View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidade do Minho: RepositoriUM

Workload-aware table splitting for NoSQL

Francisco Cruz
HASLab - INESC TEC and U. Minho
Braga, Portugal
fmcruz@di.uminho.pt

Francisco Maia
HASLab - INESC TEC and U. Minho
Braga, Portugal
fmaia@di.uminho.pt

Rui Oliveira
HASLab - INESC TEC and U. Minho
Braga, Portugal
rco@di.uminho.pt

Ricardo Vilaça
HASLab - INESC TEC and U. Minho
Braga, Portugal
rmvilaca@di.uminho.pt

ABSTRACT

Massive scale data stores, which exhibit highly desirable scalability and availability properties are becoming pivotal systems in nowadays infrastructures. Scalability achieved by these data stores is anchored on data independence; there is no clear relationship between data, and atomic inter-node operations are not a concern. Such assumption over data allows aggressive data partitioning. In particular, data tables are horizontally partitioned and spread across nodes for load balancing. However, in current versions of these data stores, partitioning is either a manual process or automated but simply based on table size. We argue that size based partitioning does not lead to acceptable load balancing as it ignores data access patterns, namely data hotspots. Moreover, manual data partitioning is cumbersome and typically infeasible in large scale scenarios. In this paper we propose an automated table splitting mechanism that takes into account the system workload. We evaluate such mechanism showing that it simple, non-intrusive and effective.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Algorithms

Keywords

Distributed Systems, NoSQL, Table splitting

1. INTRODUCTION

One of the outstanding challenges of large scale web systems is data management. In fact, some of the most popular internet services (Facebook, Flickr, Twitter) face the need to handle massive amounts of data. In order to cope with large scale data management, a new class of data management systems surfaced. These systems, commonly known as NoSQL data stores, are characterized by their high scalability and high availability properties. These desirable properties stem from a new approach to data management. In contrast with traditional relational databases, NoSQL data stores do not offer atomic multi-item operations and there is no clear relationship between data from different entities. These assumptions allow NoSQL data stores to avoid the need for inter-node operations and hardly any kind of synchronization mechanisms. This comes at the expense of a richer, more powerful, query language (SQL). In fact, a typical NoSQL data store provides a simple *put* and *get* interface, and the computation of more complex operations (e.g. join operations) is executed at the client side. Even though new massive scale data stores provide a simpler API and require extra work at the client side for certain operations, they are well suited for a large class of applications. In particular, it is possible to take advantage of the lack of relationship between data entities to scale the system through data partitioning and load balancing.

In the usual case [9, 14, 4, 8, 3], data is organized into tables. These tables are horizontally partitioned into groups of tuples, which we will call *regions* from now on. Regions are spread across the cluster nodes in order to balance the load.

In current systems [9, 3], region split is automatically triggered whenever it reaches a certain threshold in size. When the decision to split is made, the data store will split it into two regions of roughly the same size. Note that, following this approach, regions are always split in half independently of their data access patterns as if assuming a near uniform data access pattern.

As an alternative, the splitting procedure can also be done manually. By manual we mean that it requires a human manually choosing splitting points. Such approach allows for fine tuning and optimal load balancing. However, in the typical case, this is unfeasible. In fact, the user would need to gather information about the data access patterns of each region in order to figure out the correct splitting point. The massive scale of data, the high number of regions and the fact that data is continuously growing make manual partitioning an impractical task.

As manual partitioning is not an option, we are left with partitioning of regions in half. At this point, we argue that such mechanism is not sufficient and that the fact it does

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'14 March 24–28, 2014, Gyeongju, Korea
Copyright 2014 ACM 978-1-4503-2469-4/14/03 ...$15.00.

not take into account data access patterns highly impairs performance. Moreover, it is common for data workloads over these data stores to exhibit non-uniform distribution of requests over data [15, 5]. Not taking into account the distribution of requests renders partitioning ineffective as a load balancing mechanism.

