Autonomic Configuration of HyperDex via Analytical Modelling

Nuno Diegues, Muhammet Orazov, João Paiva, Luís Rodrigues, Paolo Romano
INESC-ID / Instituto Superior Técnico, Universidade de Lisboa
{nmld, muhammet.orazov, joao.paiva, ler, paolo.romano}@tecnico.ulisboa.pt

ABSTRACT

HyperDex is a recent multi-dimensional key-value store that allows efficient search for objects using their secondary attributes. However, the advantage of supporting complex queries comes at the cost of a complex configuration. In this paper we address the problem of automating the configuration of this sort of novel key-value stores. We first show that a misconfiguration may significantly affect the performance of such systems. We then derive a performance model that provides key insights on the behaviour of HyperDex. Based on this model, we derive a technique to automatically and dynamically select the best HyperDex configuration.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management; Systems; Distributed Databases

Keywords

Autonomic Configuration; Key-Value Store; Multi-dimensional; Analytical Modelling; HyperDex

1. INTRODUCTION

Key-value data stores, often so called NoSQL storage systems (in opposition to classic databases), are widely used as a fundamental building block for large scale distributed systems. For scalability and performance reasons, most key-value stores adopt simplistic interfaces, in which objects are only accessible through a single key. BigTable [2], Dynamo [5], and Cassandra [11] are examples of such systems.

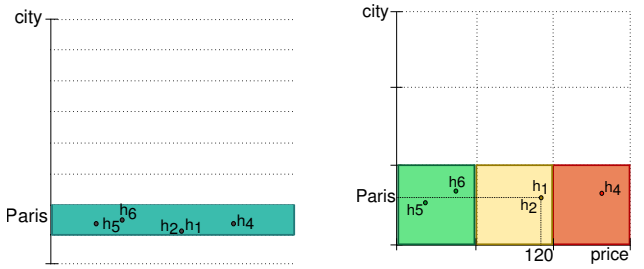
Yet, querying/accessing objects solely by their primary key is rather restrictive. Consider a website for booking hotel rooms: it is easy to conceive that the system must support searches for hotels in a given location and price. It is therefore imperative to support the search for the objects, which represent the hotels, by other attributes rather than their primary keys. Recently, several proposals have used mappings to multi-dimensional spaces both in key-value stores and peer-to-peer systems[7, 8]. Among these, HyperDex [7] owns a unique set of characteristics that makes it a very appealing solution to the problem. The main idea of HyperDex is to use *hyperspace hashing*, an extension of consistent hashing [9]. Briefly, an object with a set of attributes $\mathcal{A}$ is mapped to an Euclidean space with $|\mathcal{A}|$ dimensions (i.e., its cardinality) by hashing the values of its attributes, and interpreting it as a vector of coordinates.

HyperDex provides a rich API with support for searches on any object's attributes, also called partial searches. By leveraging on hyperspace hashing, HyperDex can handle partial searches very efficiently. On the other hand, maintaining indexes does introduce additional costs on the execution of inserts and updates; hence, they should be used wisely. HyperDex allows the programmer to configure the Euclidean space according to the requirements of the target application. Unfortunately, it is far from obvious which configurations provide best results. As we shall see, misconfigurations that are likely to occur with non-expert users may affect drastically the performance of the system, with differences in performance up to 47× (measured in our experiments). One of the key challenges is that the number of possible configurations grows exponentially with the number of attributes considered, making exhaustive testing a tedious or often impossible task. On top of this, the underlying mechanisms and implementation of HyperDex are complex, which makes any attempt to identify the best configuration for each workload a daunting task. This motivates the main claim of this paper, which is to develop techniques that support the auto-configuration of HyperDex.

In this paper we study hyperspace hashing in detail, and in particular the inner-workings of HyperDex, both from an analytical as well as experimental perspectives. We present two contributions with the objective of autonomically maximizing HyperDex's performance for a given workload and deployment setting: 1) a predictive model of HyperDex's performance that obtains an average accuracy of 92%; and 2) an architecture that takes advantage of the previous contribution and allows HyperDex to adapt to the current system workload and self-configure to maximize its performance.

