Optimizing Hyperspace Hashing via Analytical Modelling and Adaptation

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

Hyperspace hashing is a recent multi-dimensional indexing technique for distributed key-value stores that aims at supporting efficient queries using multiple objects' 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 innovative distributed indexing mechanism. We first show that a misconfiguration may significantly affect the performance of the system. We then derive a performance model that provides key insights on the behaviour of hyperspace hashing. Based on this model, we derive a technique to automatically and dynamically select the best configuration.[1]

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance, Experimentation

Keywords

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

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 simplified interfaces, in which objects are only accessible through a single key. BigTable [5], Dynamo [10], and Cassandra [19] 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[15, 16]. Among these, HyperDex [15] 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 [17]. 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 to determine which configurations provide the 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 even 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 goal 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 a system for auto-

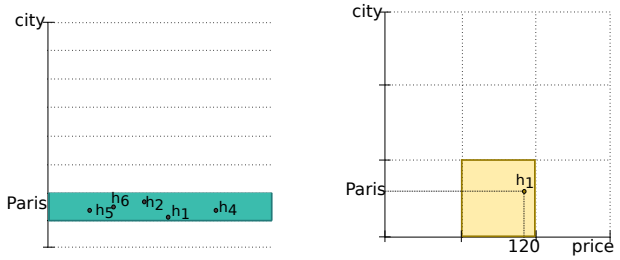
matically configuring HyperDex and we evaluate the accuracy of the implemented algorithms against a set of heuristics. In Section 6, we overview the related work. Finally, Section 7 concludes the paper.

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.

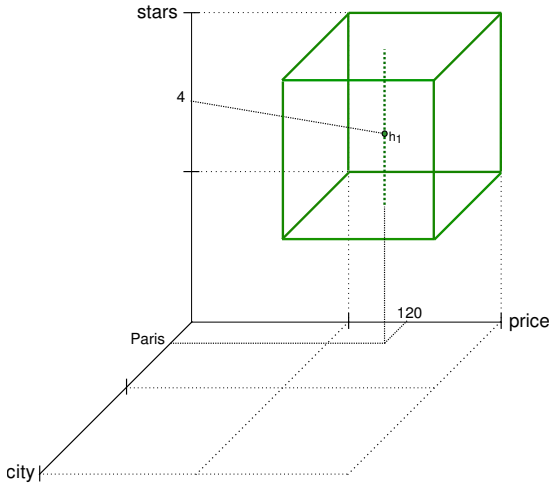
2.1 Hyperspace Hashing in HyperDex

Consider that the objects to be stored have $\mathcal{N}$ distinct attributes. A 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.



(a) Subspace $\langle city \rangle$.

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



(c) Subspace $\langle city, price, stars \rangle$.

Figure 1. Three different configurations and corresponding visualization of a search specifying values for all attributes indexed by the subspace.

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 are increasingly likely 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 hotel database example briefly introduced before. Each hotel is an object with various attributes, such as the primary key (name), category, price, address, among others. In Fig. 1 we show three 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. Furthermore if we consider a three dimensional subspace (see Fig. 1c), we need to specify three attributes in the query to have an efficient operation that contacts only one server. 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 hyperspace hashing on performance: by configuring it to use 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 we argue that

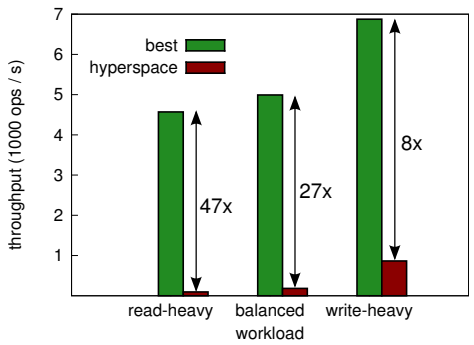


Figure 2. Performance of HyperDex with a single hyperspace against a configuration with subspaces.

this process should be automatized. In order to understand how hyperspace hashing can be configured, we need to first provide additional background on its functioning.

2.2 Search Operation

We first describe how queries are processed using hyperspace hashing. 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 of 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.

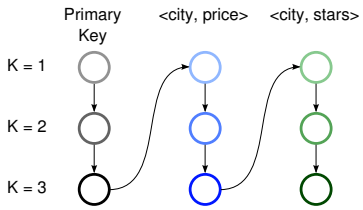
2.3 Update Operation

Updating an object using hyperspace hashing implies modifying all the defined subspaces, as well as an additional subspace that has the primary key as the single dimension (this subspace must exist in every hyperspace hashing configuration). Note that, 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 are maintained in each subspace.