In this paper, we focus on the need for an automated mechanism to find region splitting points that take into account the system workload. We propose a workload aware table splitting mechanism. The mechanism proposed estimates, in an autonomous way, a region splitting point that leads to optimal load balancing. We show that the algorithm is as simple as effective. Moreover, it is a generic approach applicable to different NoSQL data stores. We also evaluate our mechanism over the HBase data store.

The paper is organized as follows. Section 2 provides a brief overview of related work. Section 3 describes our algorithm to compute region splitting points. In Section 4 the table splitting system is implemented over the HBase data store and evaluated. Finally, Section 5 concludes the paper.

2. RELATED WORK

Work on data load balancing for NoSQL is closely related to the one presented in this paper. In particular, in [6] the authors present a workload aware elastic manager for NoSQL called *MeT*. *MeT* has as its core component a load balancer. Such component supersedes the randomized load balancing mechanism, typically used as the default one [9]. Moreover, it is our claim that an autonomous table splitting mechanism would greatly enhance a system like *MeT*. In fact, being able to split tables that are overloaded would allow the load balancing mechanism of *MeT* to better distribute regions across system nodes.

Another load balancing system was proposed in [13]. However, similarly to *MeT*, this system does not consider table splits.

In [1] tables are replicated across different nodes according to their popularity. This is related to our work as it is an workload aware approach to load balancing.

An example of a splitting mechanism is included in the Yahoo! Cloud Datastore Load Balancer [11]. In this system table splitting is performed in two situations. If the table reaches a certain size or based on table load. The key difference when compared to our approach is that, even though this system takes into account table load, it does not use such information to decide the table splitting point. It is our understanding that, in this case, the Yahoo! Cloud Datastore Load Balancer splits tables into two with similar size.

Table splitting was also subject of research work in the area of relational databases [7, 16, 15]. However, in these works the main goal was to avoid multi-table queries, and thus to avoid distributed transactions. In the present one, the assumption is that there will be no multi-table queries at all. This assumption places those previous works in a different class of approaches.

There is also research work on online median estimation. Specifically in [2], the authors follow a similar overall approach to the one presented in this paper, in the domain of fetal heart rate interpretation.

Algorithm 1: Split key search algorithm.

```
begin
    foreach Region do
        Data: LowestKey ← ∞
        Data: HighestKey ← 0
        Data: splitKey ← null
On request :
    Data: key ← Request.getKey()
    Data: region ← key.getRegion()
    Data: splitkey ← region.getSplitKey()
    if key > region.HighestKey then
        Data: region.HighestKey ← key
    if key < region.LowestKey then
        Data: region.LowestKey ← key
    if splitKey == null then
        Data: splitKey ← key
    if key > splitkey then
        splitkey.increase()
        if splitkey > region.HighestKey then
            splitkey = region.HighestKey
    else
        splitkey.decrease()
        if splitkey < region.LowestKey then
            splitkey = region.LowestKey
```

3. WORKLOAD-AWARE TABLE SPLITTING

Current data stores split regions in order to distribute load across cluster nodes. The decision of when to split is made based on a size threshold. However, the splitting point itself is also size based. Typically, regions are always split in half. We argue that such splitting impairs load balancing as different regions, due to non uniform workloads, may be subject to very different load patterns.

In this paper, the main problem we address is finding a good splitting point. A good splitting point is the one that splits the region into two new regions with similar load. We define a good splitting point in this manner as it leads to better overall load balancing of requests across regions. Note that the considering multiple splitting points for a single region simultaneously is out of the scope of the present paper.

In Section 3.1 we describe our algorithm for workload aware table splitting point estimation. The algorithm is analyzed in Section 3.2.

