Resource Usage Prediction in Distributed Key-Value Datastores

Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, João Paulo, José Pereira, Ricardo Vilaça

▶ **To cite this version:**

Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, João Paulo, et al.. Resource Usage Prediction in Distributed Key-Value Datastores. 16th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS), Jun 2016, Heraklion, Crete, Greece. pp.144-159, 10.1007/978-3-319-39577-7_12 . hal-01434791

HAL Id: hal-01434791
https://inria.hal.science/hal-01434791

Submitted on 13 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Resource usage prediction in distributed key-value datastores

Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, João Paulo, José Pereira, and Ricardo Vilaça

INESCTEC & Minho University
{fmcruz,fmaia,miguelmatos,rco,jtpaulo,jop,rmvilaca}@di.uminho.pt

Abstract. In order to attain the promises of the Cloud Computing paradigm, systems need to be able to transparently adapt to environment changes. Such behavior benefits from the ability to predict those changes in order to handle them seamlessly. In this paper, we present a mechanism to accurately predict the resource usage of distributed key-value datastores. Our mechanism requires offline training but, in contrast with other approaches, it is sufficient to run it only once per hardware configuration and subsequently use it for online prediction of database performance under any circumstance. The mechanism accurately estimates the database resource usage for any request distribution with an average accuracy of 94%, only by knowing two parameters: i) cache hit ratio; and ii) incoming throughput. Both input values can be observed in real time or synthesized for request allocation decisions. This novel approach is sufficiently simple and generic, while simultaneously being suitable for other practical applications.

1 Introduction

The ability to predict how a system will behave is critical in Cloud Computing systems. Accurate prediction would allow administrators to make better informed decisions on resource allocation, systems configuration or even the technology to use. Currently, this typically requires extensive testing while still lacking the desirable accuracy levels. This is particularly true for massive scale distributed key-value datastores (often named NoSQL databases), Notably, their highly desirable performance, scalability and availability properties cannot be achieved without careful resource allocation and judicious data placement, which requires extensive testing.

In this paper we demonstrate that, for distributed key-value datastores, it is possible to achieve accurate performance prediction, in real-world scenarios, resorting to only a small fraction of the systems resources. NoSQL datastores make heavy use of buffer caching, specially to improve the performance of read requests. In this work we show that the success of such caching layer is directly related to the datastore's resource consumption and we leverage such relation for resource prediction purposes. In fact, it is known that, for a given throughput, the higher the cache hit ratio, the lower the resource usage of the NoSQL datastore. This is true since each cache hit avoids resource consumption stemming from lower layer access. Moreover, contrary to relational databases, which due to their inherent complexity require more elaborate models, in this work we

show that for distributed key-value datastores this correlation is actually enough to accurately predict resource usage of any workload. Such accurate prediction of resource usage then allows system optimization, preparation, and simulation under different conditions. This is particularly important if we aim to effectively deploy NoSQL data stores in the pay-as-you-go model, which is common in the Cloud Computing paradigm.

Contributions. a) We provide a mechanism to build a read operation resource usage model and a write operation resource usage model. Both models are hardware dependent, meaning they need to be rebuilt when the hardware changes, but they are generated only once per hardware configuration and can then be used to predict the resource usage for any workload. b) Leveraging these models, we are able to predict a NoSQL datastore resource usage, only by knowing two parameters: i) the cache hit ratio and ii) the incoming throughput. From our experiments using HBase, we accurately predict resource usage for any request distribution and any throughput of read-only and a mix of read and update operations. We achieve an average prediction accuracy of 94%.

Roadmap. The rest of this paper is organized as follows. We begin by providing some background about caching mechanisms and NoSQL datatores in Section 2. Section 3 presents evidence on the correlation between the cache hit ratio of NoSQL datastores and resource usage. Section 4 focuses on the prediction of resource usage for read-only operations while Section 5 focuses on write operations. We validate our mechanism using HBase and mixed (read/write) workloads in Section 6, present related work in Section 7 and conclude with Section 8.

