throttle parallel streaming transfers through a Cassandra configuration setting (stream_throughput_outbound_megabits_per_sec). To highlight the benefits of incremental streaming over this solution, we repeat the previous experiment using standard Cassandra set for limiting streaming throughput to a level comparable to the aggregate network throughput achieved with incremental streaming. Under incremental streaming (Figure 3b), the joining node receives tokens at 210Mbps (vs. 490Mbps with unthrottled



(a) Parallel streaming



(b) Incremental streaming

Fig. 3: Elasticity under YCSB workload B (95% read, 5% writes), consistency QUORUM

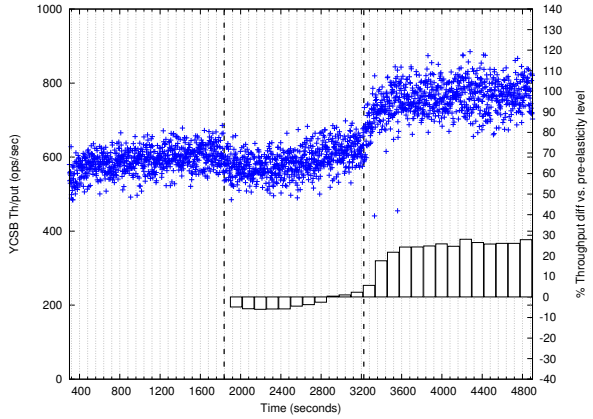


Fig. 4: Parallel streaming with network transfer throttling (same configuration as in Figure 3)

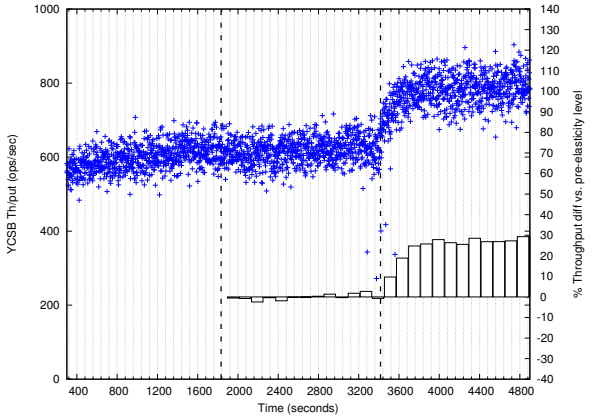
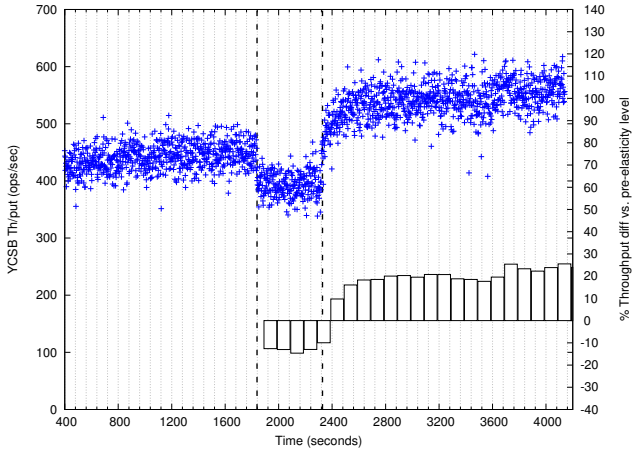
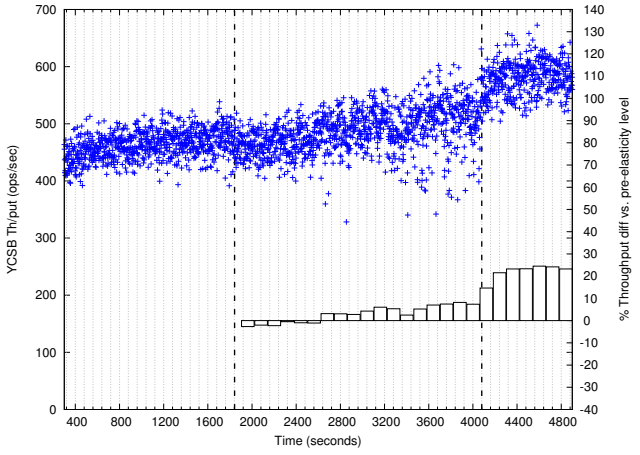


