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*—Elasticity actions in NoSQL data stores move large amounts of data over the network to take advantage of new resources. Here we propose *incremental elasticity*, a new mechanism for scheduling data transfers to a joining server, leading to smoother elasticity actions with a reduced performance impact.**

I. Introduction

There is usually a trade-off between the duration of elasticity actions and their performance impact. In this work we focus on the performance impact of data elasticity actions in NoSQL data stores. In particular we target data stores that partition data horizontally in a fine-grain manner (referring to data partitions as *shards*) and spread them as far as possible on the available nodes for better load balancing. Several popular data stores [1], [2], [3], [4] map keys to nodes using consistent hashing [5]. The specific variation of consistent hashing used in this work associates each physical node with a number of *tokens* hashed to a ring. Each token identifies a *key range*, starting from the previous token up to it. Keys are mapped first to tokens and then to nodes. Cluster expansion via the addition of a new node introduces a number of new tokens to the ring. The new node takes responsibility for key ranges identified by its tokens, and receives the corresponding data via network transfers (*streaming sessions*) from previous owners.

One common option to carry out these network transfers [1], [3], [4] 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. Second, the new node is solely engaged in data transfer at the speed of its network link and cannot contribute processing capacity until 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. Fourth, the many-to-one communication pattern in this phase is known to be a cause of throughput collapse in data centers under certain circumstances [6]. Our *incremental elasticity* technique introduced here aims to address these challenges.

Incremental elasticity replaces parallel network transfers with a sequential communication schedule where senders take turns sending tokens to the new node. As soon as a transfer is over, the associated tokens are becoming available for access on the new node, while a subsequent transfer of tokens is taking place. The expected performance benefits of incremental elasticity can be summarized as follows: Fewer nodes are involved in network transfer at any time, reducing the overall performance drop during elasticity. With tokens becoming available on the new node as soon as token transfers complete, processing capacity should increase in a step-wise incremental fashion. When only a single sending server is active at a time, the impact due to its higher load may be masked by other replicas of the tokens that this sender owns.

Our contributions include a new elasticity mechanism for NoSQL data stores called incremental elasticity; an implementation in the context of the Cassandra column-oriented key-value store [1]; and an experimental evaluation of the benefits of incremental elasticity vs. simultaneous parallel network transfers under Yahoo! Cloud Serving Benchmark workloads.

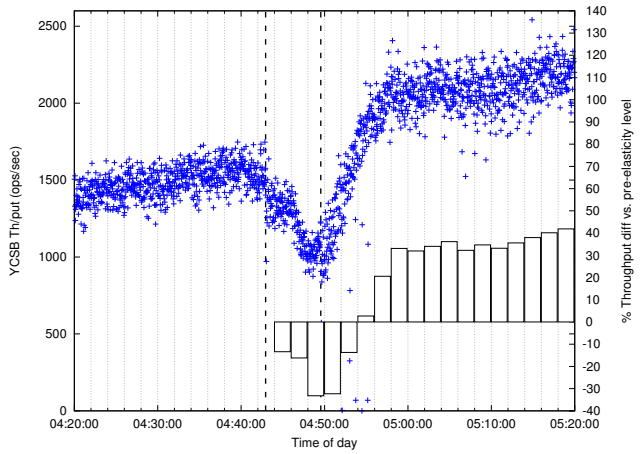
II. Design and implementation

To enable incremental elasticity we must coordinate data transfers to a node joining the cluster, scheduling consecutive pair-wise transfers between each of the existing nodes and the new node. As soon as a streaming session is complete, the new node starts to serve client requests on that subset of tokens. In our implementation (extending Apache Cassandra version 3.7, particularly its streaming and gossip components) the new node enters an intermediate state called *incremental* in which it can serve requests while still streaming. Initially, the node is in the *bootstrapping* state in which it does not serve client requests. When the first streaming session is over, it enters the *incremental* state, announcing it through a new type of gossip message that informs the cluster that the joining node has new data to serve. Nodes receiving this message update their tokens-to-node mapping, redirecting client requests to the new node. When all sessions are over, it enters the *normal* state indicating that it has been fully integrated to the cluster.