2 Background

Caching mechanisms. Databases make use of buffer caching to improve their read performance. By keeping most frequently accessed data in fast access structures (either implemented by software or hardware) performance can be significantly improved. As a result, the flow of a read request usually takes the following path: i) the client issues a request to read some tuple; ii) the database verifies if the requested tuple is in cache; iiia) if it does the tuple is returned to the client, iiib) otherwise the database tries to fetch the tuple from secondary memory. When using caching one of the main goals is to try to maximize the percentage of requests that are served from cache, also known as the cache hit ratio. A high hit cache rate means that a good number of requests are being served exclusively by the cache, thus avoiding higher CPU and I/O costs from using less efficient storage mediums. When the data size exceeds the cache size, eventually, some data in the cache needs to be removed to give room to more frequently accessed data. This is handled by cache replacement algorithms [23]. There are several cache replacement algorithms, however one of the most widely used algorithms is the Least Recently Used (LRU) algorithm [25]. Under this replacement algorithm when the cache is full, the algorithm discards data that was least recently used. This algorithm is the one typically used by distributed key-value datastores [12][18].

Distributed key-value datastores. Distributed key-value datastores run in a distributed setting with tenths to hundreds of nodes, usually composed of commodity hardware. The application data is partitioned and these partitions are assigned to the available nodes according to a data placement strategy. Contrasting with relational database

management systems (RDBMS), these datastores only provide a simple key-value interface to manipulate data by means of put, get, delete, and scan operations and they do not offer strong consistency criteria. Complex operations like joining and aggregation are not present and data is denormalized. Considering these characteristics, the success of caching mechanisms is key for performance. In this paper, we focus on HBase which is one of the most successful and widely used key-value datastores [12]. Inspired by BigTable [4], 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. HBase follows a hierarchical architecture where there is a Master node and there is one or more slave nodes called *Regionservers*. The row range of a *HTable* is horizontally partitioned into *Regions* and distributed over different nodes. Each *Region* is stored as an append-only 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*. HBase has a *block cache* implementing the LRU replacement algorithm. Several key-values are grouped into blocks of configurable size and these blocks are the ones used in the cache mechanism. The block size within the *block cache* is a parameter but defaults to 64KB.

3 Interdependence of resource usage and cache hit ratio

Let us consider a server usage metric related to the CPU waiting time on I/O operations (I/O_{wait}), the time spent on user space (CPU_{user}) and the time spent on kernel space (CPU_{system}) in the form: $Server_{usage} = I/O_{wait} + CPU_{user} + CPU_{system}$. In the following we show the cache hit ratio is effectively related to server usage. To this end, we set up three experiments using a HBase deployment and YCSB [6] as the workload generator. These experiments cover a wide spectrum of possible behaviors. With these we are able to show a clear and direct relationship between the cache hit ratio and server usage in NoSQL systems, which lays the foundation for the rest of the paper.

Experimental setting: In all experiments, one node acts as master for both HBase and HDFS, and it also holds a Zookeeper [14] instance running in standalone mode, which is required by HBase. Our HBase cluster was composed of 1 *RegionServer*, configured with a heap of 4 GB, and 1 *DataNode*. HBase's LRU *block cache* was configured to use 55% of the heap size, which HBase translates into roughly 2.15 GB. The *RegionServer* was co-located with the *DataNode*. The YCSB workload generator ran in a separate node and was configured with a *readProportion* of 100% (read-only), and with a fixed throughput of 2000 operations per second with 75 client threads. All experiments were set to run for 30 minutes with 150 seconds of ramp up time and the results are the computed average of 5 individual runs. The server usage was logged every second in the *RegionServer/DataNode* machine using the UNIX *top* command. The *top* command gives us the CPU_{idle} metric that is converted to our $Server_{usage}$ metric in the form: $Server_{usage} = 100\% - CPU_{idle}$. By the end of each experiment, we gathered the *RegionServer*'s achieved cache hit ratio. All nodes used for these experiments have an Intel i3 CPU at 3.1GHz, 8GB of main memory, a 7200 RPM SATA disk, and are interconnected by a switched Gigabit network.

First experiment: In this first experiment, a single *region* was populated using the YCSB generator with 4,000,000 records (4.3 GB). This means that the *region* cannot be fitted entirely into the *block cache*: about 1.1 millions records (1.21 GB) remain on secondary memory and must be brought into main memory when requested. There were four different scenarios each with a differently configured request popularity:

1. A *uniform* popularity distribution, that is all records have equal probability of being requested (the case where the cache hit ratio is minimum);
2. A *hotspot* popularity distribution, where 50% of the requests access a subset of keys that account for 30% of the key space;
3. A *zipf scrambled* popularity distribution, highly skewed, but because it is scrambled it means the most popular keys are spread across the key space;
4. A *zipf clustered* popularity distribution, highly skewed, and clustered, meaning the most popular keys are contiguous, which makes them fall in the same cache block.

