# DBalancer: Distributed Load Balancing for NoSQL Data-stores

Ioannis Konstantinou[⋆] Dimitrios Tsoumakos[◊] Ioannis Mytilinis[⋆] Nectarios Koziris[⋆]

[⋆]CSLAB, National Technical University of Athens
{ikons,gmytil,nkoziris}@cslab.ece.ntua.gr

[◊]Department of Informatics Ionian University
dtsouma@ionio.gr

## ABSTRACT

Unanticipated load spikes or skewed data access patterns may lead to severe performance degradation in data serving applications, a typical problem of distributed NoSQL data-stores. In these cases, load balancing is a necessary operation. In this demonstration, we present the *DBalancer*, a generic distributed module that can be installed on top of a typical NoSQL data-store and provide an efficient and highly configurable load balancing mechanism. Balancing is performed by simple message exchanges and typical data movement operations supported by most modern NoSQL data-stores. We present the system's architecture, we describe in detail its modules and their interaction and we implement a suite of different algorithms on top of it. Through a web-based interactive GUI we allow the users to launch NoSQL clusters of various sizes, to apply numerous skewed and dynamic workloads and to compare the implemented load balancing algorithms. Videos and graphs showcasing each algorithm's effect on a number of indicative performance and cost metrics will be created on the fly for every setup. By browsing the results of different executions users will be able to grasp each algorithm's balancing mechanisms and performance impact in a number of representative setups.

## Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Distrib. databases*

## Keywords

NoSQL, Load balancing, Cloud computing

## 1. INTRODUCTION

The data explosion we are witnessing has pushed centralized data-management approaches to their limits. Where strict adherence to the relational and ACID model is not required, a number of distributed, shared-nothing, systems called NoSQL engines [1] have been proposed. Balancing large and dynamically changing numbers of user requests in such distributed systems has always been an area of focus. Load balancing techniques based, for instance, in consistent hashing [13], are employed by the majority of modern NoSQL stores [17, 11] to equally allocate data and incoming requests to the available nodes. Although hashing initially solves the data to machines allocation problem, there are many situations in which this proves suboptimal.

First, although data placement can be equally balanced, this is not the case with data access in highly skewed distributions where a small portion of popular data may get the majority of the applied load [9]. This is a primary reason for degraded performance even in over provisioned infrastructures. For instance, on-line gaming clusters that handle user requests can experience a high density of items and queries around a specific terrain area (e.g., users during a "battle") served by a single server [18]. Skew detection and handling is more difficult in distributed environments [16] in which coordination is not straight-forward. Moreover, unanticipated high loads that occur due to a popular event, e.g., the death of Michael Jackson [2], the earthquake in Japan, etc, lead to the flash crowd effect [12], in which the computing infrastructure fails to accommodate the high volume of incoming requests.

Second, since hashing destroys locality, it cannot be employed in situations where semantically close items need to be stored in an order-preserving way in order to support range queries. Range queries are present in many popular applications and include queries such as "find all hospitals between `3rd Avenue` and `7th Avenue`" and prefix queries like "find all words that begin with the letters `goo`", etc.

Consequently, NoSQL systems offer load balancing methods that re-partition and re-distribute data items between nodes. For instance, Cassandra's load balancing [3] is performed only during node additions by transferring half of the keys of the most loaded node, or with a user-executed script. This operation does not minimize the amount of transferred data and does not balance data or user-generated load. Voldemort [21] supports a similar operation [4] with the same drawbacks: centralized and user-triggered execution. It uses a static number of partitions and its goal is to allocate them "evenly" to physical nodes. Riak [5] uses the notion of vnodes, inspired by Chord's [20] virtual nodes, and assigns them to physical nodes in a many-to-one manner. Balancing is performed only during node addition/removals and lacks customization. MongoDB utilizes the cluster balancer module [6], which migrates data chunks between different shards when the chunk number ratio of the biggest shard to the smallest one reaches a certain threshold. This is a similar operation that lacks flexibility, apart from the threshold parameter setting. Hadoop's HDFS balancer [7] works in the same way: when a node's utilization crosses the mean cluster's utilization over a user-defined threshold, a rebalancing operation is initiated.