3.1 Algorithm

With the goal of finding the splitting point we designed a simple yet effective algorithm. An important requisite is that any kind of mechanism we devise does not impose high overhead over the data store system. Doing otherwise would render it highly undesirable. Moreover, it needs to divide the region into two regions with similar load independently of the request distribution applied to the system. Having these constraints in mind we propose an algorithm that can be run asynchronously and has no impact on the data path. Moreover, as we will see, it achieves highly accurate results with negligible memory and CPU consumption.

Relying on a simple mechanism to access region load it is possible to have a good estimation of the key that splits a

region into two with similar load. The algorithm, depicted in Algorithm 1, works as follows. The key range is assumed to have a well defined order over keys. The algorithm initiates by estimating that the splitting point is the key of the first request it intercepts for each region. By taking into account subsequent requests it will progressively improve its estimation of the splitting point. For each request, if the requested key is smaller than the current estimation the algorithm decreases it. Otherwise, the estimated splitting point is increased. At each request it also updates the smallest ($LowestKey$) and the highest ($HighestKey$) object keys, of which that region is responsible for. Such information is useful to know the region boundaries.

As the reader easily notices, the increase and decrease methods are not defined in Algorithm 1. This is intentional as their implementation may vary and will impact the performance of the algorithm.

3.2 Instantiation

We consider three instantiations of the *increase* and *decrease* methods. The simplest case is to have linear increase and decrease behavior. This means that, for instance, if a request arrives for a key whose value is greater than the current splitting point, the latter will be increased by a constant value. The second instantiation is an exponential function. This means that when two or more steps are done in the same *direction* the step size increases in a quadratic fashion. The final instantiation is achieved by mixing both strategies as described later in the paper.

Considering these instantiations we set up a few experiments. We considered a key range of 10,000 keys and generated 20,000 key requests that followed a ZipFian distribution. This distribution was chosen as it is representative [10]. At each request, we looked at the splitting point estimation given by the algorithm. As the distribution was known beforehand, we used the distribution's cumulative distribution function to calculate how the regions would be split should such estimation be used. The optimal splitting point corresponds to the point where 50% of the requests fall into each one of the new regions i.e. $P(X \leq 50)$ of the cumulative distribution function.

We configured the algorithm with the linear strategy and the results of the experiment are depicted in Figure 1.

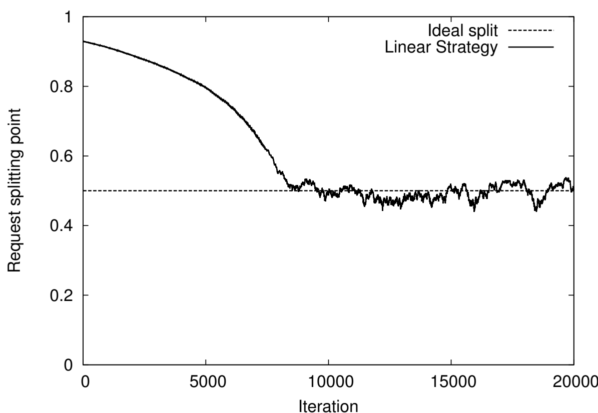


Figure 1: Split key search algorithm with linear strategy.

As shown by the experiment results, the algorithm tends to yield good approximations to the ideal split value after 8,000 iterations (that corresponds to 8,000 requests). However, by manipulating the implementation of the increase / decrease methods it is possible to improve the algorithm. The slow convergence of the approach above is due to the fact that, at each request, the algorithm is taking very small steps towards the desired point. Relying on the exponential strategy this can be avoided. As observable in Figure 2, the algorithm is now much faster at the expense of stability.