The results for this experiment are depicted in Table 1. As expected, the *uniform* request popularity is the one that achieves the lower cache hit ratio ($p_{hit} = 49\%$) , and thus consumes more server resources (58.35%) while the *zipf clustered* request popularity has the higher cache hit ratio (93%). This is true because popular keys are found in the same block, which is maintained in memory avoiding cache misses.

Distribution	p_{hit}	Average $Server_{usage}$	#Records
Uniform	49%	58.35%	4,000,000
Hotspot	56%	46.19%	4,000,000
Zipf Scrambled	68%	35.91%	4,000,000
Zipf Clustered	93%	19.28%	4,000,000

Table 1. Average $Server_{usage}$ and cache hit ratio (p_{hit}) with a *region* larger than the *block cache*.

Second experiment: We set up a second experiment, to demonstrate that the behavior observed in the first experiment is independent of request popularities. This experiment is identical to the first one except for the *region* size, which has now 2,000,000 records (2.14 GB). As a result the *region* fits entirely into the *block cache*, thus the expected cache hit ratio is 100%. Table 2 depicts the results. As all data is served only by the *block cache*, the different request popularities are, as expected, irrelevant to server resource consumption and all distributions use roughly the same resources.

Distribution	p_{hit}	Average $Server_{usage}$	#Records
Uniform	100%	12.29%	2,000,000
Hotspot	100%	12.14%	2,000,000
Zipf Scrambled	100%	12.92%	2,000,000
Zipf Clustered	100%	12.89%	2,000,000

Table 2. Average $Server_{usage}$ and cache hit ratio (p_{hit}) with a *region* that fits in *block cache*.

Third experiment: In this experiment we show that two different distributions, with different data sizes but with the same cache hit ratio, will have the same server resources consumption if subject to the same fixed throughput. We used a similar setting to the first experiment's but changed the number of records of the *uniform* distribution to 2,141,881(2.3 GB) so its cache hit ratio could also be 93%. The throughput is again fixed at 2000 operations per second. Table 3 depicts the results that support our claim

Distribution	p_{hit}	Average $Server_{usage}$	#Records
Zipf Clustered	93%	19.28%	4,000,000
Uniform	93%	19.76%	2,141,881

Table 3. Average $Server_{usage}$ and cache hit ratio (p_{hit}) results for 2 distributions with different sizes, but with same cache hit ratio.

that, for a given throughput, an identical cache hit ratio, regardless of the data size and the distribution results in the same resource consumption.

Correlation between server usage and cache hit ratio: A correlation test using the *Fisher's z transformation* [11] with the data from the previous experiments, shows that in fact there is a negative correlation for *p-value* < 0.001, making it statistically significant. Based on these results, we argue that it is possible to estimate the server usage given the incoming throughput and the cache hit ratio and, in the following sections, we show how this can be done.

4 Estimating resource usage of read operations

Previous section demonstrated that there is an intrinsic relation between resource usage and cache hit ratio. The cache hit ratio reflects not only the data size, but also the underlying distribution of requests which, in combination with an incoming throughput, corresponds to a given server usage. Furthermore, for a fixed throughput this relation is univocal: for some throughput if two distinct workloads consume the same amount of resources, then they must have the same cache hit ratio. In this section, we show how the server usage of any workload can be estimated simply by knowing its cache hit ratio and incoming throughput. We build on the aforementioned properties to build a tridimensional model, that models the server usage for a NoSQL datastore, when the cache hit ratio and the throughput vary. The objective is to build a model that, for a given hardware configuration, a given hit cache ratio and certain request throughput of an HBase node, allows us to predict the resource consumption of such node. To achieve this we require an initial training step, which is hardware dependent. Consequently, each generated model is only valid for a single hardware configuration but is required to be generated only once. Once we have the model we are able to predict HBase node resource usage for any given workload. As shown previously, request distribution is irrelevant in terms of the relationship between hit cache ratio and resource consumption. Taking advantage of this observation we always consider the uniform distribution in the generation of the prediction models. In fact, such model will still be valid if, on runtime, a different request distribution is observed.