Section 2 presents an in-depth description of HyperDex. Using this knowledge, in Section 3 we derive an analytical model of HyperDex, which is then validated in Section 4. In Section 5, we present the architecture of our solution for automatically configuring HyperDex and evaluate its accuracy against that of a set of heuristics. In Section 6, we overview the related work. Finally, Section 7 concludes the paper.

SAC'14 March 24-28, 2014, Gyeongju, Korea.



(a) Subspace $\langle city \rangle$. (b) Subspace $\langle city, price \rangle$.

Figure 1: Two different configurations.

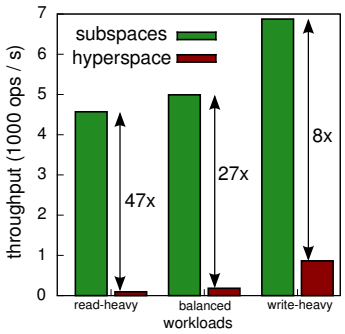


Figure 2: Performance of HyperDex with an hyperspace against a configuration with subspaces.

2. OVERVIEW OF HYPERDEX

One of the main goals of HyperDex is to support efficient partial searches by secondary attributes, mainly by reducing substantially the number of servers involved in each query. The main idea is to use hyperspace hashing, in which the system can deterministically calculate the smallest set of servers that may contain data matching a given query.

2.1 Hyperspace Hashing in HyperDex

Consider that the objects to be stored have $\mathcal{N}$ distinct attributes. An hyperspace in HyperDex is an Euclidean space with $\mathcal{N}$ dimensions, such that each dimension i is associated with an attribute $\mathcal{A}_i \in \{\mathcal{A}_1, ..., \mathcal{A}_\mathcal{N}\}$. Hyperspace hashing maps an object in the hyperspace by applying a hashing function to the value of each attribute $\mathcal{A}_i$ of the object. In this way, we obtain a vector of $\mathcal{N}$ coordinates that correspond to the point in the hyperspace where the object is located. The hyperspace is partitioned in multiple disjoint regions that are assigned to servers. A directory keeps the mapping among regions and servers, such that the right nodes can be contacted when a query or update is executed.

In the description above, we assumed that the key-value store uses a single hyperspace, with as many dimensions as the attributes of the objects. Unfortunately, the volume of an hyperspace grows exponentially with each additional attribute. As a result of this growth, partial searches become increasingly prone to contact regions (and corresponding servers) that contain no relevant data for the search. To address this problem, HyperDex allows the system to be configured using multiple hyperspaces, called *subspaces*, each with a number of dimensions smaller than $\mathcal{N}$.

The possibility of using multiple subspaces increases the complexity of configuring HyperDex: the programmer has to define the set of subspaces (denoted by $\mathcal{S}$), and for each subspace $\mathcal{S}_i \in \mathcal{S}$, which attributes $\langle \mathcal{A}_1, ..., \mathcal{A}_k \rangle$ to be used. This can be illustrated resorting to the example of the hotel database briefly mentioned before. Each hotel is an object with various attributes, such as the primary key (name), category, price, address fields, among others. In Figs. 1a and 1b we show two possible subspaces with the corresponding regions (distributed to servers) and some points representing hotels. Considering a query for hotels in Paris: using the subspace of Fig. 1a it is necessary to contact only 1 region, whereas in Fig. 1b it is necessary to contact 3. If the query also specifies an additional requirement of price 120, only one region is contacted in both cases. Note that, independently of the number of dimensions of a subspace, the strategy adopted in HyperDex is to divide each dimension of a subspace such that the total number of regions per subspace is close to a predefined value $\mathcal{R}$.