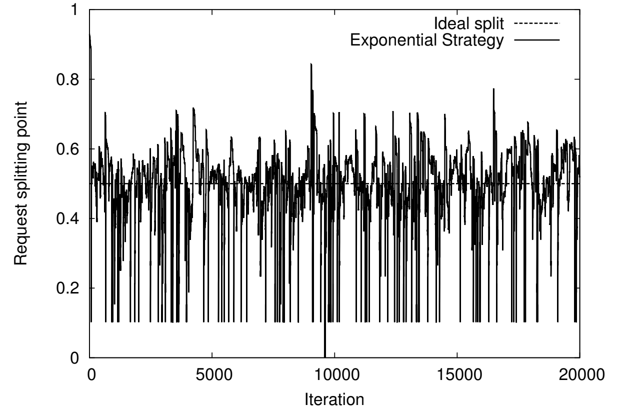


Figure 2: Split key search algorithm with exponential strategy.

The bottom line is that neither strategy is very attractive. On the one hand, the linear strategy requires a lot of iterations to converge to the optimal split point. On the other hand, the exponential strategy converges rapidly to the optimal split point, but proves to be unstable.

Therefore, our approach is based on the combination of both strategies. The idea is to start with the exponential strategy and, when sufficiently close to the ideal splitting point, change to the linear one. The challenge of this approach is knowing what *sufficiently close* means and how to detect it. To address this problem we try to detect what we call a *PingPong* zone.

Intuitively, if a splitting point is ideal, it means that the probability of a request being for a key smaller than the splitting point is equal (or roughly equal) to the probability of the request being for an higher key. Consequently, the algorithm will fall into a *PingPong* zone where the value of the splitting point will be continuously being increased and decreased. When in a *PingPong* zone, the algorithm mutates to the linear strategy.

In order to detect a *PingPong* zone we try to detect consecutive *PingPong* pairs. A *PingPong pair* is a single increase / decrease sequence. The algorithm is configurable in order to define the number of *PingPong* pairs needed to trigger the algorithm mutation. Figure 3, depicts the behavior of the algorithm configured with the linear strategy and two mixed strategies. It is possible to observe that a mixed strategy proves to be effective. Moreover, in our experiments we observed that, for this scenario, configuring the algorithm with a small number of *PingPong* pairs allows for very good results.

Another important aspect of the algorithm is that it should achieve good results independently of the request distribution. In Figure 4 are depicted the results of an experiment

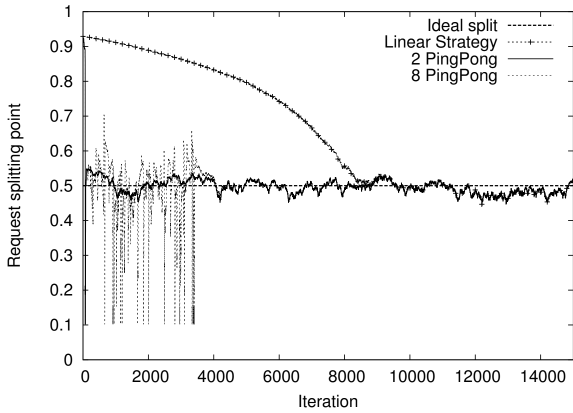


Figure 3: Split key search algorithm with PingPong detection.

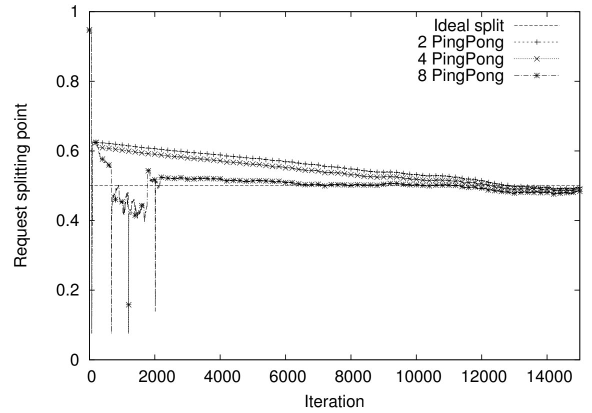


Figure 5: Split key search algorithm with PingPong detection for a large key range.

where a Poisson distribution was used. Using the same key range size of the previous experiment, 10,000 unique keys, and 20,000 requests following the Poisson distribution. As it is observable, even for this distribution, the algorithm achieves acceptable results. It is however worth noting that a Poisson distribution is a worst case scenario for finding a good splitting point. It is sufficient for the splitting key to miss the ideal one by a few intervals in order to yield very different splitting ranges.