In order to generate the model, our approach is to judiciously choose a number of representative combinations of cache hit ratio and throughput, test them against the desired hardware configuration and then, by using linear interpolation between the different server usage levels measured, we are able to build a tridimensional model that correlates data size, with throughput and expected server usage. Notably, with this approach we are able to achieve very high levels of accuracy. At this point, it is important to note that, for a generic workload generator, it is not possible to define the desired cache hit ratio. Instead we can only set the data size and desired throughput. However,

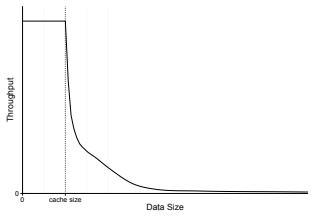


Fig. 1. Typical relation between cache size and throughput for a fixed $Server_{usage}$.

we can take advantage of a simple approach proposed by *Che et al.* [5] that provides an estimation without error of the cache hit ratio for the uniform distribution. This way, we can represent the cache hit ratio by its correspondent data size when building the model. Therefore, in the remainder of this section we will mention data sizes implicitly mentioning their correspondent cache hit ratios. Another reason for choosing the uniform distribution is because it allows to reduce the overall training time since it represents the worst case for LRU caches (lowest possible hit ratio for a given data size), thereby the time it takes to populate the data in the NoSQL database is smaller

Regarding the process of choosing the representative measures to take, let's begin by looking at an illustrative example. Figure 1 shows the behavior of incoming throughput when data sizes increase for a fixed server usage percentage. Note that, if the data size is smaller than the cache size then only the throughput impacts server usage. In this case, the cache hit ratio is always 100% and the throughput constant for all possible data sizes between 0 and *cache size*. For larger data sizes, the cache hit ratio drops and cache swapping begins, which in turn means that in order for the server usage to stay the same the throughput must decrease. As a result, this is a boundary point (where *data size* equals to the *cache size*). This observation allows us to reduce the number of points to calculate for that section as we just need to build the model from that point onwards. Then, other observations help us choosing the points to measure. For data sizes slightly larger than the boundary point, there is a big drop on throughput in order to resource usage to remain the same. This drop can be more or less abrupt depending on the speed of the secondary memory. In order to capture this behavior in the model we need to increase the number of tested combinations of pairs data size and throughput immediately after the boundary point. Conversely, when the data size is largely increased we can be confident of a long and flat tail, thus not requiring many training points to achieve high accuracy.

The uniform distribution server usage model is automatically generated resorting to a developed Python script and using YCSB as the workload generator[1]. Generally, this script has 2 main parameters: i) a list of cache hit ratios and ii) a list of targeted server usage levels. Hit cache ratios are, as explained earlier, converted to data sizes using the Che's approximation. Then, resorting to a binary search, the script tries to find the necessary throughput of read operations to achieve each specific percentage of server usage for each data size defined as input. Fixing the server usage level and allowing the

[1] All the scripts used in this work are openly available at `github.com/fmcruz/suhcr/`

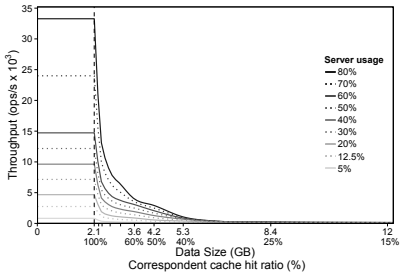


Fig. 2. Instantiation of the server model for read operations based on a uniform distribution.

throughput to be experimentally calculated via the script, allows us to have a representative number of server usage levels without having to test multiple cache hit ratio and throughput combinations in order to have a usable model. When a sufficient number of points for a specific server usage level are found and we resort to interpolation between those points. Namely, using the monotonic spline interpolation of the R project [2] embedded into the Python script. This process is repeated for each of the targeted server usage levels. This list does not comprehend all of the possible values between 0 and 100%. Instead, from our experience we noted that a few of them is sufficient (usually 5 equally spaced). Furthermore, by again using linear interpolation between the different server usage levels we achieve very accurate results, and ultimately build a tridimensional model that correlates data size, with throughput and expected server usage.

Model instantiation in our cluster We ran the automatic server model generator in our cluster using the same setting as the experiments of Section 3. The generated server model is as depicted in Figure 2. There were defined 10 different cache hit ratios: 100%, 95%, 90%, 80%, 70%, 60%, 50%, 40%, 25%, and 15%. These cache hit ratios were then transformed in their data size equivalents to be used as input in the model generator. The first point is the boundary point corresponding to 2,000,000 of YCSB records. As previously stated, for data sizes slightly larger than the *cache size* we need to increase the density of points tested to ensure the model is more accurate. Thus, the next point is only a 5% decrease, and the subsequent 6 points are decreases of 10% in the cache hit ratio. On the other hand, predicting a flat long tail from that point on, we just defined 2 points much more apart from each other, 25% and 15% of cache hit ratio, corresponding to 8,000,000 and 12,000,000 records.