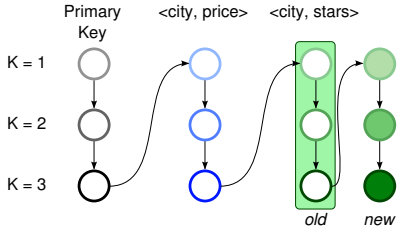
To coordinate the update, HyperDex uses chain replication [27]. HyperDex organizes the replication chains using a technique called "value-dependent chaining", in which the chain of an object depends on the values of its attributes. Whenever an attribute contained by some subspace is updated, the position of the object in that subspace may change, which causes additional servers to participate 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 (as each subspace maintains a full copy of each object). 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 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 (hence, in subsequent operations the replication chain will no longer involve the servers of the *old* sub-chain).

3. MODELLING HYPERSPACE HASHING

Based on the insights provided in the previous section on the inner workings of hyperspace hashing, we now derive an analytical model that captures its performance. In the following we assume scenarios with peak throughput, in which the servers' processors are fully utilized and the network resources are not restraining the performance.



(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.

3.1 Modelling Searches

Under the above assumptions, the cost of searching using hyperspace hashing 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 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)):

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

3.2 Modelling Updates

From the description of updates provided in Section 2.3 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.

Our model considers a typical programming pattern according to which an update is always preceded by a fetch operation to obtain the object (by its primary key). This fetch operation implies contacting 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

In this section we assess the accuracy of our model to predict the performance a given configuration and for a certain workload. We are interested in understanding: 1) the accuracy of the model in correctly predicting the absolute throughput of the system for each configuration; and 2) the ability of the model to correctly rank the performance obtained for each configuration. We note that our intent is to maximize the performance, whichever it is, and hence the most important ability is to correctly rank the configurations so the system may use the best one. Predicting the actual performance is not our focus, although we can also do it as a side-effect of our analytical model.

To assess the previous questions about our model we used a real data set about 140 thousand hotels in the USA. We present the data set and our various workloads to exercise HyperDex in Section 4.1. In Section 4.2 we describe our deployment environment, and in Section 4.3 we discuss how

to estimate the parameters to feed to our model, and the impact of this estimation. Finally, in Section 4.4 we evaluate the accuracy of our model.

4.1 Workload Characterization

Our data set contains information about 140 thousand hotels in the USA. Each one is identified uniquely by an 8 byte key. However, as motivated earlier in the paper, most users of such a data set will search for other, more meaningful, attributes such as the location or the price range of the hotel. For each hotel we have 44 such attributes: 11 strings, 15 booleans, 8 numbers and 10 enumerates. Examples of these range from data about the location and name of the hotel, their characteristics (accepting pets, swimming pools, etc.), to their price and reviews' ratings. This data set suits naturally our problem as typical booking websites allow users to perform queries by numerous attributes such as those presented. It is highly desirable to have a system such as HyperDex that allows efficient indexing and search along attributes other than the primary key of the storage.

To simulate an application using this data set, we devised a series of workloads. On one hand we seek to exercise contrasting scenarios, those that favour the indexing abilities of hyperspace hashing, as well as those that stress its chain replication and duplication of data due to many subspaces. On the other hand, we also desire to perform these queries close to what would be deemed as a realistic workload.

As such, we created workloads A and B, which are quite constrained and allow us to reason more carefully as to what mechanisms of hyperspace hashing will be exercised more thoroughly with each workload. These workloads are contrasting in the sense that they perform either very specific searches, or very broad ones; as explained in Section 3.1, this difference has a dramatic impact on the resulting configurations. On top of this, we devised workload C with a large mix of operations that resemble those typically performed on hotel booking websites.

The detailed profile of our workloads is shown in Table 1. We further describe these workloads in the following:

Updates: The update operations are common to all workloads. The two attributes that are most likely to be updated more often (namely *price* and *ratings*), which are neither the most frequent nor the least frequently searched for, may be written with equal probability by an update operation. This simplifies the understanding of the workloads and allows us to create more complex scenarios on the searches, which are the main driving force of the complexities underlying the challenges addressed in this paper.

