MeT: Workload aware elasticity for NoSQL

Francisco Cruz Francisco Maia Miguel Matos Rui Oliveira
João Paulo José Pereira Ricardo Vilaça
HASLab - INESC TEC and U. Minho
[fmcruz,fmaia,miguelmatos,rco,jtpaulo,jop,rmvilaca]@di.uminho.pt

Abstract

NoSQL databases manage the bulk of data produced by modern Web applications such as social networks. This stems from their ability to partition and spread data to all available nodes, allowing NoSQL systems to scale. Unfortunately, current solutions' scale out is oblivious to the underlying data access patterns, resulting in both highly skewed load across nodes and suboptimal node configurations.

In this paper, we first show that judicious placement of HBase partitions taking into account data access patterns can improve overall throughput by 35%. Next, we go beyond current state of the art elastic systems limited to uninformed replica addition and removal by: i) reconfiguring existing replicas according to access patterns and ii) adding replicas specifically configured to the expected access pattern.

MeT is a prototype for a Cloud-enabled framework that can be used alone or in conjunction with OpenStack for the automatic and heterogeneous reconfiguration of a HBase deployment. Our evaluation, conducted using the YCSB workload generator and a TPC-C workload, shows that MeT is able to i) autonomously achieve the performance of a manual configured cluster and ii) quickly reconfigure the cluster according to unpredicted workload changes.

1. Introduction

Cloud Computing is the current trend in systems design and conception. The Cloud is a complex environment composed of various subsystems that, although different, are expected to exhibit a set of fundamental features: high availability, high performance and elasticity.

While high availability and high performance are common goals to all systems, elasticity is specific to the Cloud environment and closely tied to the pay-as-you go model. Elasticity can be defined as the ability of a system to grow or shrink its resource consumption according to demand. It is still an open challenge and a topic of a considerable amount of recent research [18, 24].

The ability to adjust resource consumption according to demand, favors the pay-as-you-go model and improves resource utilization. In addition, current Cloud providers make available their infrastructure (IaaS), platform (PaaS) or software (SaaS) to multiple customers in a multi-tenant environment. Therefore, optimal resource utilization becomes an even greater concern, since if one customer is using more resources than needed, it may impact the performance of other customer's applications, resulting in poorer overall performance. From a Cloud provider's perspective, the ability to dynamically optimize resource usage according to the contracted level of service is fundamental to the business model.

In this paper, we focus on the elasticity of a specific component: the data store, often referred to as NoSQL databases. These databases have been designed to take advantage of large resource pools and provide high availability and high performance. Moreover, these databases were designed to cope with resource availability changes. For instance, it is possible to add or remove database nodes from the cluster and to have the database handle such change transparently. Even though NoSQL databases can handle elasticity, they are not autonomously elastic: an external entity is required to control when and how to add or remove nodes.

Ideally, nodes would be added to the cluster when it is under heavy load, in order to maintain service levels, and removed in the opposite case, to reduce costs. Currently, these operations are mainly a manual task that motivated some recent research work striving for elasticity in NoSQL databases [13]. Briefly, the approach is to gather system-level metrics such as CPU usage, memory consumption and disk load, and then add or remove nodes from the cluster according to demand. This is an important step towards autonomous elasticity of NoSQL databases.

Simply adding and removing nodes is insufficient. In fact, current approaches consider that all nodes of a NoSQL cluster share identical, and thus homogeneous configurations. But in practice, different applications have different access patterns, which may even change over time. In addition,

Eurosys'13 April 15-17, 2013, Prague, Czech Republic

NoSQL databases assume data partitioning, meaning that even within an application there may exist data hotspots.

As our experiments show, fine tuning the available parameters of a NoSQL database on a per node basis, significantly boosts overall performance, specially when considering the workload characteristics. Consequently, the heterogeneity of data access patterns should be taken into account to optimize the use of available resources.

In this paper, we present the design and implementation of MET, an elastic system that not only adds and removes nodes, but also heterogeneously reconfigures them according to the observed workloads. We achieve this by leveraging on an existing IaaS system as the basic provider of elasticity. We expose new database engine metrics regarding workload's access patterns, which are constantly monitored along with the IaaS nodes. This information feeds our decision component that then performs online cluster reconfiguration as needed.

Contributions. We make three main contributions.
First, we propose a heterogeneous configuration of NoSQL databases and, using a standard benchmark for NoSQL databases, we show that it outperforms the default homogeneous setting. Second, we present the design of MET, an elastic system that not only adds and removes nodes, but also reconfigures them in a heterogeneous manner according to the workload's access patterns. Third, we validate MET's design, showing that it is able to autonomously achieve the performance of a manually configured heterogeneous cluster and also quickly reconfigures the cluster according to unpredicted workload changes.

Roadmap. The rest of this paper is organized as follows. In Section 2 we present a brief overview of NoSQL databases and, in particular of HBase. Next, in Section 3 we motivate our work showing how a heterogeneous configuration of HBase clearly outperforms the default homogeneous one. In Section 4 we describe MET's design, we describe its implementation in Section 5 and present the experimental evaluation in Section 6. Related work is analyzed in Section 7 and Section 8 concludes the paper.

2. Background

NoSQL databases run in a distributed setting with tenths to hundreds of nodes. The application data is partitioned and these data partitions are assigned to the available nodes according to a data placement strategy. This strategy is dependent on the specific NoSQL database used.

In the remainder of this paper we focus on HBase, which has a hierarchical architecture [25] and is one of the most successful and widely used NoSQL database[11]. Previous studies indicate that HBase is the best choice to handle elasticity [13].

2.1 HBase

Inspired by BigTable [6], HBase's data model implements a variant of the entity-attribute-value (EAV) model and can be thought of as a multi-dimensional sorted map. This map is called *HTable* and is indexed by the row key, the column name and a timestamp. A *HTable* may have an unbounded and dynamically created number of columns, which are grouped into *ColumnFamilies*. Data is maintained in lexicographic order by row key. HBase provides a key-value interface to manipulate data by means of put, get, delete, and scan operations. Write operations are atomic and immediately available to any subsequent read.

The row range of a *HTable* is horizontally partitioned into *Regions* and distributed over different nodes, named *RegionServers*. Data partitioning can be either manual or automatic. The automatic partitioning of a *HTable* occurs when it grows to a parametrized size by default 250MB. Moreover, the assignment of *Regions* to *RegionServers* recurs to a randomized data placement component. The strategy followed by this component is to evenly distribute the load of the cluster based on the number of *Regions*, i.e. ensuring every *RegionServer* has the same number of *Regions*.

Each *Region* is stored as an appendable file in the Hadoop Distributed File System (HDFS) [3], whose instances are called *DataNodes*. Usually, *RegionServers* are co-located with *DataNodes* to promote the locality of the data being served by the *RegionServer*. It is important to note, however, that when a cluster rebalacing is triggered, a *Region* may be assigned to a *RegionServer* not co-located with the *DataNode* responsible for that *Region*'s data, thus negatively impacting access performance. This problem can be reduced by means of data replication or by using an operation called *major_compact* that is automatically or manually triggered. After a cluster rebalancing, this operation presents the only way to restore data locality at the expense of merging all *Region's* files into a single new file. This new file is then stored in the co-located *DataNode*.

Parameters. Both HBase and HDFS have many configuration parameters[11]. For HBase, there are two configuration parameters that most significantly affect the performance of the cluster:

- *block cache size* - sets the amount of main memory available for caching blocks from *Regions*;
- *memstore size* - sets the amount of main memory available for updated data, before being flushed to disk.

These two parameters are expressed in terms of a percentage of the total *java heap space*[1] allocated to a *RegionServer* and allow to give more privilege to read or write operations, in a continuous fashion.