Fig. 5: Serial *but not incremental* streaming (same configuration as in Figure 3)

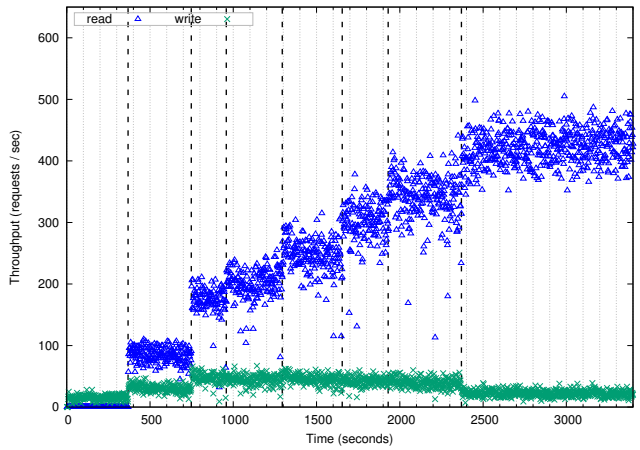


(a) Parallel streaming

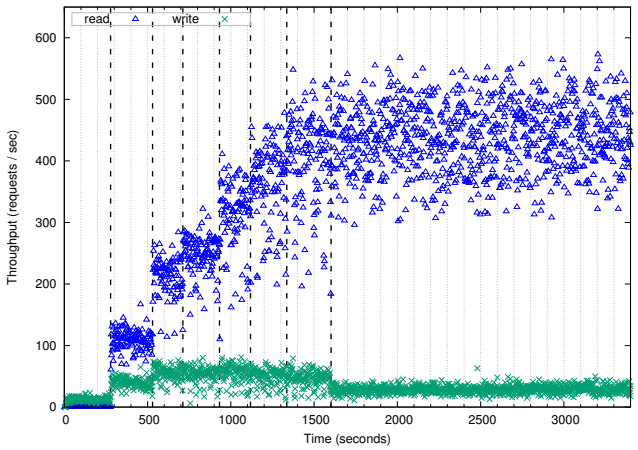


(b) Incremental streaming

Fig. 6: Elasticity under YCSB workload B (95% read, 5% writes), consistency ALL



(a) Consistency Level ALL



(b) Consistency Level QUORUM

Fig. 7: Request rate served by joining node during streaming, YCSB workload B (95% reads, 5% writes)

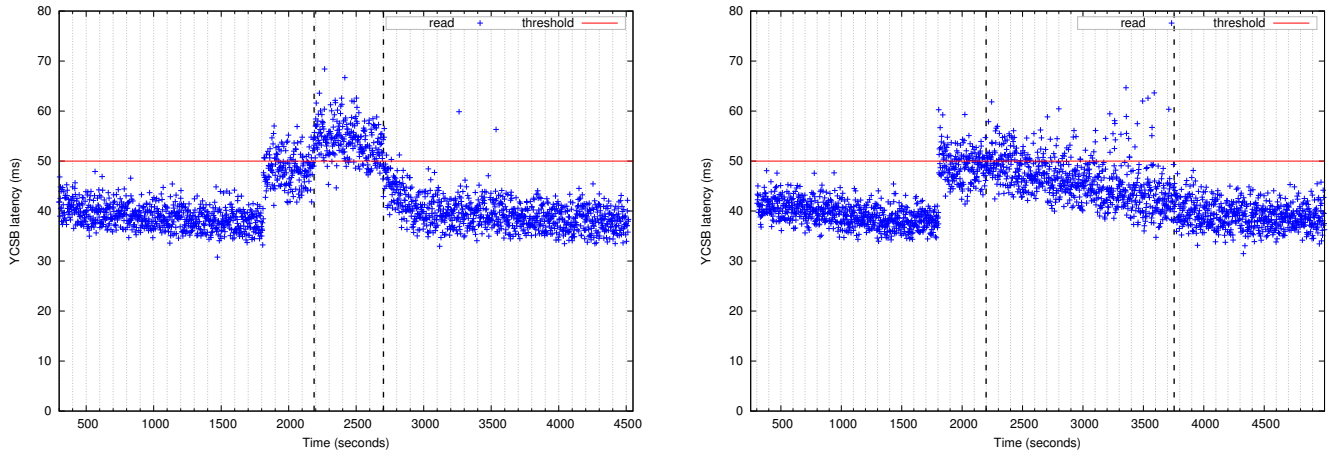
parallel transfers). We thus set the throughput limit of each (parallel) sender to 30Mbps so that the joining node receives at the same rate (30×7=210Mbps) as in incremental streaming. Figure 4 shows that the performance impact under throttled parallel transfers is reduced to -6% over a 23-minute elasticity interval vs. -14% over 9 minutes in unthrottled parallel streaming. Receiving at 210Mbps however, means that a large fraction of the processing capacity of the joining node goes unused, explaining the performance advantage of incremental elasticity.