In Fig. 2 we present a particularly interesting experiment, illustrating the impact of the configuration of HyperDex on its performance: by configuring it with a single hyperspace, or with the best (albeit complex) combination of subspaces, the performance may be improved from 8× to 47× depending on the ratio of queries and updates. We discuss the reasons underlying this difference in performance when presenting our analytical model in Section 3. Unfortunately, as we shall see, it is not trivial to manually decide on the best configuration, for which reason this process should be automatized. In order to understand how HyperDex can be configured, we need to delve into its operation.

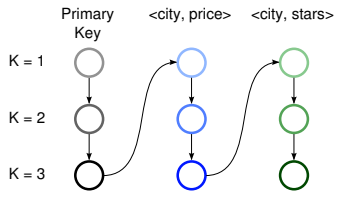
2.2 Search Operation

We first describe how queries are processed in HyperDex. We define a search query $\mathcal{Q}$ as the set of attributes that the query accesses (and respective values). In the general case, to execute a query it is necessary to send a message to the servers responsible for the regions touched by the query. The number of servers contacted varies according to the subspace chosen and the specification (partial, or complete) of the query with regard to the dimensions of the subspace; for instance, in the example Fig. 1b, a search $\mathcal{Q} = \langle city = Paris, price = 120 \rangle$ results in contacting only one region, but in a query for $\mathcal{Q} = \langle city = Paris \rangle$ in the subspace $\langle city, price \rangle$ all three regions are contacted. In order to obtain the best throughput possible, HyperDex always executes a query on the subspace $\mathcal{S}_i \in \mathcal{S}$ which yields the minimum number of regions. Note that HyperDex maintains a full copy of each object in each configured subspace.

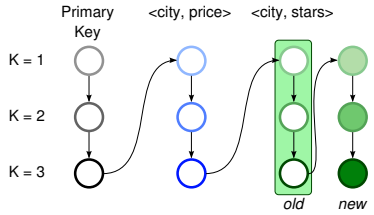
2.3 Update Operation

Finally, we describe how updates are processed in HyperDex. First the object is searched, using a subspace that has the primary key as the single dimension (this subspace must exist in every HyperDex configuration). Then, since a full copy of the object is stored in each sub-space, all copies need to be updated. For fault-tolerance, $\mathcal{K} = f + 1$ copies may be maintained in each subspace.

To coordinate the update, HyperDex uses chain replication [16]. HyperDex organizes the replication chains using a technique called "value-dependent chaining", in which the chain of an object depends on the values of its attributes. Whenever an attribute is updated, the position of the object may change, which causes additional servers to participate



(a) Update of $\langle tel \rangle$.



(b) Update of $\langle stars, tel \rangle$.

Figure 3: The chain of servers resulting from two different update operations in the same configuration of HyperDex.

in the chain. Fig. 3 shows two examples of replication chains for different updates. Consider the update $\mathcal{U} = \langle tel \rangle$, meaning that it changes the telephone of a given hotel, shown in Fig. 3a. In this case 3 replicas have to be updated for each subspace. In Fig. 3b, the update $\mathcal{U} = \langle stars, tel \rangle$ additionally changes the stars of the given hotel. This results in a more complex chain because the attribute *stars* is present in a dimension of one subspace. By changing its value, the hotel changes its position in the subspace $\langle city, stars \rangle$, which may cause it to move from one region (*old*) to another (*new*): the *old* server deletes it from its storage while the *new* one has to insert it (in subsequent operations the replication chain will no longer involve the servers of the *old* sub-chain).

3. MODELLING HYPERDEX

Based on the insights of the previous section on the inner workings of HyperDex, we now derive an analytical model that captures its performance. In the following we assume scenarios with peak throughput, meaning the servers' processors are fully utilized and the network resources are not restraining the performance. In deployments where the network is the bottleneck, the configuration becomes trivial, as the optimal solution consists in using as few servers as possible, based on the storage capacity of the nodes.

3.1 Modelling Searches

In situations of peak throughput, the cost of searching in HyperDex is proportional to the number of regions contacted. Consider a generic search query $\mathcal{Q}_i$.