Other important configuration parameters include: the *block size* of the *block cache* and the *handler count*. The

[1] Their sum should not exceed 65% of the total *java heap space* [11]

former defaults to 64KB with lower values favoring random read operations. The latter controls the number of threads available to answer incoming requests and defaults to 10.

Other parameters allow to adjust the behavior of the garbage collector or the write buffer sizes. In addition, HBase allows for the use of compression algorithms that greatly reduce the disk I/O and network traffic between *RegionServers* and *DataNodes*.

3. Heterogenous performance analysis

In NoSQL databases, data is distributed across the cluster, thus each node is responsible for a subset of data. In clear contrast to relational databases (both single node and distributed), in NoSQL databases the co-location of data partitions in the same node, which are usually queried together, is no longer needed because:

- there is no clear relationship between data from different entities, and data is de-normalized;
- computation is done on the client side, for instance queries joining two data partitions do not take advantage of data co-location;
- NoSQL databases do not provide atomic multi-item operations, thus atomic inter-node operations are not a concern.

The fact that data is fairly unrelated and can be highly partitioned across the NoSQL cluster, allows for a rebalancing of the cluster to maximize performance without further concerns, such as data locality for join operations. In order to achieve it, NoSQL databases often require extensive fine tuning. These configuration tasks are typically manual and dependent on the administrator's expertise. Usually, the system administrator will analyze the expected workloads and homogeneously configure the cluster nodes to cope with the expected load with best performance. Such a configuration takes into account the overall cluster performance and each node in the cluster is configured identically. However, different applications have different data access patterns and even within the same application there may exist data partitions that are hotspots while others are seldom accessed.

The heterogeneity in access patterns should therefore be taken into account during distribution and data partitioning. Moreover, regardless of the application they refer to, data partitions with similar access patterns should be placed in the same physical nodes configured specifically and exclusively to serve them. The heterogeneity in access patterns leads to a heterogeneous cluster, i.e. with different node configurations, optimized to achieve better performance under the expected workloads. For instance, in HBase by increasing the *block cache size* (see Section 2) we can have one *RegionServer* optimized for read operations, and thus assign read intensive data partitions (or *Regions*) to that *RegionServer*. For clarity of presentation, we can refer to nodes as *RegionServers* or data partitions as *Regions*. This difference in nomenclature depends on whether we are referring to our algorithms that are independent of the implementation, or whether we are referring our prototype, which is based on HBase.

In the following we set up an experiment that validates our intuition and further motivates the rest of the paper.

3.1 Workload description

We evaluated HBase in a multi-tenant environment using YCSB [8] as a workload generator configured with different, but simultaneous workloads. The reason to use different workloads simultaneously, is to simulate a multi-tenant setting as expected in a Cloud environment. YCSB provides six pre-configured workloads that simulate different application scenarios. In order to achieve a overall read/write ratio of approximately 1.9:1 [7], we modified the configuration parameters of two workloads, namely of $WorkloadB$ and $WorkloadD$. We used the following workloads:

$WorkloadA$**:** readProportion=50%; updateProportion=50%; Application scenario: session store recording recent actions;

$WorkloadB$**:** updateProportion=100%; Application scenario: stocks management;

$WorkloadC$**:** readProportion=100%; Application scenario: user profile cache, where profiles are constructed elsewhere (e.g., Hadoop);

$WorkloadD$**:** readProportion=5%; insertProportion=95%; Application scenario: logging/history;

$WorkloadE$**:** scanProportion=95%; insertProportion=5%; Application scenario: threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id);

$WorkloadF$**:** readProportion=50%; readmodifywriteProportion=50%; Application scenario: user database, where user records are read and modified by the user or to record user activity.

All workloads were initially populated with 1,000,000 records, except $WorkloadD$. This workload simulates a logging/history application that produces a very fast growing log, thus it was initially populated with 100,000 records. Overall, the cluster starts with around 7GB of data and during a 30 minute run it grows, on average 6GB.

With the exception of $WorkloadD$ with only one data partition, each of the remaining workloads has four data partitions (*Regions* in HBase) of the same size. The keys were drawn from YCSB's *hotspot* distribution, with 50% of the requests accessing a subset of keys that account for 40% of the key space. In terms of the load distribution on each data partition, it means that one partition is a hotspot (34% of the requests), other partition has an intermediate load request (26%), and the remaining two have few but evenly distributed requests (20% of the requests each).

3.2 Experimental setting

In all experiments, one node acts as master for both HBase and HDFS, and it also holds a Zookeeper instance running in standalone mode. Our HBase cluster was composed of 5 *RegionServers*, each configured with a heap of 3 GB, and 5 *DataNodes*. It is noteworthy that the *RegionServers* were co-located with the *DataNodes* with a replication factor of 2.

We used two other nodes to run the YCSB workload generators: $WorkloadA$, $WorkloadB$ and $WorkloadC$ in one node, $WorkloadD$, $WorkloadE$ and $WorkloadF$ on the other. All workloads were configured to run for 30 minutes with a ramp-up time of 2 minutes. In addition, all workloads were run with 50 threads each except for $WorkloadD$ with 5 threads. Likewise, there were no limitations imposed on the throughput of each workload except for $WorkloadD$ with a target throughput of 1500 operations per second. We have imposed these limits to $WorkloadD$ so that all scenarios had identical conditions and were not, therefore, overly influenced by a too rapid growth of data. In our first experiments data grew so fast that far exceeded the capacity of our 5 *RegionServer* cluster, which would then negatively impact the performance of other workloads (multi-tenancy). This behavior was observed especially in the $Manual - Homogeneous$ strategy explained below.

All nodes used for these experiments have an Intel i3 CPU at 3.1GHz, with 4GB of memory and a local 7200 RPM SATA disk, and are interconnected by a switched Gigabit local area network.

3.3 Placement and configuration strategies

We defined three different strategies representative of different data placement and node configurations, namely: $Random - Homogeneous$, $Manual - Homogeneous$ and $Manual - Heterogeneous$.

Random-Homogeneous: This strategy represents the regular behavior of HBase with a manual, homogeneous configuration of nodes and using the out-of-the-box randomized data placement component that evenly distributes data partitions across all cluster nodes. Because it is random, it assumes uniformity on the number of requests per data partition. Besides the necessary optimization of the default configuration parameters of HBase, we also configured the two parameters that allocate a percentage of the available memory for read and write operations (*block cache size* and *memstore size*, respectively; see Section 2). We adopted a direct mapping between these two parameters and the overall read-/write ratio. That is, we assigned 60% of memory to the *block cache size* for read operations and, 40% to *memstore size* for write operations.

Manual-Homogeneous: In this strategy, we manually balanced data, so hot data partitions would be as dispersed as possible across all nodes. Furthermore, since configurations are homogeneous data partitions were distributed so that the number of read/write requests would be evenly balanced across all nodes. In order to do this, we conducted an exhaustive search to find the best distribution. That meant trying out all possible combinations of data partitions to nodes that balanced the number of read/write requests across all nodes. We evaluated 15 possible distributions and we chose the one that showed better throughput.

Note that this strategy represents one possible distribution that $Random - Homogeneous$ could achieve. The configuration parameters are the same as in $Random - Homogeneous$, so any performance improvement obtained is solely due to the data placement.

Manual-Heterogeneous: In order to take advantage of heterogeneity in access patterns, this strategy comprises manual data placement and heterogeneous node configuration. The objective of this strategy is to cluster data partitions with similar access patterns. In addition, each node is specifically configured according to the type of load it is expected to handle.