Another way to showcase the positive impact of increasing the processing rate of the joining node during elasticity is to contrast it with a system that schedules sequential pair-wise transfers (as in incremental streaming) but does not make transfered tokens incrementally available on the joining node. Results with such a system (Figure 5) demonstrate that performance during elasticity remain at the same level as before elasticity, through the end of the streaming process: Although the joining node receives data from the existing servers in pair-wise manner (similar to incremental elasticity) the transfered data are not available for access to the new node. This means that the capacity of the cluster remains the same until the end of elasticity action when the new node announces the ownership of the received tokens and is fully integrated to the system (as in parallel streaming).

The increase of the cluster processing capacity in a step-wise fashion can also be observed by examining the rate of requests handled at the new (joining) server. Figure 7 illustrates that the joining node indeed serves progressively more client requests (reads/writes served locally –excluding those just coordinated– by the joining node) during streaming. The time between two dashed lines in the graph corresponds to a streaming session between the joining node and an existing server. Each data point represents the rate of requests served in a time window of two seconds. Figure 7a shows the requests that are served by the joining node under consistency level ALL. In this case each request must be acknowledged by all replicas, thus the joining node is involved on all requests for tokens it has already received. Figure 7b depicts the number of client requests served by the joining node under consistency level QUORUM. In both cases, we can observe the increasing number of requests as the streaming process progresses. In case of consistency level QUORUM the steps are somewhat more noisy (the concentration of points is wider) as the new node may or may not be asked to participate in the majority (2 of 3) of replicas that serve a request. In Figure 7 we observe that the rate of write requests is higher during the elasticity action in both cases. This is due to the extra overhead of shadow writes during streaming.

Figure 6 depicts YCSB throughput for the same dataset for 95% reads 5% writes (workload B) under the stricter consistency level ALL. We observe similar trends as with the previous configuration applying consistency level QUORUM. With parallel streaming the throughput drops up to -17% during the elasticity action. Incremental streaming exhibits little degradation (-2.5%) at the start of the streaming phase. Under this configuration the performance increases in smaller steps during the elasticity action. Although with incremental elasticity one streaming node is active at a time, every request involves all replicas and thus performance is determined by the slowest replica. A small fraction of requests corresponding to tokens that are part of the active streaming session involve the streaming source node, and thus are expected to be impacted due to its higher load during streaming. Requests that do not hit this node will not be impacted under this consistency level.

Repeating these experiments with the 50%-50% read/write workload mix shows similar trends with the exception that the performance impact during streaming increases under both parallel and incremental streaming. With QUORUM consistency, parallel streaming exhibits steady performance degradation (similar to that observed Figure 3) up to -11%, while incremental elasticity exhibits a smoother transition to the new configuration with a maximum performance hit less than -4% during the early stages of elasticity, increasing to +10% as



(a) Parallel streaming

(b) Incremental streaming

Fig. 8: YCSB latency under workload B (95% read, 5% writes), consistency QUORUM

the streaming process progress. With consistency level ALL, parallel streaming exhibits a performance penalty up to -16%, whereas incremental elasticity exhibits a performance hit of up to -6.5% at the early stages of streaming, increasing slightly over the base line (+3%) until the elasticity action completes.