In any case, we notice that all load balancing solutions are not automated, non-customizable and do not always lead the system in a balanced state in a fast and efficient way. What is more, although every balancing method utilizes the same primitive operations (i.e., data transfer operations between servers with different loads), it is designed and implemented with only a specific system in mind, leading to a suite of similar and not generally applicable approaches.

In this work, make the following contributions:

- We present the *DBalancer*, a generic distributed module that performs fast and cost-efficient load balancing on top of any distributed NoSQL datastore. The *DBalancer* offers the following novel features:
  **Datastore Abstraction:** the *DBalancer* is completely independent of the underlying NoSQL data-store. Data-store users need only define a set of specific primitive actions such as item movement and routing table information/management commands to take advantage of the *DBalancer*'s features.
  **Algorithm abstraction:** New load balancing algorithms can be easily defined by giving a set of message types and actions along with the appropriate trigger conditions.
  **Distributed operation:** The module does not need any central co-ordination point in order to operate. it is installed on a per-node basis and is functional even when only a portion of the network's nodes is operational.
  **Light-weight design:** the *DBalancer* has been designed to have the least impact on the running node. Only a small number of sync and probing messages are utilized in order to co-ordinate the load balancing procedure.
- We build the *DBalancer* on top of Cassandra [17] a state-of-the art distributed NoSQL data-store.
- We present a suite of load balancing algorithm implementations on top of the *DBalancer* a state of the art algorithm proposed by [14], and a collection of algorithms presented in [15].

In this demonstration, the participants will be able to examine the behavior of the *DBalancer* when varying the following: (a) cluster setup: allow for different cluster size ranging from 4 to 40 nodes of different hardware configurations for various number of stored items and different *DBalancer* algorithm setup with a selection between *NIX*, *MIG*, *NIXMIG* and *IB* algorithms (b) workload: select from a variety of skewed and dynamic workload types. Zipfian, where the probability of a key $i$ being asked is analogous to $i^{-\theta}$, or pulse-like, where a range of keys has a constant load and the rest of the keys are not requested, workloads with a varying degree of skewness will be used. Moreover, dynamic workloads in which the load suddenly changes its skew will be employed. The participants will be able to watch on-the-fly generated videos showing the effect on the cluster nodes' load during time when the algorithms redistribute items and nodes. Apart from this, a set of real-time graphs and aggregated system statistics such as the number of messages and keys transferred, the variation of the system's imbalance, mean query response time and throughput, CPU usage, network traffic, etc during time for each execution will be presented.

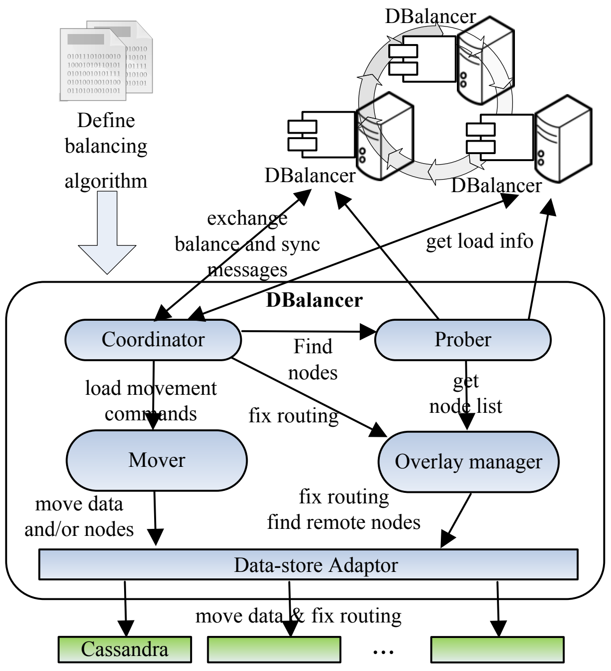


**Figure 1: The *DBalancer* Architecture**

## 2. ARCHITECTURE

The *DBalancer* features an architecture that is illustrated in Figure 1. In essence, the *DBalancer* component runs in every datastore's node and it is configured with the desired balancing algorithm. It performs message exchanges in order to find balancing partners, co-ordinate the balancing procedure and collect load information (upper part of Figure 1). When the appropriate nodes have been found and reserved, it utilizes the data-store's specific implementations to exchange keys and fix routing table information between balancing partners (lower part of Figure 1). The *DBalancer* consists of the following modules:

**Coordinator Module:** It initiates or participates in balancing decisions. It takes as input and implements the user's load balancing algorithm. The balancing co-ordination is performed by simple message exchanges, in which the node's load status, current state and desired action is disseminated to other nodes. This module decides each node's role in the balancing procedure according to the collected messages and load statuses. It also resolves all conflicts that may occur because of concurrent load balancing attempts and makes sure that the whole procedure is being carried out. When the decisions are made, higher level commands are issued at the mover and the overlay manager modules.

**Prober Module:** It locates remote nodes and examines their load. Remote nodes are found by contacting the Overlay manager module, which contains information about other data-store nodes. It exchanges simple messages containing load statuses and maintains a list with the observed nodes and their respective loads. This list is used by the coordinator module when it wishes to locate remote underloaded nodes.

**Mover module:** it translates higher level load movement commands into specific item exchanges or node migrations. For instance, a higher level command instructing a movement of 10 load units from Node A to Node B may be translated into the movement of all items between $[N_{Afrom}..N_{Ato}]$ from Node A to Node B. Higher level node migration commands are translated as follows: the remote nodes first transfer their data to (often) neighboring nodes and then they join next to the overloaded node and take a part of its load. When the specific actions have been calculated, they are forwarded to the lower-level Data-store adaptor module which translates them into application specific calls.

**Overlay Manager Module:** It abstracts lower-level data-store specific routing information and operations. It connects with the Data-store Adaptor module to get and set routing table information. The Coordinator module contacts this module when the load movement commands require overlay maintenance operations. For instance, during node migrations the routing table entries need to be refreshed. Moreover, this module also provides a list with node addresses to the prober module which acts as a pool for possible message receivers.

**Data-store Adaptor module:** It hides the complexities of the underlying data-store implementations, by offering a set of abstract operations to the higher level modules. For each supported data-store, a set of different commands need to be implemented or utilized. For instance, a node movement operation in the Cassandra case results in a `nodetool move` command, whereas a data movement operation in the HBase case results in a set of `HDFS` data block movements between different DataNodes.

## 3. ALGORITHM DESCRIPTIONS

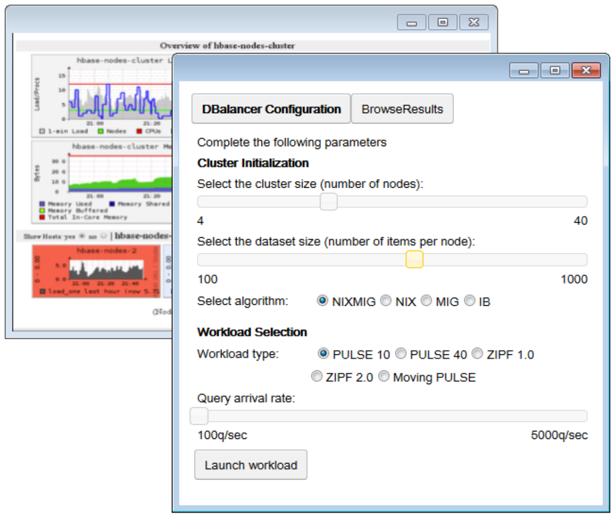
In the following section we give a quick overview of the suite of load balancing algorithms currently implemented on top of the *DBalancer*. We briefly present the *NIX*, *MIG*, *IB* and *NIXMIG* load balancing algorithms. For a thorough description, theoretical and experimental evaluation under skewed, dynamic and realistic workloads please refer to [15].