Observation 1. *The worst possible performance for a search $\mathcal{Q}_i$ happens whenever $\nexists_{\mathcal{S}_i \in \mathcal{S}} : \mathcal{Q}_i \cap \mathcal{S}_i \neq \emptyset$.*

Rationale. Since no subspace contains (at least) one attribute being searched, then the query must contact $\mathcal{R}$ regions (i.e., all) in some subspace. The subspace chosen is irrelevant, because all regions should be evenly split among servers in all subspaces. Hence, $\mathcal{Q}_i$ will be received and processed by all nodes, over all data stored locally, leading to the worst possible performance. □

Observation 2. *Every configuration where $\exists_{\mathcal{S}_i \in \mathcal{S}} : \mathcal{S}_i \subseteq \mathcal{Q}_i$ leads to the optimal performance when searching for $\mathcal{Q}_i$.*

Rationale. Since there are $\mathcal{O}$ objects scattered uniformly among $\mathcal{R}$ regions, then each region contains $\frac{\mathcal{O}}{\mathcal{R}}$ objects. Each attribute in $\mathcal{S}_i$ is also contained in $\mathcal{Q}_i$, meaning that the search defines values for all coordinates of $\mathcal{S}_i$. Consequently, the set of coordinates results in a point in the subspace, which is contained in a single region. Thus the search only contacts one region, whose server processes $\frac{\mathcal{O}}{\mathcal{R}}$ objects. □

Observation 3. *For any subspace $\mathcal{S}_i \in \mathcal{S}$ and search query $\mathcal{Q}_i$, the expected number of contacted regions by $\mathcal{Q}_i$ is:*

$$\mathcal{CR}^{exp}(\mathcal{Q}_i) = \sqrt[|\mathcal{S}_i|]{\mathcal{R}}^{|E|} \quad \text{such that: } E = \mathcal{S}_i \setminus \mathcal{Q}_i \qquad (1)$$

Rationale. The set E represents all the attributes present in subspace $\mathcal{S}_i$ but not defined by the partial search $\mathcal{Q}_i$. For each of those undefined attributes, all the regions along that dimension will be contacted. Generally, to ensure a total number of regions $\mathcal{R}$, each subspace dimension is split in $\sqrt[|\mathcal{S}_i|]{\mathcal{R}}$ partitions. As a result, the number of regions contacted is the product of this number of partitions $|E|$ times, as that is the number of dimensions not defined by the query — they can be seen as extra, or unnecessary for the query. □

We can now estimate the cost of a given search. This is proportional to the product of the number of regions contacted (given by Equation (1)) by the number of objects in each region. To obtain an absolute estimation of throughput we consider a factor β, which is a constant cost associated with processing a single item and dependant on the hardware configuration of the evaluated system. Then, the expected throughput of a search query $\mathcal{Q}_i$ that uses some subspace $\mathcal{S}_i$ is obtained by:

$$T^{exp}(\mathcal{Q}_i) = \frac{1}{cost(\mathcal{Q}_i)} \;,\; cost(\mathcal{Q}_i) = \sqrt[|\mathcal{S}_i|]{\mathcal{R}}^{|E|} \times \frac{\mathcal{O}}{\mathcal{R}} \times \beta \quad (2)$$

We finally consider workloads where there may exist several search queries $\mathcal{Q}$, and each query $\mathcal{Q}_i$ occurs with some likelihood p_i. Naturally, the sum of all probabilities adds to 1. We can then define the query set $\mathcal{Q}^S$ as composed by all $\mathcal{Q}_i$. This way we can predict the throughput of the system through the weighted combination of costs (Equation (2)):

$$T^{exp}(\mathcal{Q}_s) = \frac{1}{\sum_{i=0}^{|\mathcal{Q}^S|} (cost(\mathcal{Q}_i) \times p_i)} \qquad (3)$$

3.2 Modelling Updates

From the description of updates in HyperDex we predict a cost proportional to the length of the replication chain involved in the operation.