C. Response time

The response-time trends observed in previous experiments are identical to those observed for throughput, with throughput drops corresponding to response-time increases (response-time figures omitted due to space limitations). In this section we aim to extend our evaluation in two directions: First, to provide a response-time view of incremental vs. parallel streaming. Second, to demonstrate a scenario of a variable-load client and a simple elasticity controller that drives an elasticity action to accommodate the increased load. Our evaluation sets the incremental elasticity technique into context with quality of service (QoS)-aware data stores that often rely on response-time targets to maintain specific goals [6], [18], [19], [20].

Variable levels of client load may occasionally result to a system exceeding a fixed response-time goal. A QoS-aware controller typically incorporates a monitoring component [20] that combines response-time metrics reported across client processes, analyzes, plans, and executes elasticity actions to re-provision service capacity as needed. We do not aim to fully evaluate an elasticity controller in this paper. We use a simple such controller to compare the impact of incremental vs. parallel elasticity in maintaining a given response-time target. Figure 8 depicts YCSB read latency with a 95% read, 5% writes workload mix, consistency QUORUM, under variable load. The targeted response time of 50ms is highlighted with a horizontal red line in Figure 8.

Load generated by 20 YCSB threads results initially in 40ms average response time for read operations. 30 minutes into the experiment when the system has reached steady state, we increase the load by adding 10 more client threads to a total of 30, leading to an increase of the average response time above 50 ms. About 5 minutes later, the controller detects a response-time target violation and triggers an elasticity action whose duration is highlighted by dashed vertical lines. Each data point in Figure 8 is average response-time over a 2-second window. Under parallel streaming (Figure 8a) we observe that the elasticity action leads to a further increase of response times, further exceeding our target. In contrast, incremental elasticity (Figure 8b) has lower impact on response time, producing fewer target violations and achieving a smoother transition to the new configuration.

To quantitatively compare incremental to parallel streaming with respect to violations of the response-time service-level objective (50ms in this case) that they produce, we count the proportion of those requests with response time exceeding our objective, using the following formula:

$$\text{score} = \sum_{i=S_1}^{S_n} \text{penalty}_i \tag{1}$$

where S_1 and S_n are the first and last interval of the streaming process, and

$$\text{penalty}_i = \begin{cases} 1, & \text{if resp. time} > 50 \\ 0, & \text{if resp. time} \leq 50 \end{cases} \tag{2}$$

The score function with the above penalty expresses the amount of time intervals that the elasticity process results in a target violation. The execution of Figure 8a (parallel streaming) results to a score of 237 (237 x 2 = 474 seconds) where the response time exceeds the targeted threshold, whereas the execution of Figure 8b (incremental streaming) has a score of 144 (144 x 2 = 288 sec), thus incremental streaming has 39% fewer response-time violations compared to parallel streaming.

In the above analysis, any violation of the response-time target is considered equally harmful and contributes a penalty of 1. However, one can argue that the amount by which the threshold is violated is also practically relevant. We can

capture this factor by replacing formula (2) with (3) in the score function, thus taking into account by how many *ms* the average response time exceeds the target at each data point.

$$\text{penalty}_i = \begin{cases} 50 - (\text{resp. time}) & \text{if resp. time} > 50 \\ 0 & \text{if resp. time} \leq 50 \end{cases} \quad (3)$$

According to this penalty function, parallel streaming has a score of -1170 while incremental streaming has a score of -440, indicating that incremental elasticity scores 2.6x higher than parallel streaming when taking into account both the number of response-time violations and their extent.