### 3.1 NIX and MIG

Balancing is performed by transferring items from overloaded peers to less loaded ones. We distinguish two cases:

**NIX:** In this case, balancing is performed between neighboring nodes, i.e., nodes with adjacent key ranges. The overloaded node transfers its largest keys to its forward neighbor or its smallest keys to its backward neighbor. The overloaded node's Coordinator contacts and reserves the other node, whereas the Mover module calculates the exact range of items to be transfered. The data-store Adaptor performs the specific keyvalue-dependent operations. A major disadvantage of *NIX* is that possibly many iterative such operations are needed in order to balance load inside large regions of loaded peers.

**MIG:** In this approach, an overloaded node utilizes the Prober module until it locates an available distant underloaded node. When this happens, the overloaded node's Coordinator instructs the distant nodes to depart from their place, join in the overloaded area and take a portion of its keys. While this operation seems more efficient, a large number of message exchanges is required for the remote node location (since the heavy Prober usage) and the overlay structure maintenance caused by the Overlay manager.


DBalancer Configuration | BrowseResults

Complete the following parameters

**Cluster Initialization**

Select the cluster size (number of nodes): 4 – 40

Select the dataset size (number of items per node): 100 – 1000

Select algorithm: [✓ NIXMIG] NIX MIG IB

**Workload Selection**

Workload type: [✓ PULSE 10] PULSE 40 ZIPF 1.0 ZIPF 2.0 Moving PULSE

Query arrival rate: 100q/sec – 5000q/sec

Launch workload

**Figure 2: Demo web interface.**

### 3.2 Item Balancing

In the work of Karger and Ruhl [14] (Item Balancing, hence *IB*) a *work-stealing* technique is applied: peers utilize the Prober to randomly contact distant nodes and compare their loads. The Coordinator's logic is as follows: if the load of the less loaded node is smaller than a fraction of $0 < \varepsilon < \frac{1}{4}$ of the more loaded node's load (i.e., $l_j \leq \varepsilon \leq l_i$ where $l_j$ is the less loaded node and $l_i$ the more loaded node) then a migration (*MIG*) or an neighboring item exchange (*NIX*) is performed.

### 3.3 NIXMIG

The goal of *NIXMIG* is to balance load by adaptively choosing to utilize either *NIX* or *MIG*. NIXMIG is using only local knowledge and identifies conditions where *MIG* is necessary to speed up the balancing process but is not excessively utilized.

*NIXMIG* is performed in three phases. In the first phase (Exam phase), the overloaded node examines the load status of a number of neighboring nodes and, if necessary, an additional number of distant nodes is contacted. The node examination is performed in a wave-like manner towards one direction of the structure, where each node contacts its successor. When the first phase is successful, then the algorithm proceeds to the *NIX* phase and portions of keys are iteratively transferred from one neighbor to another. Finally, the algorithm proceeds to the *MIG* phase, where the reserved underloaded nodes of the remote wave offload their keys to their neighbor and take a portion of the range of the final node of the *NIX* wave.

## 4. DEMONSTRATION DESCRIPTION

The demonstration will allow attendees to interact with the *DBalancer* on two levels: cluster configuration and workload setup. A comprehensive real-time GUI will enable users to enter the initialization parameters regarding cluster, algorithm and workload types, launch the experiment and watch in real time the system's behavior, as we can observe in Figure 2. In `http://youtu.be/B9HhN_kJntg` and in `http://youtu.be/F3iu7fj-IPc` we have uploaded some representative videos showing the *DBalancer*'s balancing procedure under different algorithms. Both real-time and previously collected performance metrics will be available in order to compare each setting's performance and cost. The interface consists of two parts described in detail:

**Cluster Initialization:** In this part, users can select to deploy *DBalancer* on top of various types and sizes of Cassandra clusters. Clusters ranging from 4 to 40 nodes will be deployed on a private OpenStack IaaS Cloud [8]. Users can also set the dataset size, by setting the NoSQL storage per node between 100K and 1000K YCSB objects (1KB each). Finally, users can select the load balancing algorithm that will be used by the *DBalancer* between NIXMIG, NIX, MIG, and IB. In order to avoid timely virtual cluster initializations, a number of pre-cooked virtual cluster setups with the respective datasets will be available.