Workload A: This workload simulates scenarios where users frequently perform very specific searches. The searches are composed by 4 different classes of searches, with increasing probability of having an added number of attributes specified. As such, the most popular search queries are those with numerous attributes, which allow to specify subspaces with a low dimensionality and have most searches fully specified.

Workload B: Simulates situations where users most frequently perform very broad searches. As such, the search operations are composed by the same 4 classes of workload A, but with the inverse order of likelihoods, such that the query with a single attribute is the most common one.

Table 1. Profile of operations according to the workload. Note that the update operations are common to all workloads. The frequency of each operation is calculated among the operations of each type, i.e., all searches amount to 100% within the workload.

	Operation Details		
Workload	Freq(%)	Type	Attributes
All	50	U	⟨ *price* ⟩
	50	U	⟨ *ratings* ⟩
A	6	S	⟨ *locality* ⟩
	13	S	⟨ *region*, *price* ⟩
	26	S	⟨ *locality*, *ratings*, *price* ⟩
	55	S	⟨ *locality*, *price*, *pets*, *pool* ⟩
B	55	S	⟨ *locality* ⟩
	26	S	⟨ *region*, *price* ⟩
	13	S	⟨ *locality*, *ratings*, *price* ⟩
	6	S	⟨ *locality*, *price*, *pets*, *pool* ⟩
C	20	S	⟨ *locality*, *price* ⟩
	20	S	⟨ *locality*, *ratings* ⟩
	20	S	⟨ *locality*, *price*, *ratings* ⟩
	10	S	⟨ *postcode*, *price* ⟩
	10	S	⟨ *postcode*, *ratings* ⟩
	10	S	⟨ *postcode*, *price*, *ratings* ⟩
	2	S	⟨ *locality*, *stars*, *ratings* ⟩
	2	S	⟨ *locality*, *stars*, *price* ⟩
	1	S	⟨ *locality*, *price*, *category* ⟩
	1	S	⟨ *locality*, *ratings*, *category*⟩
	1	S	⟨ *postcode*, *stars*, *ratings* ⟩
	1	S	⟨ *postcode*, *stars*, *price* ⟩
	1	S	⟨ *region*, *price* ⟩
	1	S	⟨ *region*, *ratings* ⟩

Workload C: In this workload we seek to build a more challenging pattern of utilization. This is obtained by crafting a larger set of search operations with a reasonable combination of attributes commonly used to search for hotels in booking websites. Naturally, the large combination of attributes and their heterogeneous frequency makes it extremely hard to even predict coarsely what kind of strategy is most adequate for this workload. Therefore its purpose is to complement workloads A and B, which were devised purposely to exercise contrasting driving forces, and which in this workload are mixed in a non-trivial way.

Finally, we consider 3 patterns of utilization of each workload, that lead to variants of each one: read-heavy (RH), with 90% searches and 10% updates; a balanced configuration (BAL) with 50% updates and other 50% searches; and a write-heavy (WH) variant with 90% updates and the rest for searches.

4.2 Experimental Setup

For all our hardware deployments we used 9 virtual machines running on top of OpenStack, a widely used open-source virtualization infrastructure that relies on the Xen hypervisor. Each actual underlying machine is equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) E5506 processors and 40GB RAM, running Linux 2.6.32-33-server and interconnected via a private Gigabit ethernet. Each virtual

machine was mapped to a different machine in the same cluster and used all 8 cores and up to 16GB of RAM.

HyperDex requires a coordinator service that is responsible for providing a centralized consistent view of the cluster members at any time. For that, we used one dedicated machine, whereas the other 8 servers executed the HyperDex daemon that serves requests. We followed the same testing environment of the authors of HyperDex [15] by deploying 1 client process in each of the 8 servers, with each client executing 32 threads issuing requests without think time (but blocking until the full answer was received from the server).

4.3 Parameter Estimation

Recall that our model includes three parameters that depend on the hardware configuration. Once estimated, these can be used in our model, independently of the workload to be assessed.

We now assess the impact of possible errors in estimating these parameters on the final accuracy of our predictive model. For this, we describe our approach by using the example of the α parameter required for estimating the performance of update operations; analogously conclusions can be similarly derived using an equivalent procedure for the other parameters.