In Figure 2 the solid lines correspond to the 5 targeted levels of server usage, namely 80%, 60%, 40%, 20% and 5%. It is general practice in frameworks for automated elasticity of NoSQL datastores [17] that the rule governing the addition of new nodes indicate 80% as the maximum usable CPU before a new node is needed in the cluster. This is an empirical higher bound on usable CPU to accommodate operating systems processes, account for possible load spikes and compactions. Therefore, the highest defined level was 80%. When eventually the generator has finished searching for the throughput needed to reach the targeted levels of server usage for the various data sizes,

[2] http://www.r-project.org

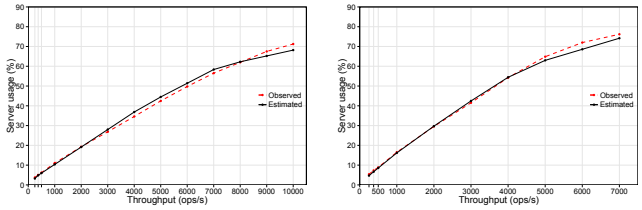
it then interpolates the data that resulted in the represented continuous curves. Finally, we just need to do a final and linear interpolation between these curves. The curves that correspond to the linear interpolation are represented by dotted lines for the server usage levels of 70%, 50%, 30% and 12.5%, which are example levels.

Model accuracy Revisiting the first experiment of Section 3, we can now use the generated model to estimate the server usage for the different distributions. The results are depicted in Table 4. As can be seen, the estimated server usage is almost the same as the observed average server usage, despite all four different distributions with very different cache hit rates. It should be noted that, as expected, the approach predicts the server usage of the uniform distribution with accuracy of 100% due to the similarity between the input usage levels of the model and the ones used in the test. We can also use the generated model to accurately estimate the server usage when the incoming throughput varies. In that regard, we set up two different experiments using the exact same setting as in the experiments of Section 3. For every data point there were 3 independent runs, and the results presented are the computed average. In the first experiment, we populated the HBase instance with 4,000,000 records (4.3 GB). The YCSB's client was configured to use the *zipf clustered* distribution with 100% read operations, and for a fixed throughput ranging from 250 ops/s to 10,000 ops/s. We also wanted to validate what happens when using a data size not used in the model generator. As a result, we populated the HBase cluster with 3,000,000 records (3.15 GB), and this time using the *zipf scrambled* distribution, which yields a much lower cache hit ratio (78.8%). As a result, the configured read throughput ranged from 250 ops/s to 7,000 ops/s. The results for each experiment are depicted in Figure 3(a) and in Figure 3(b). They show the estimated server usage compared to the observed one. The estimated results are drawn from our approach using the generated model for read operations, and observing the cache hit ratio as provided by HBase exported metrics. As expected, the estimated server usage in both experiments is very similar to the observed counterpart.

Distribution	Observed *usage*	Estimated *usage*	Accuracy
Uniform	58.35%	58.35%	100%
Hotspot	46.19%	45.87%	99.31%
Zipf Scrambled	35.91%	36.29%	98.94%
Zipf Clustered	19.28%	19.15%	99.33%

Table 4. Observed $server_{usage}$ and Estimated $server_{usage}$ results under four different distributions.

Discussion The approach described in this section allows to accurately estimate the server usage resorting to an offline trained model based on the uniform distribution. Using the cache hit ratio and the incoming throughput as the only parameters that affect resource utilization may appear oversimplifying. Specially, when taking account related approaches to usage prediction in RDBMS. However, key-value datastores are fundamentally different from relational databases. In order to attain high scalability, high throughput and high availability, these datastores offer a simple key-value interface based on put and get operations without providing multi record atomic operations nor complex operations like joins and aggregations. On the other hand, RDBMS must cope with a large number of concurrent and lock-prone ACID transactions and need different more complex models for resources, such as CPU, RAM, disk I/O and database



(a) HBase: 4 million records; *zipf clustered*.

(b) HBase: 3 million records; *zipf scrambled*.

Fig. 3. Experiments for read-only operations.

locks. These differences allow our simple but effective technique to work. The empirical intuition of why other parameters, such as the I/O costs, do not need to be considered separately is because they are already concealed in the training model. Taking a closer look into the behavior of each distribution in the first experiment, and decomposing the overall throughput into operations hitting and missing the cache, we have:

- *Uniform* - 49% of cache hit ratio; thus 980 ops/s are cache hits, the remaining 1020 ops/s miss the *block cache*;
- *Hotspot* - 56% of cache hit ratio; thus 1120 ops/s are cache hits, the remaining 880 ops/s miss the *block cache*;
- *Zipf Scrambled* - 68% of cache hit ratio; thus 1360 ops/s are cache hits, the remaining 640 ops/s miss the *block cache*;
- *Zipf Clustered* - 93% of cache hit ratio; thus 1860 ops/s are cache hits, the remaining 140 ops/s miss the *block cache*.