In Figure 8 we observe that Cassandra with parallel transfers is about twice as fast in bringing response time to its stable post-elasticity level compared to our implementation using incremental streaming. Thus if an application is insensitive to service-level violations or performance cost during elasticity, parallel streaming may be a preferable elasticity mechanism.

V. Related work

NoSQL systems, like nearly all data-intensive systems, require reconfiguration over time so that data is redistributed across server nodes for the purpose of load balancing, expanding/shrinking resources, or rehashing data to server nodes. Elasticity is a special form of reconfiguration where a fraction of the overall dataset is moved to new nodes (or taken out of nodes about to be decommissioned) to grow or shrink processing capacity in an online manner. In what follows we look into previous research on reconfiguration in NoSQL data stores, with a special focus on elasticity.

Several data stores are able to expand or shrink their use of server resources through migration of shards and replicas. Petal was an early elastic storage system offering a virtual disk abstraction featuring incremental reconfiguration [21], namely the ability to re-stripe a logical disk on fewer or more storage servers without blocking access to the entire disk. Incremental elasticity shares the concept of transferring a piece of the overall state at a time but has different goals.

Several NoSQL data stores use *consistent hashing* to map key ranges to server nodes. The basic consistent hashing algorithm presents some challenges. First, the random position assignment of each node on the ring may lead to non-uniform data and load distribution. Second, the basic algorithm is oblivious to the heterogeneity in the performance of nodes. To address these issues, a number of NoSQL data stores (among them Dynamo [6], Cassandra [5], Riak [7], and Voldemort [8]) use a variant of consistent hashing where instead of mapping a node to a single point in the circle, each node gets assigned to multiple points in the ring, each such point called a *virtual node*. This approach was originally introduced by Karger *et al.* [12] and Stoica *et al.* [22]. Each node in the NoSQL system can be responsible for more than one virtual node. Systems using virtual nodes involve all-to-one transfers during elasticity and can benefit from the use of incremental elasticity.

Ghosh *et al.* [23] described Morphus, a methodology for supporting online reconfigurations in sharded NoSQL systems. Morphus introduces algorithms for re-sharding a database aiming to reduce the total network transfer volume during reconfiguration and to achieve a load balanced assignment. Morphus is implemented within MongoDB and applied to the case of changing the sharding key of an existing table. It leverages internal MongoDB mechanisms to reconfigure secondary shard replicas and transfer data via many-to-all communication patterns inherent in such resharding scenarios. Changing the sharding key entails a heavier reconfiguration compared to adding new servers considered in this paper.

DDS [24] was an early scalable key-value store with the ability to split partitions (shards) and migrate replicas between nodes. DDS chooses shard size so as migration of a full shard is a quick process and opts for migrating replicas at the full speed of the network rather than a controlled, background process. The authors of DDS do not detail a complete elasticity mechanism. The term *incremental elasticity* has also been used in the context of array databases [25]. In this work, Duggan *et. al.* design an elasticity controller that decides when to expand an array database cluster and how to repartition a growing multi-dimensional data set within it. Our use of the term refers to progressive increases of processing capacity in a fine-grain manner during an elasticity action and is thus complementary.

GoogleFS and HDFS support elasticity by migrating replicas of file blocks between nodes. HBase, a NoSQL data store using HDFS as a storage back-end, handles elasticity at two levels: at the higher level, addition of an HBase node takes over responsibility for a fraction of shards, without immediately migrating the HDFS data blocks associated with those shards. These migrations happen at the next HBase compaction when new HDFS files are being created.

MongoDB utilizes the cluster balancer module, which migrates data chunks between different shards when the chunk number ratio of the biggest shard to the smallest one reaches a certain threshold. MongoDB appears to support an elasticity mode where data is being served by newly added nodes as soon as they arrive [26]. Based on the scarce documentation available for this feature, it does not appear related in any other way to incremental elasticity. Voldemort [8] uses a rebalance controller that allows a joining node to participate in multiple parallel transfers, ensuring that each sender has only one active token-transfer at a time. Riak [7] supports the adjustment of the number of node-to-node transfers using a per-node *transfer_limit* parameter (default 2). It appears that this parameter controls the number of tokens involved in data transfers rather than the number of parallel transfers towards a joining node. It has been used on disk-capacity issues in expanding clusters [27]) but we have found no systematic evaluation of its performance impact or a specification of how data transfers using it are coordinated/managed across nodes.