The first step to implement this strategy was to observe the workloads described earlier, in order to understand if and how we could cluster them according to their access patterns. Just by looking at the distribution of requests for each workload, one can easily conclude that $WorkloadA$ and $WorkloadF$ have a mix of read/write operations; $WorkloadC$ produces only read operations; $WorkloadE$ is mainly composed of scan operations; while $WorkloadB$ and $WorkloadD$ generate almost only write operations. These observations lead us to our first conclusion: we can aggregate the workloads into four main groups according to their access patterns, namely *Read/Write* mix, *Read*, *Scan* and *Write*.

The next step is related to the mapping of the data partitions to the *RegionServers* available. Intuitively, the number of *RegionServers* to assign each group should be proportional to the number of data partitions it contains. For instance, if we have a *Read* group containing 20 data partitions and a *Write* group containing only 5 data partitions, it is clear the number of *RegionServers* to assign to the *Read* group should be higher than to the *Write* group. Our experiments confirmed this intuition. Consequently, in the current context we used the following distribution: each of the groups considered were assigned one *RegionServer*, except for the *Read/Write* group. In fact, this group was assigned two *RegionServers*, because it contained 8 data partitions as opposed to the 4 or 5 data partitions of the other groups.

Once we have the mapping of groups to *RegionServers*, we distribute data partitions following an approach similar to $Manual - Homogeneous$. In other words, for the data partitions belonging to the *Read/Write* group we balanced data so each of the two *RegionServers* had a similar load (i.e. similar number of requests). Once more, we recurred to an exhaustive search that culminated with the hotspots of each workload being in different *RegionServers*, and with the same number of data partitions in each *RegionServer* (i.e. 4 data partitions in each one).

After the data placement stage was completed, we then manually configured each *RegionServer* taking into account the load they were expected to handle. For instance, all data partitions belonging to $WorkloadE$ were assigned to a single node with tailored configuration, namely increased *block size* (better for sequential reads) and almost all available memory set for a read workload with only marginal space for writes. On the contrary, the *RegionServer* of $WorkloadB$ and $WorkloadD$ was configured for a write workload.

3.4 Results

Figure 1 shows the throughput for all workloads under the different HBase strategies detailed above. Each bar in the plot represents a specific observation in the cumulative distributed function (CDF) of the results, for instance the medium shade of gray (50^{th} percentile) indicates half of the observations were below that value and the other half above. All presented results are the consequence of 5 runs.

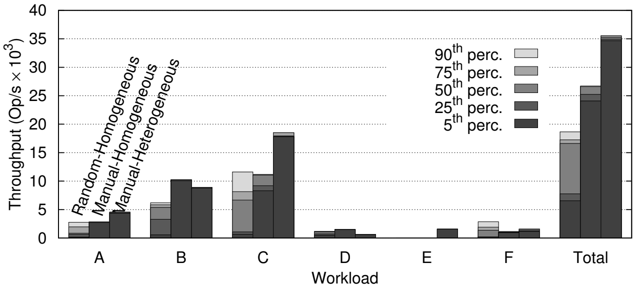


Figure 1. Manual strategies results.

It is clear that the $Manual - *$ strategies impact positively the overall cluster performance. While the $Manual - *$ strategies improve to some extent the throughput of $WorkloadA$, $WorkloadB$ and $WorkloadE$, most of the observed improvement is due to the performance of $WorkloadC$.

The variance observed in the $Random - Homogeneous$ strategy, both in each workload individually and in the total throughput is very high due to the randomness of the data placement component. As it is possible to observe, there was one run whose total throughput was close to $Manual - Homogeneous$'s result, while in another run the total throughput is almost half of the mean. This first comparison confirms that a random data placement, when the distribution of requests is not uniform, may lead to very distinct results. As such, we need to carefully distribute data partitions across the cluster when dealing with a non-uniform distribution of requests. Otherwise, the performance of the cluster is left to chance.

Thereby, with a different strategy on data placement and, accordingly, configuring the nodes for the expected load the results achieved by the $Manual - Heterogeneous$ strategy outperforms the two other strategies. As opposed to the $Manual - Homogeneous$, $Manual - Heterogeneous$ strategy improves each workload in relation to the other two strategies, except marginally for $WorkloadD$. At first glance it may seem that $WorkloadF$'s performance is better under the $Random - Homogeneous$ strategy. This is not true, since on average $WorkloadF$'s performance for the $Random - Homogeneous$ strategy is somewhat lower than under $Manual - Heterogeneous$. Nonetheless, due to the randomness of the data placement component there was at least one run (90th percentile) whose throughput was higher than $Manual - Heterogeneous$ strategy.

Regarding the total throughput, $Manual - Heterogeneous$ more than doubles the result achieved by strategy $Random - Homogeneous$, and in relation to $Manual - Homogeneous$ it improves the result by 35% on average. It is important to stress that for $WorkloadE$ (majority of scan operations) the improvement is remarkable: from around 100 scans per second, to around 1350 scans per second.

3.5 Discussion

From the analysis of these results it is possible to see that a heterogeneous HBase cluster can outperform the default configuration, supporting our initial claims. Even when using a judicious data placement, but still with homogeneous nodes, the results are worse the $Manual - Heterogeneous$. Specifically, NoSQL nodes should not be treated as homogeneous entities because it often results in a skewed load on cluster nodes leading to both poor resource usage, due to idle nodes, and degraded performance, due to overloaded nodes. These observations motivate our belief that it is not sufficient to simply add or remove HBase nodes in order to have an effective elastic database. Instead, it is necessary to take into account the database workload and adapt the cluster accordingly. Unfortunately, combining heterogeneous node configurations with resource allocation and data placement is a difficult and error prone task, thus should be automated.

In the rest of the paper, we detail the design and implementation of a mechanism that is able to autonomously achieve performance results similar to the heterogeneous configuration and manual data placement, without human intervention.

4. MeT Framework

The heterogeneous configuration of a HBase cluster has proven to achieve much better performance than the alternatives. The downside being it greatly increases the complexity of cluster management. In fact, if the number of nodes and data partitions increases to the magnitude of hundreds or thousands, the manual heterogenous configuration of a cluster is impracticable.

As a result, we developed MeT: a cloud-enabled framework that can automatically manage, configure and reconfigure a cluster in a heterogeneous fashion, according to its access patterns. Furthermore, MeT equips the underlying

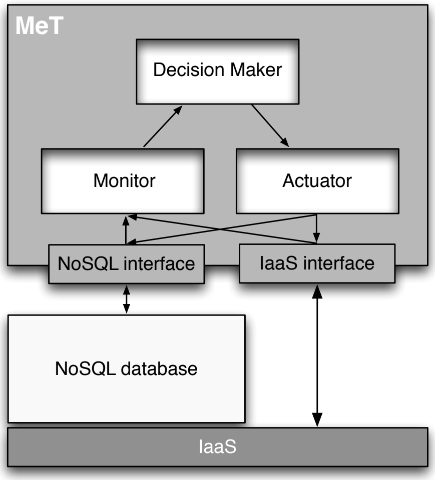


Figure 2. MET's architecture.

NoSQL database with the ability to be autonomously elastic, by the addition or removal of nodes specifically configured to the load they are expected to serve.

Figure 2 depicts MET's design that relies on three main components: *Monitor*, *Decision Maker* and *Actuator*. The *Monitor* and *Actuator* components can interface with a NoSQL database directly (through the NoSQL interface) and with an *IaaS* (through the IaaS interface). The *Decision Maker* interacts with the *Monitor* and *Actuator* components.

Giving a brief overview, the *Monitor* component gathers important statistics of the running cluster and periodically passes them to the *Decision Maker* component. This component can be considered the core of MET. Basically, it tries to converge to the same results as the $Manual - Heterogenous$ strategy (see Section 3) in an automated way. In that regard it follows a set of stages. The first step involves deciding wether the load cluster is acceptable based on the metrics delivered by the *Monitor*. If the cluster is overloaded or underloaded, then it is decided how many nodes must be added or removed from the cluster, respectively. Closely matching the process described in $Manual - Heterogenous$ strategy, this component then classifies each data partition by type of access, clusters them into groups, and proportionally determines the number of nodes to attribute each group. After that, for each group a *greedy algorithm* tries to evenly balance the load in each node. Finally, the *Decision Maker* tries to convey the best way to bring the previously configured cluster to the new computed configuration. This final output is then passed to the *Actuator* that actually implements it in the running cluster. In the following subsections we will describe in detail each component and the algorithms used.