By looking at the average resource usage for each distribution, it is obvious that the cost of a cache miss is greater than the cost of accessing the *block cache*. This implies that the server usage for read operations can be decomposed as the sum of two costs: $Usage_{read} = Usage_{hit} + Usage_{miss}$ The $Usage_{hit}$ is the cost of only accessing the cache, while the $Usage_{miss}$ represents the cost of a miss in the cache. It covers not only the cost of bringing a block into the cache (either from main memory or disk), but also the cost of discarding the least recently used data to make room for the new data block. Thus, when two workloads have identical cache hit ratios and identical incoming throughputs, it means that both workloads have the same number of operations hitting the cache and the same number of operations missing the cache. As a result, once two workloads exhibit the same $Usage_{hit}$ and $Usage_{miss}$, ultimately exhibit the same server usage.

5 Estimating the resource usage of update operations

Although workloads are generally dominated by reads, most applications also have updates. We apply a similar approach to update operations. Updates and writes can be used interchangeable, because in key-value datastores, such as HBase and Cassandra, updates and new writes are append-only, so they follow the same write path. Updates in these datastores are first written to main memory before being flushed to disk. Therefore, the resource cost of an update is essentially related with the operation of writing

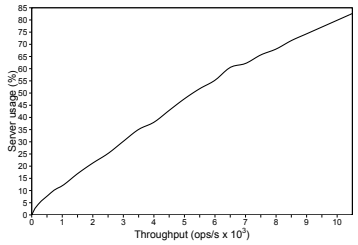


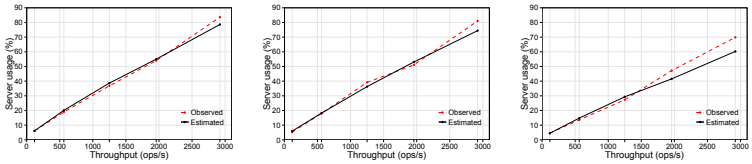
Fig. 4. Instantiation of the model for update operations.

the update to main memory and, from time to time flushing it to secondary memory. As a consequence, contrary to read requests, updates are mostly independent of the request distribution and current data size. In addition, because the write path and the read path in a NoSQL datastore are substantially separated, the overall server usage can be defined as the sum of the usage related with read operations and the usage related with update operations: $ServerUsage_{overall} = ServerUsage_{read} + ServerUsage_{update}$. As updates are independent of the request distribution and the data size, creating a model to predict the server usage of update operations is simpler than the read model counterpart. The only variable affecting the server utilization is, thus the write throughput.

Analogous to the model generator for read operations, we used a Python developed script to generate the server usage model for update operations. It also uses YCSB as the workload generator, but this time configured for updates. As the update model only depends on the throughput, the script has only one main parameter: a list of targeted update throughput points to test. For every element of targeted update throughput there are 3 independent runs, and the server utilization is logged every second in the remote machine where the datastore node is running. When all the defined points are finished, we also resort to interpolation between those points. Like the server model of read operations, the automatic server model generator was used on our own cluster, using the exact same setting. The generated server model for updates is depicted in Figure 4. There were defined 28 different targeted update throughputs from 5 updates per second to 10,000 updates per second. For increased accuracy, the first 10 targeted throughputs fall within the interval of 5 to 1000 updates per second. From that point on, there were 500 increments until 10,000 updates per second, which is the point where the server usage reaches 80%. As can be seen the server utilization for update operations grows linearly with the increased throughput.

6 Resource estimation for read-write workloads

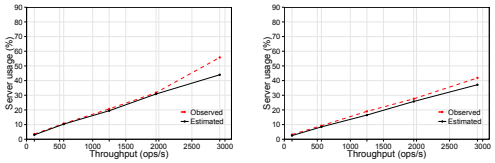
Along this section we validate our approach using HBase showing that we can accurately predict resource consumption for any given workload even if there is a mix between read and update operations. In the experiments we used the exact same setting as in the experiments of Section 3. For every data point there were 3 independent runs, and the results presented are the computed average. By using both the read and the update



(a) 90% read and 10% update.

(b) 80% read and 20% update.

(c) 50% read and 50% update.