We propose to use simple scenarios to experimentally assess these hardware-dependent parameters. For α, this can be assessed with a micro-application configured with different (yet simple) subspaces, and a workload that repeatedly invokes updates. These workloads can be synthetically created in development time and executed on the target hardware deployment. Then, this data can be used to estimate α via a re-organization of Equation (4) as follows:

$$\alpha = \frac{T_{max} - T_{real} - |\mathcal{N}|\mathcal{K}T_{max}}{2|\mathcal{M}|\mathcal{K}T_{real}} \quad (5)$$

As an example, we used 24 such simple executions to derive α in our environment. These ran for 2 minutes each, and represent 8 simple cases varying 3 different replication degrees (the value of $\mathcal{K}$).

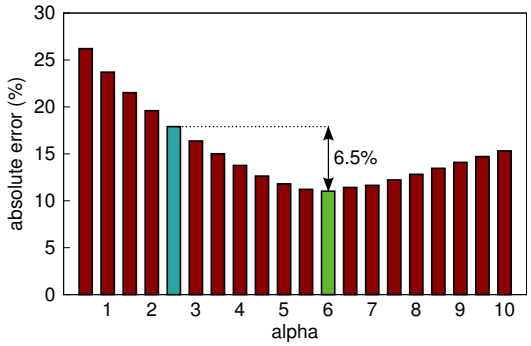
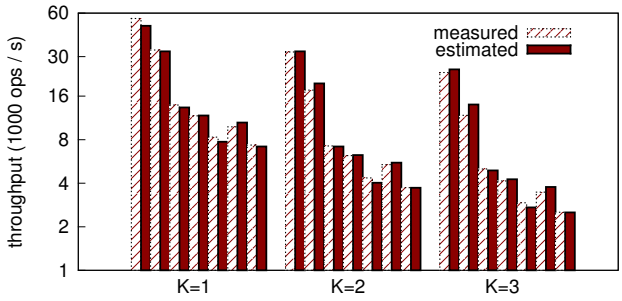
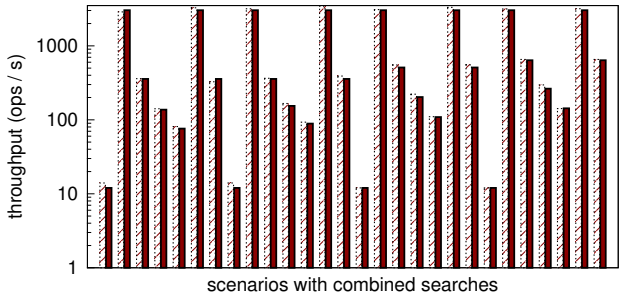


Figure 4. Error in our final estimation in workloads A and B when varying the α parameter in the predictive model. We highlight the α estimated by our micro-application and the one that would have been ideal for the workloads we experiment with in the end.



(a) Experiments with updates (7) and varying $\mathcal{K}$.



(b) Experiments with searches (28).

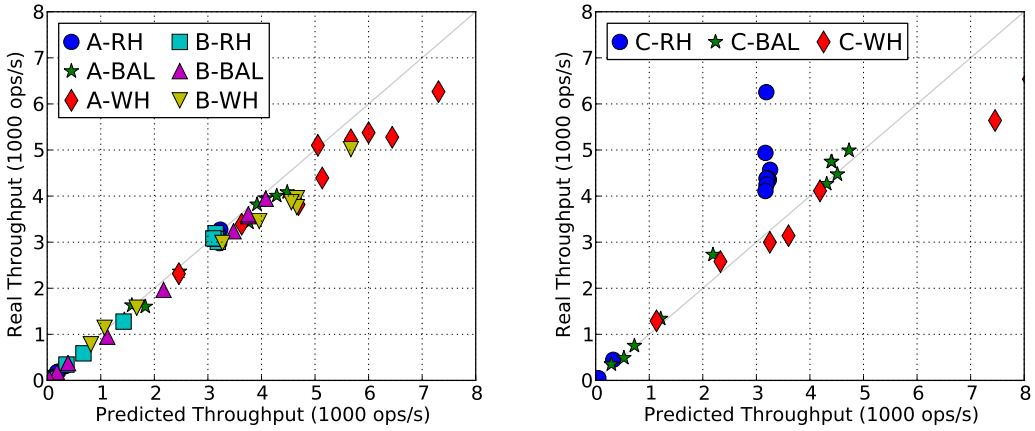
Figure 5. Comparison of estimation vs measured throughput for varying configurations in simple workloads.