Observation 4. *The cost of an update is proportional to the length of the chain replication involved in the operation, i.e., $length(\mathcal{Q}_i) = \mathcal{K}(1 + |\mathcal{N}| + 2|\mathcal{M}|)$.*

Rationale. There is always a part of the chain proportional to the product of the number of subspaces ($|\mathcal{S}|$) and the replication degree ($\mathcal{K}$). It is also necessary to account for the primary key subspace (not included in $\mathcal{S}$). For instance, the length of chain in Fig. 3a is $(1 + |\mathcal{S}|) \times \mathcal{K} = (1 + 2) \times 3 = 9$.

In the general case, we have to admit that attributes of subspaces are modified, as shown in Fig. 3b. In this case there are additional servers in the chain — the subspaces that are modified lead to two sub-chains instead of just one. Thus, we define $\mathcal{S} = \mathcal{N} \cup \mathcal{M}$, where $\mathcal{N} = \{\forall_{\mathcal{S}_i \in \mathcal{S}} : \mathcal{Q}_i \cap \mathcal{S}_i = \emptyset\}$ and $\mathcal{M} = \{\forall_{\mathcal{S}_i \in \mathcal{S}} : \mathcal{Q}_i \cap \mathcal{S}_i \neq \emptyset\}$. □

Yet, this approach considers that every server performs a similar effort. To obtain a precise estimation, we must carefully assess the amount of processing associated with the update.

Observation 5. *The cost of an update has to be weighted by a corrective factor α.*

Rationale. Indeed, there are differences in the processing of an update $\mathcal{Q}_i$, according to whether it changes an attribute mapped to a subspace, or not. Using the example in Fig. 3b, a subspace that is not modified merely needs to update the local copy of the object, using a local OVERWRITE operation. Conversely, a subspace that is modified creates two sub-chains, where the *old* servers must locally invoke a DELETE operation and the *new* servers must invoke a WRITE operation. In fact, we assessed that the OVERWRITE operation is less expensive as this operation never causes the local index to be re-balanced. Consequently, we introduce a corrective factor α to account for this difference. This factor is proportional to the number of subspaces that are modified, i.e., $|\mathcal{M}|$. Similarly to β, this factor α is dependant on the hardware configuration and HyperDex implementation, and must be estimated from a running system. □

As pointed out earlier, an update always conveys a fetch operation to obtain the object (by its primary key), which corresponds to an additional server. Finally, we also consider a parameter T_{max} to capture the maximum throughput achievable by the hardware deployment in study. This parameter can be easily obtained with a scenario where $length(\mathcal{Q}_i) = 1$, e.g. by modifying an object in a simple hyperspace containing only the key subspace:

$$T^{exp}(\mathcal{Q}_i) = \frac{T_{max}}{1 + \mathcal{K}(1 + |\mathcal{N}| + 2\alpha|\mathcal{M}|)} \quad (4)$$

3.3 Modelling Hybrid Workloads

When the workload contains diverse types of operations, the achievable performance can be estimated with a linear combination of the costs of each operation, weighted by its likelihood probability (analogously to Equation (3)).

3.4 Discussion

The important aspects to retain about these models are that the cost of the search operation is proportional to the number of regions matched by the query, whereas the cost of an update increases with the length of the chain involved in the operation. This creates two conflicting forces: on one hand, the number of different regions (and hence of different servers) to query can be decreased by including additional subspaces; conversely, the throughput of updates decreases with an increase on the number of subspaces.

4. ASSESSING THE MODEL ACCURACY

To assess the accuracy of our model, we used a real data set about hotels in the USA. The workload then follows two synthetic patterns representative of a real operating application. Each workload is composed of searches and updates with three variants: read-heavy (RH), with 90% searches and 10% updates; a balanced configuration (BAL); and write-heavy (WH) with 90% updates.

Workload A: Simulates situations where users frequently perform very specific searches. The searches are composed by 4 classes of searches, with increasing probability and with increasing number of attributes specified. There are two updates with equal probability, which alter the two attributes which are neither the most frequent nor the least frequently searched for.