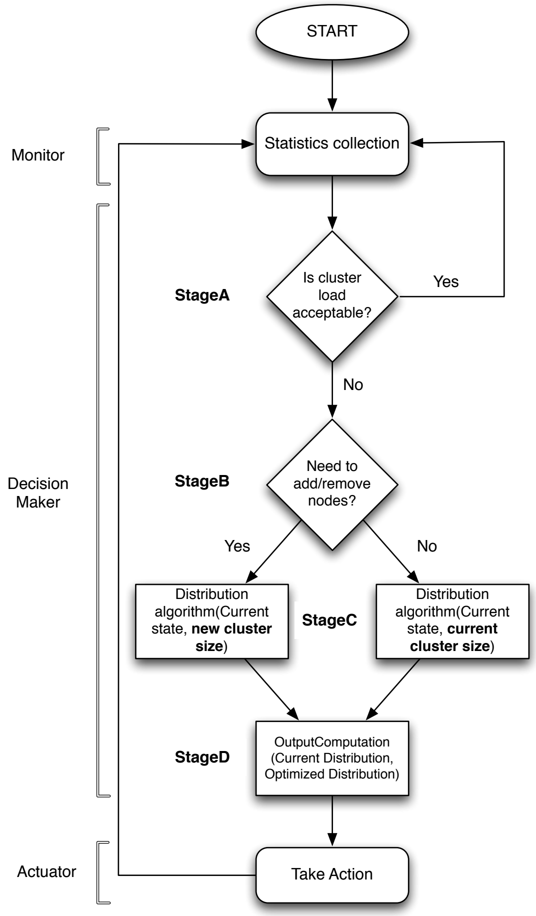

START
Monitor
Statistics collection
StageA
Is cluster load acceptable?
Yes
No
Decision Maker
StageB
Need to add/remove nodes?
Yes
No
Distribution algorithm(Current state, new cluster size)
StageC
Distribution algorithm(Current state, current cluster size)
StageD
OutputComputation (Current Distribution, Optimized Distribution)
Actuator
Take Action

Figure 3. MET's flow chart with particular emphasis on the *Decision Maker* component.

4.1 Monitor

The *Monitor* component gathers information about the current state of the cluster (Figure 3). Periodically, it collects and maintains data over several cluster metrics at two different levels: system metrics and metrics specific to the NoSQL database. System metrics are CPU utilization, I/O wait and memory usage. With regard to NoSQL specific metrics, this component needs to keep track of several metrics per node and per data partition. The metrics collected from the NoSQL database must be enough to know the access patterns of the workload. MET uses the total number of read, write and scan requests as well as each node locality index. In this regard, the locality index measures the percentage of data that is locally accessible at each node. In other words, it measures the amount of data owned by the node that is locally stored thus not requiring to be fetched through the network when queried.

In order to account for temporary load spikes that could result in poor decisions, we used exponential smoothing [5] coupled with storing only the observations after each *Actu-*

ator's action. For each monitoring interval, the last observation is the most important, exponentially decreasing in importance till the first observation. Periodically, all retrieved metrics are delivered to the *Decision Maker* component.

4.2 Decision Maker

The *Decision Maker* component is responsible for deciding what actions to take when the cluster is considered to be in a sub-optimal state. As depicted in Figure 3 it works following four different stages.

4.2.1 Determine the current state of the cluster (StageA)

StageA (Figure 3) begins with the periodical delivery by the *Monitor* component of gathered statistics about the current state of the cluster. Based on those statistics, the *Decision Maker* has to decide whether the load of each node in the cluster is acceptable or not. By acceptable, we mean that the system metrics provided are within certain defined thresholds. Example values for these thresholds are evaluated in subsequent sections.

If the cluster is healthy, the *Decision Maker* remains in StageA (Yes branch of StageA). Otherwise, three data structures are populated to be used in StageB that is immediately initiated. Such data structures are: i) $firstTime$ variable that states whether it is the first time StageB is going to run or not; ii) $subOptimalNodes$ variable which represents the percentage of nodes in a sub-optimal state; iii) $remove$ variable that states whether the cluster is under or overloaded.

4.2.2 Decision algorithm for adding and removing nodes (StageB)

In StageB (Figure 3), the main task is deciding if it is necessary to add or remove database nodes from the cluster, and if so how many of them following Algorithm 1.

A particular case arises if it is the first time StageB is running ($firstTime$ input). If this is the case, MeT distributes data partitions and heterogeneously configures the current cluster from scratch in what we call an *InitialReconfiguration*. This only happens once.

In subsequent iterations, if the cluster is still in a sub-optimal state, we decide to add or remove nodes. Because we are unable to determine a priori how many nodes we need, those nodes are iteratively added in a quadratic fashion and removed linearly. A quadratic strategy enables a fast response to demand increase by allowing to reach a sufficient number of nodes on a logarithmic number of iterations. That is, the algorithm starts by suggesting the addition of 1 node, and in the following iterations, 2, 4, 8 nodes and so forth, until the load in the cluster is acceptable. Conversely, it removes only 1 node in each iteration, also until the load in the cluster is acceptable. Of course, this strategy may incur a higher provision of temporary resources than necessary. For example, supposing that there is the need for the addition of 8 new nodes. We would only need 4 iterations to reach a sufficient number nodes, but we would have added 15 nodes in the process. In the meantime, if there was not another increase in demand, we would need 7 more iterations to linearly remove nodes until the desired 8 nodes with a total of 11 iterations. On the contrary, if we were adding nodes linearly we would be needing 8 iterations to achieve the desired cluster size. This means that it would take twice as long to reach a point where the number of nodes would be enough to handle the load (from 4 iterations to 8). On the other hand, this also means that we need 3 more iterations to shrink the cluster to the needed size. By using this quadratic strategy we privilege availability and a fast response to sudden load increase.

Algorithm 1: Decision Algorithm to add or remove nodes

```
Data: $nodesToChange \leftarrow 1$
Input: $subOptimalNodes, firstTime, remove$
Result: $result$
/* number of nodes to be added or removed from the
   cluster                                              */
begin
    if $subOptimalNodes > SubOptimalNodesThreshold$
    then
        $result \leftarrow nodesToChange$
        $nodesToChange \leftarrow nodesToChange * 2$
    else
        if $firstTime$ then
            $result \leftarrow 0$
            // $InitialReconfiguration$
        else
            if $remove$ then
                $result \leftarrow -1$
                $nodesToChange \leftarrow 1$
            else
                $result \leftarrow nodesToChange$
                $nodesToChange \leftarrow nodesToChange * 2$
    return $result$
end
```

It should be noted however, that from our experience in the case it is the first time the algorithm is invoked, but the number of sub-optimal nodes is already more than $SubOptimalNodesThreshold$ we proceed straightaway to the addition of nodes. This threshold is a MeT parameter and should be configured according to each system characteristics.

Finally, the *Decision Algorithm* computes the number of nodes to be added or removed from the cluster, and passes it to the *Distribution Algorithm* in the form of a target cluster size. If there are nodes to be added or removed a new cluster size is computed and passed as a parameter to the next stage. If not, the current size of the cluster is passed as a parameter.