Konstantinou *et al.* [28] describe a generic distributed module, DBalancer, that can be installed on top of a typical NoSQL data-store and provide an efficient configurable load balancing mechanism. Balancing is performed by simple message exchanges and typical data movement operations supported by most modern NoSQL data-stores. In a related work, Konstanti-

nou *et al.* [29] present a cloud-enabled framework for monitoring and adaptively resizing NoSQL clusters. Kuhlenkamp *et al.* [16] evaluate the elasticity of HBase and Cassandra and show a tradeoff between the speed of scaling and the performance variability while scaling. These works rely on the use of generally available elasticity mechanisms and thus could benefit from the use of the *incremental elasticity* mechanism.

Data stores offering performance-oriented service-level agreements (SLAs) such as Dynamo [6] use online elasticity mechanisms as one among different ways of adjusting processing capacity to fit client needs. Typically, a combination of vertical (increasing single-node processing capacity) and horizontal (increasing number of nodes) elasticity is used in this space, also known as scale-up and scale-out elasticity. Incremental elasticity is an instance of the latter. Recent work [20] proposed a measurement-based prediction approach to data store SLA by means of elasticity actions over the Cassandra NoSQL store. This paper improves over this approach by reducing the impact of elasticity actions as the data store adapts to workload changes.

Brown et al. [30] introduce a methodology for benchmarking the availability of RAID arrays after a disk crash, highlighting a trade-off between the speed of data reconstruction and its impact on application performance, similar in spirit to the elasticity tradeoff explored in this paper. They study the policies used by different software RAID implementations (Solaris, Linux, Windows) and find that Linux follows the slowest approach, dedicating less disk bandwidth to reconstruct data posing no significant effect on application performance, whereas Solaris defines the opposite extreme making its RAID reconstruction over 7 times faster than Linux, albeit at a significant performance hit during reconstruction.

VI. Conclusions

In this paper we propose incremental elasticity as a way to reduce the performance penalty incurred during elasticity actions in distributed data stores. Our implementation on the Cassandra column-oriented data store shows that such an implementation is feasible with reasonable complexity. Our evaluation shows that incremental elasticity results in smoother, more stable elasticity actions compared to parallel network transfers across all cases considered. Incremental elasticity reduces the performance impact (-2.32% vs. -12.96% drop in aggregate throughput for 95%-5% reads/writes and QUORUM consistency, -2.5% vs. -17% under 95%-5% reads/writes and ALL consistency) compared to parallel network transfers. In a scenario involving a service-level management controller, incremental elasticity leads to 39% fewer response-time violations compared to parallel streaming during elasticity.

References

[1] Cisco, "Cisco global cloud index: Forecast and methodology 2013-2018 white paper," https://www.terena.org/mail-archives/storage/pdfVVqL9tLHLH.pdf, accessed: 07/2017.

[2] "Amazon DynamoDB," https://aws.amazon.com/dynamodb/, accessed: 07/2017.

[3] "Using DynamoDB in production," http://blog.sendwithus.com/using-dynamodb-production/, accessed: 07/2017.

[4] "Falling in and out of love with DynamoDB," http://0x74696d.com/posts/falling-in-and-out-of-love-with-dynamodb-part-ii/, accessed: 07/2017.

[5] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, Apr. 2010.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proc. of 21st ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07*, Stevenson, Washington, USA, 2007.

[7] Basho Riak NoSQL database. http://docs.basho.com/riak/kv/. Accessed: 07/2017.

[8] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project Voldemort," in *Proc. of the 10th USENIX Conference on File and Storage Technologies, FAST'12*, San Jose, CA, 2012.

[9] Bootstrapping performance improvements for Leveled Compaction. http://www.datastax.com/dev/blog/bootstrapping-performance-improvements-for-leveled-compaction. Accessed: 07/2017.