Workload B: Simulates situations where users most frequently perform very broad searches. So, the searches are composed by the same 4 classes of searches as workload A, but with the inverse order of likelihoods, such that the query with a single attribute is the most common one. The updates simulate an environment where one of the attributes is frequently updated (e.g. the "price" attribute), and a set of other attributes is less frequently updated in the same query (e.g. the address, telephone number and zip code).

4.1 Parameter Estimation

Every test was executed with 9 servers in a private cluster, connected through Gigabit Ethernet. The coordinator ran in a dedicated machine, and the other 8 servers processed client requests. We used an environment very similar to that of [7]: one client process in each of the 8 servers, and each client executing 32 threads.

Recall that our model includes three parameters that depend on the hardware configuration. After estimated, these can be used in our model, independently of the workload to be assessed. For space constraints, we only briefly explain the process of estimating the parameters. We used simple scenarios, easily synthesized, to estimate the parameters for our test environment. Naturally, this simplicity has an effect on the final error when applying our model to complex workloads. Yet, for instance, we verified that using only 24 executions (in part repeated), of 2 minutes each, was sufficient to estimate α with only 6.5% error with regard to a perfect estimation.

4.2 Accuracy of the Model

To assess the accuracy of our model, we tested workloads A and B with a sample of all possible configurations given the 4 attributes that are queried and modified. This sample was obtained by ordering all possible configurations according to throughput estimation of our model, selecting the 5 top configurations, and selecting 5 other configurations randomly from the remaining ones. In Figure 4, we show the estimated throughput against the measured throughput for the sampled configurations and for all workloads. Ideally, if the throughputs were all estimated perfectly, all points in the graph would be placed on the diagonal line. Therefore, these results show that our system predicts the performance quite accurately, given that the average error is of only 9% with a standard deviation of 7%.

5. AUTO-CONFIGURING HYPERDEX

In this section we use the model to estimate the best configuration for the given workload. The objective is to have HyperDex autonomically react to the current workload and

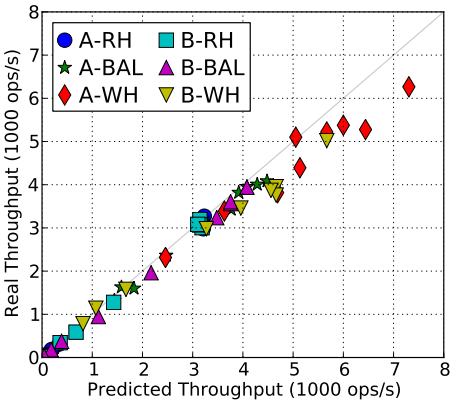


Figure 4: Model accuracy.

self-configure without the intervention of the programmer or the application administrator.

5.1 Architecture of the system

Our solution is composed by three main modules: the *Analyser*, the *Oracle* and the *Configurator*. We briefly describe each of these components below.

- The *Analyser* runs on each server and monitors the system to generate a profile of operations $\mathcal{Q}_i$ and their frequency p_i. The several individual profiles are then aggregated to build the profile of the workload. To allow the system to adapt to changes in the workload profile, the *Analyser* runs periodically, building a new profile for every period.

- The *Oracle* is a centralized component which determines the best configuration for the workload. We consider two different classes of Oracles, namely: i) oracles that work based on heuristics and, ii) an oracle that operates using the analytical model described before.

- The *Configurator* is in charge of applying the changes to HyperDex. This involves changing the hyperspace configuration according to the best configuration derived by the Oracle.

5.2 Oracles

Oracle Based on Heuristics: We consider the following heuristics: *no-subspace*, *hyperspace*, and *dominant*. *No-subspace* is similar to a common key-value store, and provides just a baseline configuration, used for comparison. The *hyperspace* heuristic parses the workload profile and collects all attributes that are currently accessed; it then proposes a configuration that uses all those attributes. The *dominant* heuristic parses the workload profile and picks the most commonly searched attribute; it then proposes a single subspace with that attribute.