4.2.3 Distribution algorithm (StageC)

The *Distribution Algorithm* corresponds to StageC of the *Decision Maker*'s component (Figure 3) and is in fact divided in three parts: *classification*; *node grouping*; and *as-*

signment. Note that this stage is only reached if the cluster is in sub-optimal state. Even if StageB's result states that there is no need to add or remove nodes, the fact that StageC is running means that a cluster reconfiguration should be attempted in order to improve cluster health.

Classification: data partitions are divided into groups according to access patterns. As stated earlier, we defined 4 groups: read, write, read/write and scan (see Section 3.3). Using the metrics related to the number of write, read and scan requests of each data partition, the *Classification* algorithm assigns each data partition to one of the 4 groups. The assignment of partitions to groups is parameterized with threshold values. In MET, such values have been obtained by experimental observation and are presented in Section 5. In order to accommodate workload changes, metric values obtained for each data partition are refreshed at the beginning of every monitoring interval.

Grouping: the number of nodes to attribute to each group is computed. Each group will be assigned a number of nodes equal to the division of the number of partitions in that group by the total number of partitions, and then multiplied by the total number of nodes available. More formally:

$$\forall g \in G : \frac{\#partitions\ in\ g}{total\#partitions} * total\#nodes$$

Assigment: from node grouping and data partition classification an assigment of data partitions to nodes is established. The assignment is done attempting to balance the load and the number of data partitions in each node. This task falls in a classical problem called *makespan minimization* or *multiprocessor scheduling*, which in turn is related to bin-packing problems. These class of problems are known to be NP-hard [14] but there are greedy algorithms that provide good results in polynomial time. As a result, we used the greedy algorithm proposed in [12], and because we know in advance all data partitions we could use the variant of this algorithm that provides better results - *Longest Processing Time (LPT)*. In short, the *makespan minimization* problem can be defined as: there is a set of parallel processors and a set of jobs with a determined cost; in the LPT version, the algorithm first sorts the set of jobs by decreasing cost and then assigns the largest job to the least loaded processor, until there are no jobs left to assign. In our case, jobs can be translated to data partitions, parallel processors to nodes and the cost of each job to the number of requests of each data partition.

Furthermore, we added a new constraint to the problem to also attempt to balance the number of data partitions assigned to each node. In this regard, the algorithm takes into account node capacity and establishes a maximum number of data partitions per node. This maximum value is estimated by dividing the number of data partitions in the group by the number of nodes in the group.

Algorithm 2: Assignment Algorithm

```
Data: $result \leftarrow []$
Input: $nodeGroup, dataPartitions, max$
/* max stores maximum number of partitions per
   node. Calculated in order to balance load.     */
Result: result
begin
    Data: $dataPartitions.sort()$
    /* Sort by number of requests in decreasing
       order.                                      */
    while $dataPartitions.size() > 0$ do
        $partition \leftarrow dataPartitions.first()$
        $node \leftarrow nodeGroup.getMostEmptyNode()$
        if $node.numberOfPartitions < max$ then
            $node.assign(partition)$
            $dataPartitions.remove(partition)$
        else
            $result.add(node)$
            $nodeGroup.markAsFull(node)$
            /* Node already full.                  */
    return $result;$
end
```

The assignment algorithm is depicted in Algorithm 2. It should be noted that it has to be called for each group of data partitions.

4.2.4 Output Computation (StageD)

Finally, StageD has the responsibility of determining the *best* way to achieve the targeted cluster configuration. By *best* we mean the one that minimizes node reconfiguration and data partition moves. As depicted in Algorithm 3, it receives as input the current cluster distribution and the distribution suggested by the *Assignment Algorithm*. The first time this algorithm runs, it has no information about the current configuration. At the beginning, we consider that the cluster is homogeneously configured. Thus, the distribution suggestion is passed on to the *Actuator*. This results in an initially heavier full cluster reconfiguration (*InitialReconfiguration*).

In subsequent runs, the algorithm looks at the current distribution of data partitions per node and tries to match it with the new distribution. The process of matching distributions is made recurring to a set intersection algorithm between sets of partitions. In MET, the set intersection algorithm is a best effort one. For each set of partitions from the suggested configuration, it tries to find the node that currently holds the most similar set of partitions. The result is an assignment of nodes to configurations and sets of partitions to hold.

If there are new nodes added to the cluster, a set of partitions and a configuration type is assigned to these nodes. The same way, if the targeted configuration does not fully match the current cluster configuration, new sets of partitions and configuration types are assigned to existing nodes. The output of this algorithm is a cluster distribution that minimizes data partitions' reassignment and nodes' reconfiguration.

Algorithm 3: Output Computation

```
Data: $result \leftarrow []$
Input: $currentState, optimalState, firstTime$
/* Lists of nodes and correspondent sets of data
   partitions.                                  */
Result: result
begin
    if $firstTime$ then
        $result \leftarrow optimalState$
    else
        foreach $node \in currentState.nodes()$ do
            $type \leftarrow node.type()$
            $set \leftarrow node.partitionSet()$
            $opset \leftarrow optimalState.mostSimilar(set, type)$
            $optimalState.remove(opset)$
            $result.add((node, opset, type))$
        if $optimalState \neq \emptyset$ then
            foreach $node \in currentState.node()$ do
                $type \leftarrow node.type()$
                $opset \leftarrow optimalState.popPartitionSet()$
                $result.add((node, opset, type))$
    return $result;$
end
```

4.3 Actuator

The *Actuator* component carries out the necessary tasks to implement the distribution given by the *Decision Maker*. It is responsible for the actual addition and removal of database nodes. On the one hand, if we are using a *IaaS* system it means first starting a virtual machine, and only after the NoSQL database. On the other hand, if we are using the NoSQL database directly it has only to start or shutdown the respective processes. The *Actuator* is also responsible for the individual reconfiguration of nodes according to one of the possible four groups defined above. In addition, it assigns the data partitions to those nodes as determined by the *Decision Maker*.

5. Implementation

MeT is available as an open source project.[2] In the current prototype we used HBase as the NoSQL database and OpenStack as the *IaaS* platform. OpenStack has gained wide support both from the community and enterprises, and is maturing very quickly [17].

At the implementation level MeT has two main parts. It is composed by a Java module and a Python module. The pivotal module is written in Python and comprises the core of the *Decision Maker*, *Monitor* and *Actuator* components of MeT. The Java module is used to gather HBase statistics through the *HBase Administrator* interface within the *Monitor* module of MeT.

Monitoring: The *Monitor* component gathers data about CPU usage, memory usage and I/O wait of the various nodes through Ganglia [16]. Regarding the metrics specific of HBase, we collect them through JMX from each *RegionServer*, namely: the total number of read, write and scan requests; the number of requests per second; and an index that measures the data locality of the blocks in the co-located *DataNode*. It also retrieves some metrics of each data partition like the number of read, write and scan requests. The number of scan requests is not available in HBase thus we modified it to calculate and export this metric. All this data is retrieved by MeT's Java module, which interfaces with the Python module through Py4J [10]. The monitoring intervals are configurable. It is possible to define Ganglia requests periodicity and data history size. Similar to *Decision Maker* parameters, these are also defined in a properties file.

Decision Maker parameters: In order for the *Decision Maker* to work, some parameters must be set. Firstly, the classification task of Section 4.2.3 requires a set of threshold values to define types of partitions. Four groups were defined. Data partitions are classified according to the following criteria: i) read, if more than 60% of total requests are read requests; ii) write, if more than 60% of total requests are write requests; iii) scan, if more than 60% of read requests are scan requests; iv) and read-write in every other case. Secondly, $SubOptimalNodesThreshold$ must be configured. In our experiments this threshold was set to 50% of the cluster. This means that if half of the cluster is under heavy load MeT will proceed straightway to the addition of a new node. From our experience, this parameter should be set to 50%, because when most of the nodes in the cluster are under heavy load there is no benefit in the reconfiguration of the cluster without adding new nodes. Moreover, if the cluster in question is subjected to very sudden peak loads it should be adjusted to less than 50% for a faster response to increased demand.