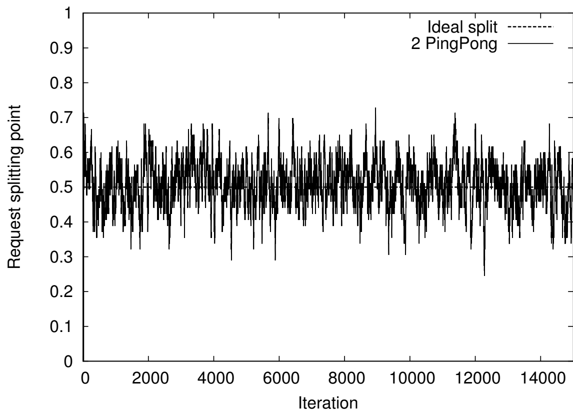


Figure 4: Split key search algorithm with PingPong detection for a Poisson request distribution.

Finally, we also wanted to evaluate the impact of the key range size in the performance of our algorithm. Figure 5 depicts the results of an experiment where the key range size was increased to 300,000 unique keys. The distribution used was, again, Zipfian. Beginning with some considerations, not depicted in the Figure, the linear strategy is much affected by the key range size, because it depends on the first request to linearly converge to the ideal splitting point. Likewise, the exponential strategy is not much affected by the key range size nor the first request, but once more has some instability. As can be observed, as opposed to the experiment with the smaller key range the different configurations of the *PingPong* really impact the behavior of the algorithm. All different configurations quickly get close to the ideal splitting point, but for 2 and 4 *PingPong* pair configuration the switch to the linear strategy occurs too soon, which impacts the convergence of the algorithm. The best results are therefore achieved by 8 *PingPong* pair configuration. Its initial instability is compensated by the closer estimation yield by the exponential strategy and converges to the ideal splitting point in almost the same number of iterations as in the previous experiment. This leads to the observation that for a larger key range size, the number of *PingPong* pairs should be slightly increased.

From the results we can safely conclude that our algorithm provides a good heuristic for finding a suitable region splitting point.

4. WORKLOAD AWARE TABLE SPLITTING IN HBASE

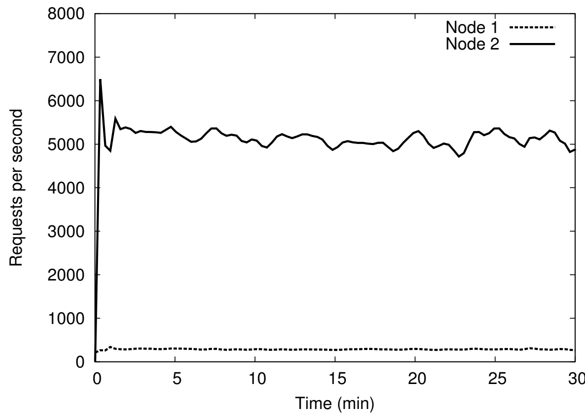
In this Section we describe and evaluate the implementation of our algorithm in HBase. Although the mechanism is generic and applicable to other NoSQL data stores, we will focus on an HBase implementation from now on. In Section 4.1 implementation details are described and the evaluation of the mechanism is presented in Section 4.2.