Naturally, this simplicity has an effect on the final error when applying our predictive model to complex workloads. Among our micro-benchmarks the value of α that minimized the error in those tests was 2.3. However, this value may not be optimal and lead to error in predicting real workloads. We looked into this by running our prediction against the variants for workloads A and B with varying values for α. In Fig. 4 we show the final error from our prediction to the actual measured throughput according to the α used, and highlight the value we estimated from the micro-benchmarks, as well as the optimal value that would have been ideal to these actual workloads. In fact, our estimation for α is not the best one, but the impact is only of 6.5% in the error. As we shall also see in our evaluation, our gross (and easy to deploy) technique for estimating the hardware-dependent parameters of the model have a reduced impact in the final goal of our work.

4.4 Accuracy of the Model

To assess the accuracy of the proposed model, we shall first run a series of simple workloads that request either search or updates alone while using different configurations. We then assess further our accuracy by considering more realistic workloads such as those presented in Section 4.1 where many different operations are issued concurrently and with different probabilities.

In Fig. 5 we show the comparison of the model's estimated throughput against actual measurements. For completeness, we refer to a longer report [20] that includes the extensive details of combinations of subspace configurations and query-sets that were tested; the idea was to try to isolate different parts of our model, and hence we strived for



(a) Workloads A and B

(b) Workload C

Figure 6. Accuracy of our predictive model for workloads A, B and C (and their variants).

simplicity in the creation of these scenarios — we opted for creating many of them instead, and delegating the assessment of complex/more realistic scenarios to the experimental evaluation presented next.

In this case, we independently assessed the accuracy of our search and updates modelling respectively in Fig. 5b and Fig. 5a. For the updates, we additionally varied the replication degree for fault tolerance ($\mathcal{K}$) because this has an impact in the performance of the system even when nothing else changes — this is visible in Fig. 5a as performance gradually decreases as $\mathcal{K}$ increases. This, however, has no impact in search operations, for which reason we omit evaluating the impact of variations of this parameter, and show instead a broader range of scenarios in Fig. 5b.

Overall, we can see that our model is able to correctly estimate the throughput for all scenarios, and independently of the operation type. In particular, for search operations, we obtain an average absolute estimation error of 5.28% and a standard deviation for the error of 3%. These results attest the accuracy of the linear combination of costs for the search queries employed in Equation (3), as well as the modelling of searches alone. Furthermore, performance of updates operations was modelled with an average error of 4.9% and a standard deviation of 2.9%.

Finally, we also tested with workloads A, B and C, which are representative of more complex scenarios. For this, we used a sample of all possible configurations given the 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. This gives us 10 possible configurations for each workload and respective variant. As we have 9 workloads/variants, we are testing a total of 90 scenarios.

In Fig. 6a, we show the estimated throughput against the measured throughput for the sampled configurations and for workloads A and B. 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 only 9% with a standard deviation of 7%.

In Fig. 6b we show instead the accuracy of our model for the more complex workload C. Given the higher unpredictability of this workload, our predictive model is expected to be slightly more challenged than for workloads A and B. In fact, the overall average error for all variants of this workload is 17% with a standard deviation of 12%, less than double than that of the simpler workloads. This is an interestingly low error given the high complexity of the workload. We may also observe in 6b that most of this error takes place in the read-heavy variant, which is the most unpredictable variant given that it contains the most diverse set of queries.

5. AUTO-CONFIGURING HYPERDEX

In this section we use the predictive model to estimate the best configuration for the given workload. The objective is to obtain the best performing configuration of hyperspace hashing according to the current workload and to self-configure without the intervention of the programmer or the application administrator. Not only is this a cumbersome task; with complex/realistic workloads such as those evaluated here, the spectrum of possible configurations can grow so large that it is simply infeasible to reason on or empirically test the whole set of alternative configurations.

We begin by presenting, in Section 5.1, a self-tuning architecture that takes advantage of our predictive model, previously presented and assessed in this paper. We then describe different oracles, in Section 5.2 that we compare in our evaluation in Section 5.3.

5.1 Architecture of the system

Our architecture for automatic configuration is illustrated in Fig. 7. We begin by intercepting the requests, which clients perform to HyperDex, in order to log them in our *Profiler*. The objective of this module is to collect enough statistics over time to generate workload profiles representative of the current usage of the HyperDex deployment. The *Profiler* 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 global profile of the workload.