Although we do not envisage that *classification* parameter values can take different values, this may not be the case for other parameters. Consequently, each one of these parameters is configurable in a properties file.

Taking actions: Addition and removal of virtual machines from the HBase cluster is done through the OpenStack interface by the *Actuator*. With regard to node reconfiguration, HBase does not currently provide a mechanism to allow online reconfiguration of a *RegionServer*. That means that every reconfiguration of a *RegionServer* implies its restart. As a result, a full reconfiguration of the cluster is a very costly operation. Bringing the whole cluster down for a full reconfiguration would reduce the amount of time needed for the full reconfiguration, but it would also mean that during that period, all data would be unavailable. Therefore, we use a strategy to incrementally reconfigure the *RegionServers* while maintaining data availability, although with a lower overall throughput. This strategy redistributes the *Regions* from the *RegionServer* that is going to be reconfigured across the remaining nodes that have not been reconfigured yet. Then, when there are no *Regions* left in the *Region-*

[2] `https://github.com/fmaia/MeT`

Server, it is restarted with the new configuration. Finally, the *Regions* determined by *Decision Maker* are assigned to it. If data locality is below 70% for *RegionServers* configured for a write workload and 90% for all the others, it invokes the *major_compact* operation (as described in Section 2) in order to reestablish data locality. The difference between the two values is that data locality is of more relevance to a read intensive workload and a *major_compact* operation is a costly one. Relaxing the condition for write intensive workloads has the objective of minimizing the load these operations impose on the system. This process is repeated for all *RegionServer*'s reconfigurations.

The concrete values used in our evaluation have been chosen based on experimental observation and our own experience. The individual study of all parameters is left out of the scope of the present paper.

6. Evaluation

This section evaluates MeT from three perspectives. First we assess if MeT is able to autonomously converge to a performance level comparable to that achieved by a *Manual-Homogeneous* configuration of an HBase cluster. In this first step an YCBS workload is used. Secondly, we evaluate MeT's versatility by exposing MeT to a PyTPCC workload without any kind of customization. Finally, we study MeT's elastic properties in a Cloud environment.

6.1 Configuration

In the experiments below, every 30 seconds the *Monitor* component gathers the metrics and sends them to the *Decision Maker* every 3 minutes. The period of 30 seconds is the same used by other approaches [13], but the *Decision Maker* is only invoked after having 6 samples to minimize the impact of sudden spikes and take advantage of the exponential smoothing algorithm. The HBase configuration parameters for each group (*Distribution Algorithm* of Section 4) are described in Table 1.

Node profile	Cache size	Memstore size	Block size
Read	55%	10%	32KB
Write	10%	55%	64KB
Read/Write	45%	20%	32KB
Scan	55%	10%	128KB

Table 1. Node configuration profiles.

6.2 Convergence

We started by accessing if MeT could autonomously achieve similar performance to a manual heterogeneous cluster configuration ($Manual-Heterogenous$ strategy). The experimental setting is the same of Section 3. We used the 6 YCSB workloads described in such section. Then, we configured a HBase cluster with optimized configuration parameters, homogeneous nodes and using the out-of-the-box randomized data placement component ($Random-Homogeneous$ strategy from Section 3).

After 2 minutes of ramp-up time, we start MeT. The experiment then runs for 30 minutes logging the throughput from the perspective of the YCSB's clients.

We then compared the results with runs without MeT for the HBase cluster configured with strategies $Manual-Homogeneous$ and $Manual-Heterogeneous$. We picked the run with the best throughput from both strategies from the results presented in Section 3. These results are depicted in Figure 4.

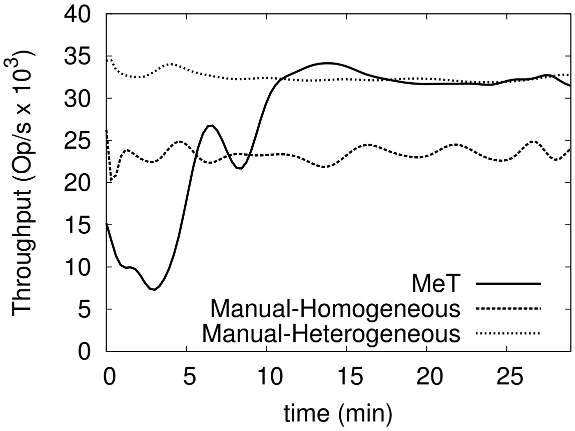


Figure 4. Evaluation results

This experiment shows that MeT behaves as expected. It is capable of reconfiguring the HBase cluster *on-the-fly* and achieve similar performance to that of a manually configured cluster. Note that for the same cluster and workload, MeT achieves a significant performance increase: it fully reconfigures a HBase cluster (initially configured with the $Random - Homogeneous$ strategy) in order to achieve a distribution of data partitions and node's configuration identical to the $Manual - Heterogeneous$ strategy.

The cost of reconfiguration is observable between the 2nd and 8th minute of the experiment (6 minutes). From this overall time, the time taken by target cluster reconfiguration calculations and data mapping is negligible. Restarting the *RegionServers* along with *major_compacts* are the time consuming operations. On the one hand, in our setting a *major_compact* takes roughly 1 minute/GB. On the other hand, most of the impact of reconfiguration on the observed throughput is due to need to restart *RegionServers*, because currently HBase does not allow online reconfigurations. Such feature would allow to greatly decrease this impact. However, by incrementally reconfiguring each *RegionServer* we not only provide continuous data availability, but we also provide reasonable performance with a minimum throughput of 7,500 operations per second. Then, the throughput quickly rises to 20,000 operations per second by the 5th minute and maintains this level of throughput until the reconfiguration is completed by the 8th minute. From this point, the performance is identical to the $Manual - Heterogeneous$ strategy. Even taking into account the reconfiguration cost, within less than 15 minutes the cumulated average throughput using MeT is greater than the default HBase with the $Manual - Homogeneous$ data placement strategy carefully defined by the administrator. These results allow us to state that MeT is able to autonomously reconfigure a running cluster, converging to a cluster configuration and performance level similar to that of a manually configured one.

6.3 Versatility

The goal of this experiment is to assess whether MeT could achieve similar results when using a significantly different workload. Moreover, without any change to MeT or its configuration parameters and without any previous knowledge about the workload itself.

For this purpose, we chose PyTPCC[3] an optimized implementation for HBase of the standard OLTP benchmark TPC-C. Note that, while TPC-C standard transactions are expected to have full ACID semantics this implementation offers the isolation semantics provided by HBase: record level atomicity.

TPC-C benchmark attempts to reproduce the behavior of any business in which sales' districts are geographically distributed along with the corresponding warehouses. There are a total of 9 tables and 5 different types of transactions, and the results are measured in transactions per minute (tpmCs). The default traffic is a mixture of 8% read-only and 92% update transactions and thus is a write intensive benchmark.

The TPC-C database was populated with 30 warehouses resulting in a database of 15GB. TPC-C tables were horizontally partitioned following the usual setting for running TPC-C in distributed databases [21]. In that sense, in our experimental setting there were 5 warehouses per *RegionServer*. Each *RegionServer* handles a total of 50 clients.

We ran this experiment on a HBase cluster of 6 *RegionServers*, each configured with a heap of 3 GB, and co-located with 6 *DataNodes*. Similarly to the previous experiment, we used another machine as the master of both HBase and HDFS as well as the Zookeeper instance. PyTPCC's clients were deployed in three other machines amounting to 300 clients (100 client threads per machine), and configured to run for 45 minutes.