(d) 20% read and 80% update.

(e) 10% read and 90% update.

Fig. 5. Read and update operations mix experiments in HBase.

model, we are able to estimate the server usage for read operations and update operations independently. However, as described in Section 5 the write path and the read path are mostly separated, thus we are able to estimate the overall server usage just by adding both estimations. In order to validate this assumption, we set up an experiment configured with different read and update mixes, namely: 90% read and 10% update; 80% read and 20% update; 50% read and 50% update; 20% read and 80% update; 10% read and 90% update. The *region* was populated with 4,000,000 records (4.3 GB) and the requests followed the uniform distribution with a fixed throughput of 118 ops/s, 562 ops/s, 1250 ops/s, 1958 ops/s and 2921 ops/s. These tested throughputs correspond to 5%, 20%, 40%, 60% and 80% server usage levels, as generated by the server model for the uniform distribution. In Figure 5 is depicted the results for this experiment. It shows that our approach is valid and it accurately predicts the server usage even when there are read and update operations simultaneously. However, as seen in Figure 5(c) and Figure 5(d) for the higher values of throughput the observed server usage is higher than the estimated one. These differences can be explained by compactions occurring during the test period that disrupt the readers of records stored on disk. Figure 6 shows the server usage along the entire 30 minute run for the 20% read and 80% update mix (Figure 5(d)) for the 2912 ops/s throughput. Until the compaction process starts (at 1277 seconds) the observed server usage average is the same as the estimated one (44%). Then, the compaction process greatly increases server usage to levels near 100%. When compaction ends regular behavior is resumed. This process greatly impacts the overall server usage average, but even at this point our estimated server usage is only off by 12%, which is the greatest difference observed. It is worth noting, however, that while more powerful hardware and particularly SSDs would attenuate the problem and help improve the estimation, in [2] it is proposed to offload compactions to a dedicated compaction server to prevent the significantly degraded read performance during compactions.

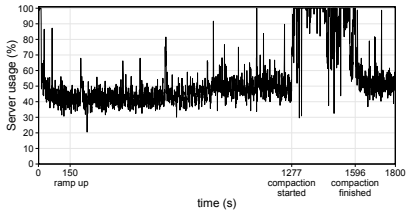


Fig. 6. Observed server usage along a 30 minute run for the 20% read and 80% update mix for 2912 ops/s throughput.

7 Related Work

A significative group of approaches aims at predicting the resource usage of generic systems such as, virtual machines, thus requiring complex models that must take into account many parameters [27] [26]. As mentioned in the literature, in order to obtain accurate models with fewer variables, it is key to focus on specific applications [15]. This is the case of performance prediction for RDBMS focused on online transaction processing (OLTP) [22] [21]. Although our work has similarities with the previous approaches, such as resorting to off-line model training, it has different assumptions from RDBMS. These differences significantly change the required approach to accurately predict the performance of key-value datastores. A single resource model is also not achievable for related work that predicts the performance of SQL queries by using models for each database operator (e.g., Sort, Merge Join), which are not present in NoSQL datastores [19]. It is worth mentioning the work on performance prediction for database consolidation, where several database instances are running in the same server and processing different types of workloads and, in many cases, even distinct schemas [1] [8]. Once again, this work needs to deal with the added complexity of RDBMS.

Regarding the techniques used to predict systems' performance, machine learning and analytical modeling are the most commonly used [20]. These can be used exclusively or in combination, by resorting to time-series analysis [16] [13], regression models [9] [28], and clustering [24]. These approaches require lengthy training phases to estimate accurately different workload distributions. It is however possible to reduce the duration of this initial phase by using a less accurate model and then refine it, in runtime, with other machine learning algorithms [10]. Because we target a specific type of system, we are able to reduce our model to only two parameters, the cache hit ratio and incoming throughput. Our approach achieves high accuracy without needing runtime improvements for the model. To the best of our knowledge our approach is the only work that can accurately predict the performance of a NoSQL datastore with a single model. Even if our solution needs offline training, it does not require system traces or runtime mechanisms to improve the precision of the estimation.