Oracle Based on the Analytical Model: This Oracle uses the predictive analytical model described in Section 3 to determine the best configuration for the workload. For that, it generates all possible configurations, queries the model for each of them, and ranks them according to the estimated performance. It then selects the configuration which is ranked highest for that workload. This oracle is labeled as "automatic" in the plots.

5.3 Evaluation

We first compare the performance of the different Oracles. The results in Fig. 5 show how the configuration selected

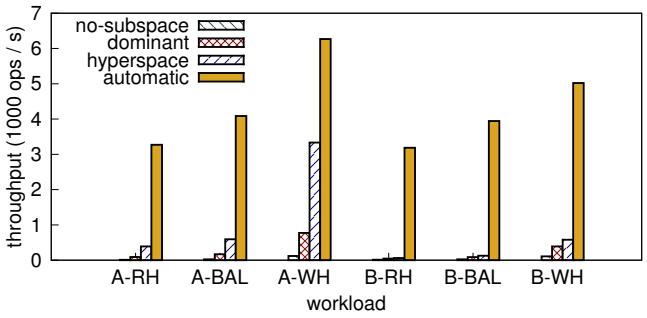


Figure 5: Performance of different oracles.

by our model always results in the best performance. The difference in performance, when comparing with the heuristics, is of one to two orders of magnitude. The complexity of the workloads' queries challenges the *dominant* heuristic performance; still, it is 4.95× better (on average) than *no-subspace*. The *hyperspace* heuristic explores the fact that the workload is highly varied to achieve 2.39× higher throughput than *dominant*. Still, our *automatic* oracle is able to obtain considerable improvements: it is approximately 10× better than *hyperspace*; 25× better than *dominant* subspace; and 126× better than *no-subspace*.

In addition to this, we have conducted further benchmarking to assess the ranking function implemented by the oracle based on the analytical model: Table 1 captures the difference between the ranking of configurations estimated by our model and the ranking that results from executing the configurations in the real system. To measure this difference we use a standard metric relying on Kendall's τ coefficient [10] to assess the agreement between the two rankings. This coefficient varies in the interval $[0, 1]$ where the accuracy is better when closer to 1. The results indicate that there is a high correlation between the rankings predicted by our system and the real rankings. We have also counted the number of pairs of elements which have a different relative ordering in the two rankings, and then multiply this distance by the relative difference in throughput between the two elements. We represent this adjusted distance by $\overline{\tau}$, where it is better to be close to 0 (i.e., the ranking was correctly predicted, or if not, the errors do not affect the throughput). Most ranks have a distance of 0; among the 60 classified configurations, only 4 have $\overline{\tau}$ over 2%; and none is above 14%. Thus, although our model is not perfect in estimating throughputs, the errors do not significantly affect the accuracy of the system. Finally, we highlight that the *automatic* oracle was able to correctly identify the optimal configuration in 5 of the 6 workloads in Figure 5 — to assess this, we had to manually analyze the possible configurations for each workload. In the single case where the *automatic* choice was suboptimal (B-RH), the selected configuration was the second best,

workload	τ coef	avg $\overline{\tau}$ dist	max $\overline{\tau}$ dist
A-RH	0.83	0.012	0.112
A-BAL	0.94	0.002	0.017
A-WH	0.88	0.017	0.139
B-RH	0.72	0.016	0.105
B-BAL	0.94	0.001	0.012
B-WH	0.88	0.008	0.049

Table 1: Difference between estimated and real ranking.

and it only yielded a loss of 6% performance when compared with a perfect prediction.

6. RELATED WORK

Key-Value stores [2, 5, 11] provide highly scalable and performing alternatives to store data [12, 2]. To achieve this, they are typically based on consistent hashing [9]. To provide richer semantics than simple operations based on the key of the object, traditional approaches either flood the network with queries [3], or insert the object multiple times in the system, one for each attribute (or keyword) of the object [13, 1]. Both strategies are particularly inefficient due to the redundancy involved. To reduce the number of servers contacted, other approaches make use of space filling curves [14]. Unlike HyperDex [7], these approaches do not scale with the number of dimensions: the curve becomes increasingly meaningless (hence preserving less and less locality), the more attributes the space has. HyperDex, on the other hand, avoids this problem by creating multiple subspaces, which, as we argue on this paper, must be configured correctly to be taken advantage of.