4.1 Implementation

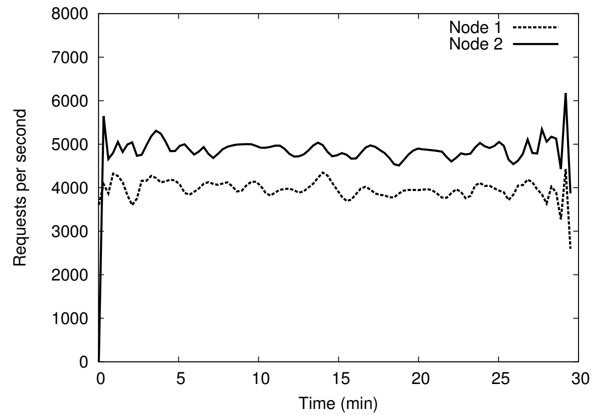
Data in HBase is organized into tables which are split into regions. HBase splits tables when these reach a certain size. As described earlier, this approach does not lead to good load balancing when data requests obey skewed distributions.

In order to implement our automated workload aware table splitting mechanism, we have added the mechanism of the previous Section to the HBase data store itself. Consequently, when HBase is running a splitting point estimation is calculated for each region continuously. In particular, it is calculated within each Region Server (HBase node) and exported as a JMX metric accessible through the HBase client interface.

With this implementation it is now possible to split regions in a workload aware fashion. It is important to notice that the default load balancer of HBase tries to achieve similar number of regions in every node. Using our mechanism, which yields regions with similar load, eases such process.



(a) HBase out-of-the-box with uniform splitting.



(b) HBase with workload-aware splitting.

Figure 6: Node load for the two scenarios.

4.2 Evaluation

In this Section we present results of our evaluation. The experiment was set up as follows. A HBase cluster was deployed across two nodes. A single table was created and placed on one of the nodes. The table was populated with 1,000,000 records (1 GB of data) using YCSB [5] that was deployed in a separate machine. The same YCSB instance was configured to produce a workload with 80% of read requests and 20% of write requests. Moreover, such workload follows a ZipFian distribution. 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.

The table was intentionally designed to be too large to be handled by a single node. Consequently, it is split into two regions one on each node. At this point, two different scenarios were considered and evaluated. Scenario one corresponds to the out-of-the-box HBase behavior. HBase splits the table into two regions of the same size regardless of the access pattern. In scenario two, our splitting mechanism is in place. The initial set up is similar however.

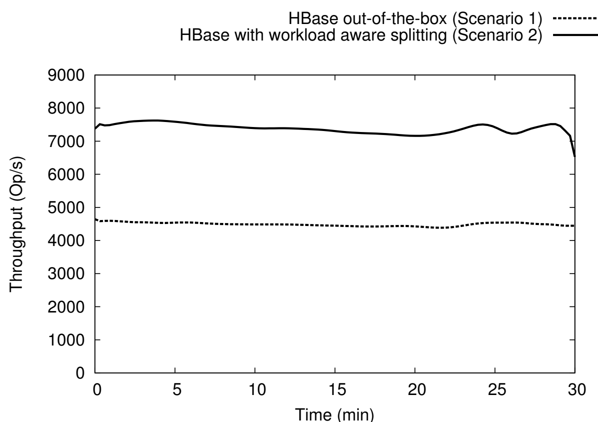


Figure 7: Evaluation over HBase. Throughput achieved in scenarios one and two.

For both scenarios we logged the load imposed on each of the nodes. In order to determine how load was being distributed across both of the cluster nodes. The results are depicted in Figure 6.

The split made by the default mechanism not only leads to an highly unbalanced cluster but is also highly infective (Figure 6a). In fact, node 2 of 6a is saturated even after the split while node 1 is practically idle. This reduces the cluster capacity to virtually the capacity of the single node. An overloaded node, with this split, remains overloaded as the load moved to the other node is negligible.

In contrast, our approach allows to almost double the overall throughput as depicted by Figure 7. Splitting the table using the splitting point given by our algorithm allows for load to be distributed across the two nodes taking advantage of the capacity of both (Figure 6b).

Note that, after the split the nodes remain under high load. Allowing a second split round and allocating new machines could increase immensely the performance of this cluster. This is not the case when using the traditional split mechanism. Splitting without taking into account the load will always result in an highly unbalanced cluster impairing performance.