8 Conclusion

Along this paper we focused on a mechanism for distributed key-value datastores resource usage prediction. Our mechanism is able to accurately predict the resource uti-

lization for every data size, request distribution and throughput combination. In contrast with previous approaches on prediction systems for cloud environments, we take advantage of focusing on a specific cloud component to improve prediction accuracy and its applicability. In particular, we observed that the majority of the NoSQL systems make use of buffer caching mechanisms to improve performance. Moreover, the effectiveness of such mechanisms is directly related with the performance and, as a consequence, to the resource utilization of the database. This effectiveness can be measured in terms of the hit ratio that the caching mechanism exhibits. The higher the cache hit ratio the more effective the cache mechanism is, and thus more efficient is the database. In this work, we show that a NoSQL workload can be characterized by the incoming throughput and by its cache hit ratio, as the latter is a reflection of the data size and of the distribution of requests. From such observation, we can use the cache hit ratio and the throughput to build a server usage model, that can then be used to predict the resource utilization of any workload only by knowing those two parameters. In our experiments the average prediction accuracy achieved is 94% with a standard deviation of 5.6. Notably, our approach can be effectively used for several practical applications. Examples are automated online load balancing systems, automated resource allocation and even cost-benefit assessment of hardware upgrades to mention a few. In effect, we are currently implementing this mechanism in an automated elasticity tool (MET [7]) aiming at improving its load balancing capabilities.

Acknowledgment

This work is part-funded by: ERDF - European Regional Development Fund through the Operational Programme for Competitiveness and Internationalization - COMPETE 2020 Programme, and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project POCI-01-0145-FEDER-006961; and project LeanBigData (FP7-619606).

References

1. M. Ahmad and I. T. Bowman. Predicting system performance for multi-tenant database workloads. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, DBTest '11, pages 6:1–6:6, 2011.
2. M. Y. Ahmad and B. Kemme. Compaction management in distributed key-value datastores. *Proc. VLDB Endow.*, 8(8):850–861, Apr. 2015.
3. Apache. Hadoop: `http://hadoop.apache.org/` (2015).
4. 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.
5. H. Che, Y. Tung, and Z. Wang. Hierarchical web caching systems: modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications*, pages 1305–1314, 2002.
6. B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.
7. 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, pages 183–196, 2013.

8. C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 313–324, 2011.
9. P. Desnoyers, T. Wood, P. Shenoy, R. Singh, S. Patil, and H. Vin. Modellus: Automated modeling of complex internet data center applications. *ACM Trans. Web*, pages 1–29, 2012.
10. D. Didona, F. Quaglia, P. Romano, and E. Torre. Enhancing performance prediction robustness by combining analytical modeling and machine learning. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 145–156, 2015.
11. R. A. Fisher. On the probable error of a coefficient of correlation deduced from a small sample. *Metron*, pages 3–32, 1921.
12. L. George. *HBase: The Definitive Guide*. O'Reilly, 2011.
13. Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *International Conference on Network and Service Management*, pages 9–16, 2010.
14. P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, 2010.
15. B. Jennings and R. Stadler. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, pages 1–53, 2014.
16. A. Khan, X. Yan, S. Tao, and N. Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *Network Operations and Management Symposium (NOMS)*, pages 1287–1294, 2012.
17. I. Konstantinou, E. Angelou, D. Tsoumakos, C. Boumpouka, N. Koziris, and S. Sioutas. Tiramola: Elastic nosql provisioning through a cloud management platform. In *International Conference on Management of Data (SIGMOD Demo Track)*, 2012.
18. A. L. and P. M. Cassandra - a decentralized structured storage system. In *LADIS*, 2009.
19. J. Li, A. C. König, V. Narasayya, and S. Chaudhuri. Robust estimation of resource consumption for sql queries using statistical techniques. *Proc. VLDB*, pages 1555–1566, 2012.
20. A. Matsunaga and J. A. B. Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *In proc. of IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID, pages 495–504, 2010.
21. B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and resource modeling in highly-concurrent oltp workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 301–312, 2013.
22. B. Mozafari, C. Curino, and S. Madden. Dbseer: Resource and performance prediction for building a next generation database cloud. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.
23. T. R. Puzak. *Analysis of Cache Replacement-algorithms*. PhD thesis, 1985. AAI8509594.
24. R. Singh, U. Sharma, E. Cecchet, and P. Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proceedings of the 7th International Conference on Autonomic Computing*, pages 21–30, 2010.
25. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, pages 202–208, 1985.
26. S. Sudevalayam and P. Kulkarni. Affinity-aware modeling of cpu usage for provisioning virtualized applications. In *2011 IEEE International Conference on Cloud Computing (CLOUD)*, pages 139–146, 2011.
27. T. Wood, L. Cherkasova, K. Ozonat, and P. Shenoy. Profiling and modeling resource usage of virtualized applications. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 366–387, 2008.
28. Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Proceedings of the 4th International Conference on Autonomic Computing*, pages 27–, 2007.