The idea of generating a predictive model of a key-value store in order to decide on its best configuration is not a new one. Works such as [15, 4, 6] apply this concept to control elastic scaling to adapt to dynamic workloads while avoiding manual configuration. In fact, similarly to our solution, the work by Cruz *et al.* [4] also considers how the data partitioning by nodes affects the throughput of the system. All these works are however directed at auto-configuring elastic scaling on "traditional" key-value stores, whereas ours is aimed at configuring the dimensions on a multi-dimensional one.

7. CONCLUSIONS AND FUTURE WORK

In this work we presented the problem of effectively taking advantage of the new generation of multi-dimensional NoSQL data stores. Subtle changes in the configuration process were shown to lead to drastic performance loss, which motivates the automatization of the process. For that, we claim that using a predictive model provides accurate enough results to allow us to obtain the optimal configuration for a given workload. We have shown that this approach can predict the system throughput with an average accuracy of 92%. In addition to that, we compared it with several (mostly static) heuristics. Our solution yielded improvements of up to two orders of magnitude in the throughput of the system, without requiring any administrator intervention.

Since our model for search queries relies only on the number of regions matched, it is applicable to all multi-dimensional key-value stores based on partitioning spaces composed by several attributes, in particular those based on space-filling curves [14]. On the other hand, predicting the throughput of update operations is tied with the existence of value-dependent chaining and subspaces, concepts that are currently only used on HyperDex, but that we expect to see in many future multi-dimensional key-value stores.

As future work, we intend to improve the accuracy of our throughput estimations by employing more complex techniques such as queue theory to model the effect of concurrent operations in the servers. In order to improve the run-time costs of our system, we also intend to include mechanisms to select configurations based on the cost-benefit ratio between predicted performance and the cost of reconfiguration.

Acknowledgments

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) via INESC-ID multi-annual funding through the PIDDAC Program fund grant, under projects PEst-OE/EEI/LA0021/2013 and CMU-PT/ELE/0030/2009.

8. REFERENCES

[1] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *Proceedings of SIGCOMM*, pages 353–366, 2004.

[2] F. Chang et al. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[3] Y. Chawathe et al. Making gnutella-like p2p systems scalable. In *Proceedings of SIGCOMM*, 2003.

[4] F. Cruz et al. MeT: workload aware elasticity for NoSQL. In *Proceedings of EuroSys*, 2013.

[5] G. DeCandia et al. Dynamo: amazon's highly available key-value store. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, pages 205–220, 2007.

[6] D. Didona, P. Romano, S. Peluso, and F. Quaglia. Transactional auto scaler: elastic scaling of in-memory transactional data grids. In *Proceedings of the International Conference on Autonomic Computing*, ICAC, pages 125–134, 2012.

[7] R. Escriva, B. Wong, and E. Sirer. Hyperdex: a distributed, searchable key-value store. In *Proceedings SIGCOMM*, pages 25–36, 2012.

[8] P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: multi-dimensional queries in p2p systems. In *Proceedings of the Workshop on the Web and Databases*, WebDB, pages 19–24, 2004.

[9] D. Karger et al. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of STOC'97*, 1997.

[10] M. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2), 1938.

[11] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[12] S. Peluso et al. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Proceedings of the International Conference on Distributed Computing Systems*, ICDCS, pages 455–465, 2012.

[13] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of the International Conference on Middleware*, pages 21–40, 2003.

[14] C. Schmidt and M. Parashar. Enabling flexible queries with guarantees in p2p systems. *Internet Computing, IEEE*, 8(3):19–26, 2004.

[15] B. Trushkowsky et al. The scads director: scaling a distributed storage system under stringent performance requirements. In *Proceedings of the Conference on File and Storage Technologies*, FAST, pages 1–12, 2011.

[16] R. van Renesse and F. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of OSDI'04*, 2004.