[10] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan, "Measurement and analysis of TCP throughput collapse in cluster-based storage systems," in *Proc. of the 6th USENIX Conference on File and Storage Technologies, FAST'08*, San Jose, California, 2008.

[11] A. Papaioannou and K. Magoutis, "Incremental elasticity for nosql data stores," in *Proc. of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Atlanta, GA, USA, 2017.

[12] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web," in *Proc. of the 29th Annual ACM Symposium on theory of Computing, STOC'97*, El Paso, Texas, United States, 1997.

[13] How data is distributed across a cluster (using virtual nodes). "https://docs.datastax.com/en/cassandra/3.x/cassandra/architecture/archDataDistributeDistribute.html". Accessed: 07/2017.

[14] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The Log-structured Merge-tree (LSM-tree)," *Acta Inf.*, vol. 33, no. 4, Jun. 1996.

[15] Read repair: repair during read path. "https://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsRepairNodesReadRepair.html". Accessed: 07/2017.

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

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proc. of the 1st ACM Symposium on Cloud Computing, SoCC'10*, Indianapolis, Indiana, USA, 2010.

[18] C. R. Lumb, A. Merchant, and G. A. Alvarez, "Facade: Virtual storage devices with performance guarantees," in *Proc. of the 2nd USENIX Conference on File and Storage Technologies, FAST'03*, San Francisco, CA, 2003.

[19] M. Karlsson, C. Karamanolis, and X. Zhu, "Triage: performance isolation and differentiation for storage systems," in *Proc. of the 12th IEEE International Workshop on Quality of Service, IWQOS '04*, Montreal, Canada, 2004.

[20] M. Chalkiadaki and K. Magoutis, "Managing service performance in NoSQL distributed storage systems," in *Proc. of 5th IEEE International Conference on Cloud Computing Technology and Science, CloudCom'13*, Bristol, UK, 2013.

[21] E. K. Lee and C. A. Thekkath, "Petal: Distributed virtual disks," *SIGOPS Operating Systems Review*, vol. 30, no. 5, pp. 84–92, Sep. 1996.

[22] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc. of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM'01*, San Diego, California, USA, 2001.

[23] M. Ghosh, W. Wang, G. Holla, and I. Gupta, "Morphus: supporting online reconfigurations in sharded NoSQL systems," in *Proc. of the 2015 IEEE International Conference on Autonomic Computing*, ser. ICAC '15, Washington, DC, USA, 2015.

[24] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler, "Scalable, distributed data structures for internet service construction," in *Proc. of the 4th Conference on Symposium on Operating System Design & Implementation, OSDI'00*, San Diego, California, 2000.

[25] J. Duggan and M. Stonebraker, "Incremental elasticity for array databases," in *Proc. of the 2014 ACM SIGMOD International Conference on Management of Data*, Snowbird, Utah, USA, 2014.

[26] T. Dory, B. Mejas, P. Van Roy, and N.-L. Tran, "Measuring elasticity for cloud databases," in *Proc. of the 2nd International Conference on Cloud Computing, GRIDs, and Virtualization*, Rome, Italy, 2011.

[27] Riak a successful failure. https://www.slideshare.net/GiltTech/riak-a-successful-failure-11512791. Accessed: 07/2017.

[28] I. Konstantinou, D. Tsoumakos, I. Mytilinis, and N. Koziris, "DBalancer: Distributed load balancing for NoSQL data stores," in *Proc. of SIGMOD'13*, New York, NY, USA, 2013.

[29] I. Konstantinou, E. Angelou, D. Tsoumakos, C. Boumpouka, N. Koziris, and S. Sioutas, "Tiramola: elastic NoSQL provisioning through a cloud management platform," in *Proc. of the International Conference on Management of Data, SIGMOD*, Scottsdale, Arizona, USA, 2012.

[30] A. Brown and D. A. Patterson, "Towards availability benchmarks: A case study of software raid systems," in *Proc. of the USENIX Annual Technical Conference*, San Diego, California, 2000.