The results validate the approach and show that the algorithm proposed is effective in practice. Moreover, it also opens future research paths as automated workload aware table splitting for NoSQL seems an objective worth pursuing.

5. DISCUSSION

In this paper we presented a workload aware table splitting algorithm for NoSQL. We evaluated it and proved it to be effective in practice. Although it may seem simple, the algorithm proposed is a pragmatic approach to automated splitting point discovery. The results obtained showed it is effective both for achieving good load balance as well as improving overall performance of HBase.

From the results obtained we believe the present work opens various very promising research paths. The most immediate concerns the mechanism itself. In fact, it should be interesting to find a convergence criteria. The current implementation needs the observation interval to be representative of the distribution. Moreover, it is not capable of knowing, in an autonomous way, if such requisite is met. It

should be possible for the algorithm to be enhanced with some technique that would allow it to stop when a suitable splitting point is found.

Another key issue is the *PingPong* zone detection criteria. Currently this is a system parameter. Finding a way to determine this parameter in an autonomous way could greatly improve the present work.

Finally, another research path is related to the work in [12, 6]. It is our understanding that the mechanism presented in the current paper can greatly enhance works as such. Moreover, heterogeneity as presented in [6] presents itself as a very interesting challenge to the workload aware table splitting algorithm. As a result it should be possible to design and implement a fully autonomous elasticity controller for NoSQL that takes advantage of table splitting for enhance load balance and performance. In particular, such controller could take into account cluster load also for the decision of *when* to split regions as this is not considered in the present work.

6. ACKNOWLEDGMENTS

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 and FCOMP-01-0124-FEDER-022701; within project PEst/FCOMP-01-0124-FEDER-037281; and European Union Seventh Framework Programme (FP7) under grant agreement nr 611068 (CoherentPaaS).

7. REFERENCES

[1] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 287–300, New York, NY, USA, 2011. ACM.

[2] T. Bylander and B. Rosen. A perceptron-like online algorithm for tracking the median. In *Neural Networks,1997., International Conference on*, volume 4, pages 2219–2224 vol.4, 1997.

[3] 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.

[4] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *VLDB Endowment*, 2008.

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

[6] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. a. Paulo, J. Pereira, and R. Vilaça. Met: workload aware elasticity for nosql. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 183–196, New York, NY, USA, 2013. ACM.

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

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, 2007.

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

[10] A. Java, X. Song, T. Finin, and B. Tseng. Why we twitter: understanding microblogging usage and communities. In *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*, WebKDD/SNA-KDD '07, pages 56–65, New York, NY, USA, 2007. ACM.

[11] M. Klems, A. Silberstein, J. Chen, M. Mortazavi, S. A. Albert, P. Narayan, A. Tumbde, and B. Cooper. The yahoo!: cloud datastore load balancer. In *Proceedings of the fourth international workshop on Cloud data management*, CloudDB '12, pages 33–40, New York, NY, USA, 2012. ACM.

[12] I. Konstantinou, E. Angelou, D. Tsoumakos, C. Boumpouka, N. Koziris, and S. Sioutas. Tiramola: Elastic nosql provisioning through a cloud management platform. In *ACM SIGMOD/PODS International Conference on Management of Data (Demo Track)*, 2012.

[13] I. Konstantinou, D. Tsoumakos, I. Mytilinis, and N. Koziris. Dbalancer: distributed load balancing for nosql data-stores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1037–1040, New York, NY, USA, 2013. ACM.

[14] A. L. and P. M. Cassandra - A Decentralized Structured Storage System. 2009.

[15] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *ACM SIGMOD International Conference on Management of Data*, 2012.

[16] A. Tatarowicz, C. Curino, E. Jones, and S. Madden. Lookup tables: Fine-grained partitioning for distributed databases. In *ICDE*, 2012.