**Workload Selection:** Users can select one of the following different workloads: a highly skewed 10% pulse workload, a less skewed 40% pulse workload, a highly skewed zipfian workload with $\theta = 2$ and a less skewed one ($\theta = 1$). We also allow a dynamic pulse workload which suddenly changes its skew in the middle of the execution. Finally, users are able to set the query arrival rate between 100 queries/sec up to 5K queries/sec. These workloads are created by utilizing YCSB [10], Yahoo's cloud serving benchmark tool.

The metrics that are going to be used in order to evaluate the scheme's performance are the number of overloaded peers, the number and type of balancing operations, the number of exchanged messages and items, the time-span that the method needs in order to minimize the overloaded servers, as well as some of the standard statistic measures that characterize the cluster as time advances: Average load, network traffic, mean query latency and throughput, etc. All these metrics will be available at real-time, with the use of the Ganglia [19], the de-facto cluster monitoring tool, as we can see in the back graph of Figure 2.

After each run, the participants will be able to browse both final aggregated results of the aforementioned metrics, along with graphs that show how these metrics were evolving during the balancing time. Moreover, videos that will depict the fluctuation of each server's load during time will be compiled and presented to the user. With the previous interface, users will have the opportunity to visually and quantitatively identify each algorithm's pros and cons, and to observe *NIXMIG*'s adaptivity in various workloads and network setups.

## 5. ACKNOWLEDGMENTS

This work has been funded by European Union and National Resources under the Hellenic General Secretariat for Research and Technology (GSRT) "COOPERATION 2009" National Action "09SYN-72-881" MODISSENSE Project.

## 6. REFERENCES

[1] `http://nosql-databases.org`.

[2] `http://www.pcworld.com/article/167435/jacksons_death_a_blow_to_the_internet.html`.

[3] `http://wiki.apache.org/cassandra/Operations#Load_balancing`.

[4] `https://github.com/voldemort/voldemort/wiki/Voldemort-Rebalancing`.

[5] `http://riak.basho.com/`.

[6] `http://docs.mongodb.org/manual/core/sharding-internals/#sharding-balancing-internals`.

[7] `http://issues.apache.org/jira/secure/attachment/12368261/RebalanceDesign6.pdf`.

[8] `http://www.openstack.org/`.

[9] G. Ananthanarayanan et al. Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters. In *EuroSys*, pages 287–300, 2011.

[10] B. Cooper et al. Benchmarking cloud serving systems with YCSB. In *SOCC*, pages 143–154, 2010.

[11] G. DeCandia et al. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP*, pages 205–220, 2007.

[12] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In *WWW*, pages 293–304. ACM, 2002.

[13] D. Karger et al. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC*, pages 654–663, 1997.

[14] D. R. Karger and M. Ruhl. Simple Efficient Load-Balancing Algorithms for Peer-to-Peer Systems. *Theory of Comp. Systems*, 39:787–804, 2006.

[15] I. Konstantinou, D. Tsoumakos, and N. Koziris. Fast and Cost-Effective Online Load-Balancing in Distributed Range-Queriable Systems. *IEEE Trans Parallel Distrib Syst*, 22(8):1350 –1364, August 2011.

[16] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions. In *ACM SoCC*, pages 75–86, 2010.

[17] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS*, 44:35–40, April 2010.

[18] F. Lu, S. Parkin, and G. Morgan. Load Balancing for Massively Multiplayer Online Games. In *NetGames*, 2006.

[19] M. Massie, B. Chun, and D. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7):817–840, 2004.

[20] I. Stoica et al. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *SIGCOMM*, pages 149–160, 2001.

[21] R. Sumbaly et al. Serving Large-Scale Batch Computed Data with Project Voldemort. In *FAST*, 2012.