This experiment involved three settings: i) a run with a $Manual - Homogeneous$ configuration; ii) MeT starting with a $Manual - Homogeneous$ configuration; iii) and an entire run with the configuration suggested by MeT. The first serves as a baseline and represents the usual way TPC-C runs. It was obtained experimentally, selecting the one that offered the best overall throughput (tpmC), as follows: 50% for the *cache size*; 15% for the *memstore size*; and 32KB of block size. The second setting begins with the same configuration as the first one and after 4 minutes we start MeT to reconfigure the cluster. In the third setting, we used the same distribution and configuration suggested by MeT, but the benchmark was allowed to run for the full 45 minutes without any reconfiguration. Therefore, it represents the maximum throughput that MeT's configuration could possibly achieve.

Setting	Throughput (tpmC)
i) $Manual - Homogeneous$	25380
ii) MeT with reconfiguration overhead	31020
iii) MeT w/o reconfiguration overhead	33720

Table 2. PyTPCC average throughput results.

The results, depicted in Table 2, are consistent with those of YCSB i.e. the heterogeneous setting improves the throughput of the $Manual - Homogeneous$ one by 33%. In addition, when comparing the results achieved by MeT and the third setting, the cost of reconfiguration during the experiment is not significant. In fact, around 10 minutes of the total 45 minutes (that is 23%) are due to the phase of ramp-up time (4 minutes) and the initial reconfiguration phase (6 minutes). Nonetheless, the overall difference between both settings is just 8%.

The results obtained in this experiment show that MeT is versatile and is able to achieve good results even in the presence of substantially different workloads and without any type of previous knowledge about them.

6.4 Elasticity

The experiments conducted so far show that an informed (workload-aware) and heterogeneous configuration of a HBase cluster leads to the best performance. Moreover, MeT is able to autonomously infer and apply such cluster configuration yielding a performance similar to a manually obtained configuration.

In these experiments we go a step further and use MeT as an elastic resource manager that adjusts the size of the cluster according to utilization. To this end, we ran a HBase cluster and MeT on top of an OpenStack deployment. Moreover, we compare MeT's behavior and performance with an existing system called *tiramola* [13]. This system, like Amazon's Cloud Watch [1] together with Amazon's Auto Scaling [2], automatically provides elasticity to NoSQL databases based on a set of system metrics defined by the client/user of the system. When those metrics reach a threshold, a new node is either launched or retracted from the cluster. Meaning, they are oblivious to the underlying NoSQL system: they just add/remove nodes from the cluster, they do not reconfigure nodes, neither they make data load balancing, nor migrate any data from node to node. We compare MeT with *tiramola* because is the only freely available system.

[3] https://github.com/apavlo/py-tpcc/wiki/HBase-Driver

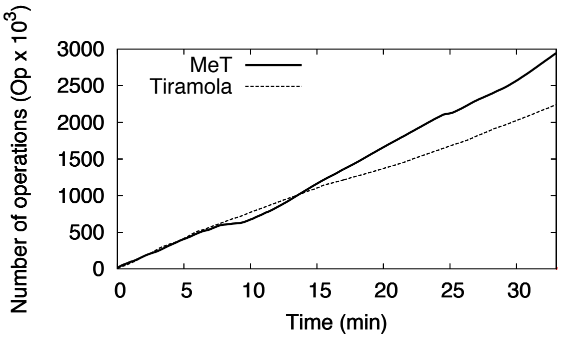


Figure 5. Cumulative throughput of MET and *tiramola* in the first phase of the experiment.

For this experiment, the HBase cluster is initially configured with seven virtual machines with 3GB of RAM: one for the HBase *Master*, the Hadoop *Namenode* and the Zookeeper in standalone mode; the remaining six for *RegionServers* co-located with *Datanodes*. In every run the initial state is identical: 100% data locality; a replication factor of 2; and data partitions manually balanced on a homogeneous configuration of the cluster.

In this experiment, we provided each system with a set of YCSB workloads that overloads the initial system. The experiment ran for approximately 60 minutes and was divided in two phases. In the first phase (33 minutes) all clients were active and we observed the throughput and the number of nodes in the cluster.

Figure 5 shows the cumulative throughput achieved in both scenarios. As it is observable, the HBase cluster managed by MET outperforms the one managed by *tiramola*. By the end of the first phase MET has completed more 706,000 operations than *tiramola*, corresponding to a 31% throughput increase. Note that these results are obtained despite the initial MET reconfiguration cost (from 4th to 11th minute), which starts to pay off after around minute 12. Equally important in a Cloud environment is the amount of resources required to achieve such throughput. This is depicted in Figure 6 that shows the throughput evolution (left YY axis) and the number of machines in each cluster (right YY axis).

MET's throughput is not only superior to *tiramola* but the number of machines is less, requiring 9 machines against 11. Also note that the peak performance achieved by MET actually corresponds to this scenario maximum achievable throughput of 22,000 operations/second where all YCSB clients are saturated.

Interestingly, even though *tiramola* adds more machines to the cluster there is no significant increase in throughput until the 20th minute. This stems from the random balancing and the degradation of data locality, which are precisely addressed by MET. MET judiciously balances the cluster and periodically performs a *major compact* for the regions loosing locality and the heterogeneous configuration achieved by MET increases the cluster throughput by configuring each *RegionServer* accordingly to the workload.

In the second phase of the experiment, we study the systems under resources underutilization. After the 33th minute we progressively switched-off some of the YCSB workloads until there was only one workload active. At minute 33 we turned off $WorkloadE$ and $WorkloadF$, then at minute 43 $WorkloadB$, and finally at minute 53 $WorkloadA$ leaving only $WorkloadC$ running. The experiment results are depicted in Figure 6 and workload removal coincides with the vertical lines in the Figure.

As can be observed, MET quickly detects the lower demand and removes one node from the system. With the progressive lower demand, this process is repeated until the number of nodes is equal to the initial cluster. Please note that, in this experiment we are allowing MET to release machines each time it detects underutilization, but such behavior is parameterized to avoid, for instance, continuous addition and removal of machines.

On the contrary, *tiramola* only releases resources when every node in the cluster is underutilized. This cannot be parametrized and is due to the homogeneous nature of the *tiramola* managed cluster where removing a single node can divert the load to other already overloaded nodes. The differences in throughput between both systems are due to this behavior, because while MET is terminating one node and reconfiguring, *tiramola* is just receiving less requests.

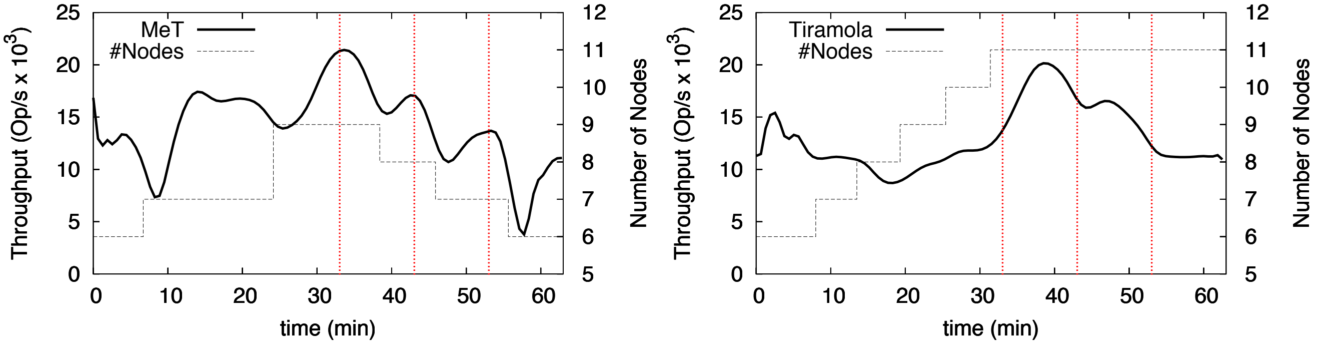
7. Related Work

This work is related with a wide range of research work. Namely, related with dynamic scale of Cloud applications. A good range of this related work is present in [24] where many of the current state of the art efforts towards an elastic Cloud are referred. In our paper however, we focus on automated elasticity for NoSQL databases. In this regard, there are some works worth mentioning.