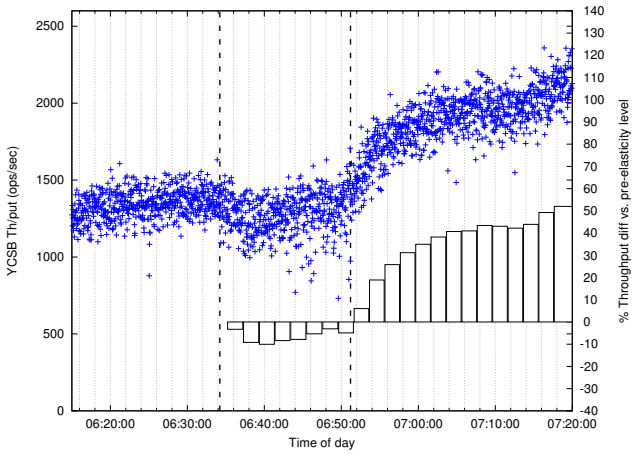
Modifying the scheduling of token transfers means that we must modify the handling of *shadow writes* (writes to keys that are being streamed, going both to originating and target nodes). In our implementation, only the tokens currently being streamed are marked pending and being shadow-written until the respective data transfer completes. This reduces the load that shadow-writes place on the joining node during streaming.

III. Evaluation

Our experimental testbed is a cluster of 7 servers, each equipped with a dual-core AMD Opteron 275 processor clocked 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



(a) Parallel streaming



(b) Incremental streaming

Fig. 1: Elasticity under YCSB workload A (50% read, 50% writes), consistency QUORUM, larger dataset (45 million records)

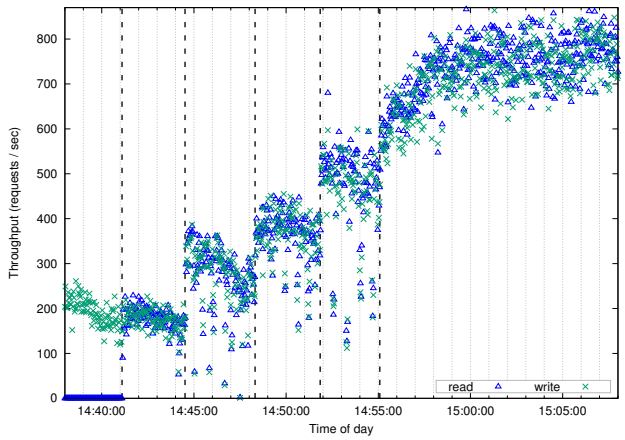


Fig. 2: Request rate served by joining node during streaming

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) [7] version 0.11 executing on a dedicated server. The benchmark is configured to produce a 50%-50% mix of reads vs. updates/writes with requested keys selected uniformly at random. Our initial Cassandra cluster consists of 5 servers. During elasticity actions a 6^{th} node joins the cluster. The replication factor is set to 3.

We use a 45M-record YCSB dataset (27GB of starting data per node) resulting in an I/O bound system. The number of YCSB client threads is set to 10, empirically determined to stress the cluster while keeping average response time under 40ms (considered a reasonable threshold). We used the default settings for node caches, namely key cache enabled and row cache disabled. 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.

Figure 1 presents the YCSB throughput involving a 50%-50% read/write mix (workload A) with consistency level QUORUM. Elasticity with parallel streaming (Figure 1a) exhibits a strong performance penalty ranging from -14% to -34% at 04:50h. Under incremental streaming (Figure 1b) the system exhibits a smoother transition to the new configuration with a maximum performance hit slightly below -10% at 06:40h.

Figure 2 illustrates that the joining node indeed serves progressively more client requests (reads/writes served locally –not coordinated– by the joining node) during streaming.

These results show that incremental elasticity results in smoother, more stable elasticity actions compared to parallel network transfers, demonstrating that incremental elasticity can be an important mechanism towards rapid adaptation with low performance impact in NoSQL data stores.

REFERENCES

[1] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[2] G. DeCandia *et al.*, "Dynamo: Amazon's Highly Available Key-value Store," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*, Stevenson, WA, USA, 2007.

[3] (2016) Basho Riak. [Online]. Available: http://docs.basho.com/riak/kv/

[4] R. Sumbaly *et al.*, "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, USA, 2012.

[5] D. Karger *et al.*, "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, TX, USA, 1997.

[6] A. Phanishayee *et al.*, "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)*, 2008.

[7] 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, USA, 2010.