In [15] and [23], two systems are presented that allow automated control of an elastic storage system, a distributed file system and a custom storage system, respectively. These works, to the best of our knowledge, represent the first attempts on designing true elastic storage systems. The idea behind these systems is having a control system that gathers information about workloads (request latency, utilization, response time, etc.) and decides whether to start or stop a computing instance.

In order the determine which servers are overloaded/underloaded, the system from [15] measures CPU utilization while the SCADS director [23] uses a steady-state performance model to predict whether a server can handle a particular workload, without violating a given latency threshold, according to the workload rate of get and put operations.

In MeT, we use several systems metrics (CPU utilization, I/O wait and memory usage) that are critical for a storage system and highly impacts server utilization's estimation.



(a) MeT

(b) Tiramola

Figure 6. Elasticity experiment.

Besides, our actuator component applies different heterogeneous configurations, a departure from previous approaches.

With respect to heterogeneous configuration of computational instances, in [20] the authors propose a system to autonomously change virtual machine configurations in order to adjust how resources are allocated. This allows for a certain virtual machine to be granted more resources if it has higher demand. In this particular case, the idea was applied to virtual machines running relational database management systems. For instance, a certain database management system with an heavy workload would be given more memory, thus boosting performance without impacting other lighter databases, and improving the overall performance. If the same resources would be given to every virtual machine, resources would be wasted and the system would perform below its actual capabilities. The idea of heterogeneous configuration of a pool of computational instances is similar to the one we present along this paper. It differs in the fact that we are dealing at the application level, rather than at the level of the virtual machine controller.

In [13], elasticity of NoSQL databases is subject to analysis. Three different NoSQL databases (HBase, Cassandra and Riak) are tested in order to assess their elastic capabilities. The paper presents extensive experiments that measure the cost of adding or removing nodes from those NoSQL systems. It is important to notice, however, that this control system is restricted to operations such as add or remove database instances. Data distribution is left as a database responsibility and all instances are considered equal. Furthermore, *tiramola* is oblivious to workload information or any database metric. It relies solely on CPU usage, memory consumption and other system-level metrics for its decision model. Similar behavior is obtained by the use of Amazon's Cloud Watch [1] together with Amazon's Auto Scaling [2]. The Amazon Cloud Watch service gathers system metrics while the Auto Scaling allows a user to define rules based on such metrics. These rules define what action to take (add or remove nodes) when certain metric values reach some thresholds.

Finally, there is relevant work on workload-aware partitioning in relational database management systems (RDBMS) [9, 19, 22]. Although following a workload-aware approach for database partitioning, the main goal of such works is to avoid the distributed transactions overhead. Our work focuses on workload-aware elasticity for NoSQL databases.

8. Conclusions

In this paper, we focused on automated elasticity for NoSQL databases. Firstly, we motivated our work by looking at previous approaches and introduced heterogeneous configurations of NoSQL database clusters. Current approaches to automated elasticity for NoSQL databases look at the different cluster nodes as identical entities. Therefore, elasticity is limited to the decision of adding or removing nodes from the cluster according to demand. Introducing the possibility of having cluster nodes configured heterogeneously proved to allow for better performance and resource usage. Outcome only possible when taking the workload into account.

Following our motivation tests we designed and implemented MeT. The MeT framework provides automated workload-aware elasticity for NoSQL databases. Currently, our prototype is compatible with HBase and OpenStack as the underlying IaaS. Our experiments showed that MeT was able to autonomously reconfigure an HBase cluster without the need to stop it, and achieve similar performance to that of a judiciously and manually configured one. Furthermore, we compared the performance of MeT with an existing system. From this comparison it was possible to see that MeT achieves a cluster configuration that outperforms the cluster obtained using such approach. On top of that, this result was achieved with less resources.

There is still room for improvement. Namely, considering new metrics and policies for node addition and removal. For instance, with the use of statistical machine learning to allow dynamic scaling [4]. Moreover, it is also possible to enhance MeT to consider dynamic configurations, even if such approach adds significant complexity to the reconfiguration process.

Finally, at the time this paper was written, HBase is preparing to include in its next release a new load balancer. The new load balancer, *StochasticLoadBalancer*, intends to solve some of the problems stemming from the use of a random balancer. The use of such load balancer would improve the results obtained by *tiramola* however, MeT is a step forward and proposes a refined and workload-aware load balancer that, under the heterogeneous assumption, would still achieve better performance.

Acknowledgment

We thank the anonymous reviewers and our shepherd Tim Kraska for their helpful suggestions. We would like also to thank to J. M. Valério de Carvalho for his valuable insights on bin-packing problems.

This work is part-funded by; ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project Stratus/FCOMP-01-0124-FEDER-015020; and European Union Seventh Framework Programme (FP7) under grant agreement n° 257993, project CumuloNimbo.

References

[1] Amazon.com. Amazon CloudWatch. `http://aws.amazon.com/cloudwatch/`.

[2] Amazon.com. Auto Scaling. `http://aws.amazon.com/autoscaling/`.

[3] Apache. Hadoop: Hadoop. `http://hadoop.apache.org/` (january 2011).

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 2010.

[5] R. G. Brown. *Smoothing, forecasting and prediction of discrete time series*. Prentice-Hall, 1963.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI*, 2006.

[7] S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica. TPC-E vs. TPC-C: characterizing the new TPC-E benchmark via an I/O comparison study. *ACM SIGMOD*, 2010.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[9] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. In *VLDB*, 2010.

[10] Barthélémy Dagenais. Py4J - A Bridge between Python and Java. `http://py4j.sourceforge.net/`.

[11] L. George. *HBase: The Definitive Guide*. O'Reilly Media, 2011.

[12] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 1969.

[13] I. Konstantinou, E. Angelou, C. Boumpouka, D. Tsoumakos, and N. Koziris. On the elasticity of NoSQL databases over cloud management platforms. In *CIKM*, 2011.

[14] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. In *Studies in integer programming (Proc. Workshop, Bonn, 1975)*. North-Holland, 1977.

[15] H.C. Lim, S Babu, and J.S. Chase. Automated control for elastic storage. *IEEE/ACM ICAC*, 2010.

[16] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation And Experience. *Parallel Computing*, 2003.

[17] Openstack blog entry: 'openstack foundation update'. `http://www.openstack.org/blog/2012/04/openstack-foundation-update/`. [Online; last accessed July-2012].

[18] D. Owens. Securing elasticity in the cloud. *Communications of the ACM*, 2010.

[19] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *ACM SIGMOD*, 2012.

[20] A.A. Soror, U.F. Minhas, A Aboulnaga, K Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. *ACM SIGMOD*, 2008.

[21] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, 2007.

[22] A.L. Tatarowicz, C. Curino, E.P.C. Jones, and S. Madden. Lookup Tables: Fine-Grained Partitioning for Distributed Databases. In *IEEE ICDE*, 2012.

[23] B Trushkowsky, P. Bodík, A Fox, M.J. Franklin, M.I. Jordan, and D.A. Patterson. The SCADS director: scaling a distributed storage system under stringent performance requirements. *FAST*, 2011.

[24] L. M. Vaquero, L. Rodero-Merino, and R. Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM*, 2011.

[25] R. Vilaça, F. Cruz, and R. Oliveira. On the expressiveness and trade-offs of large scale tuple stores. In *OTM*, 2010.