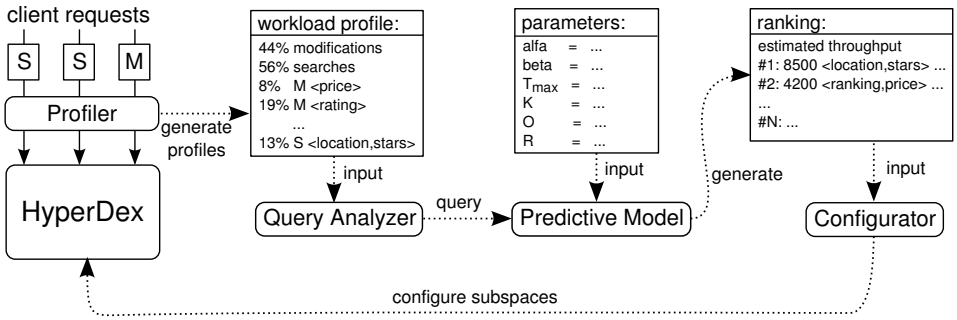


Figure 7. Architecture of the system to automatically configure HyperDex based on our predictive model.

We then have three main components, whose interactions we describe below:

- The *Query Analyser* receives the workload profiles that were logged by the *Profiler* and computes a set of attributes for the underlying data set. With that information, it generates their power set, and then performs combinations of these to obtain all possible sets of configurations of subspaces for HyperDex. To allow the system to adapt to changes in the workload profile, the *Query Analyser* runs periodically, building a new profile for every period.
- The *Predictive Model* consists of our equations that estimate the throughput of the system. For this, we need to feed the equations with some parameters that were previously estimated for the hardware deployment in particular. As a result, we can query the model for each possible configuration and obtain its estimation. Therefore this produces a ranking for the configurations according to their estimation.
- Finally, 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 ranking.

In short, our algorithm works in three phases, which are described in the following. The first phase generates all possible combinations of subspaces based on the set of attributes that was provided by the profiler. In the second phase the throughput of each of the considered configurations is estimated using the model we provided in this paper. Finally, in the last phase all the configurations performing similarly (within a user tunable threshold) are clustered, and ordered in decreasing order of performance. Finally, the most efficient configuration is picked and deployed into the system.

5.2 Oracles

Oracle Based on Heuristics: We consider the following heuristics:

- *no-subspace* is similar to a common key-value store, and provides just a baseline configuration, used for comparison. As such search operations are very inefficient because they need to span all machines of the cluster (if we exclude the replication degree for fault tolerance).
- *hyperspace* heuristic parses the workload profile and collects all attributes that are currently accessed; it then proposes a configuration that uses all those attributes.
- *subspaces-all* in which a subspace is configured with one dimension for each attribute found in the searches of the workload.
- *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 labelled as "automatic" in the plots.

5.3 Evaluation

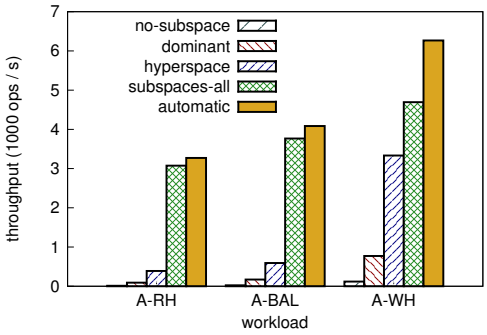
We now first compare, in Section 5.4, the performance of HyperDex when configured via the different Oracles presented. Then, in Section 5.5, we evaluate how often our automatic approach actually finds the optimal configuration.

5.4 Comparison between Oracles

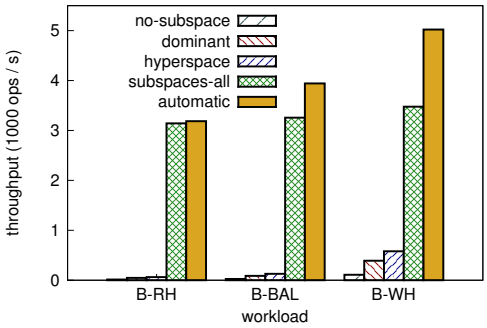
We first compare the performance of the different Oracles in Fig. 8. There, we show the three different workloads presented in Section 4.1, and the corresponding performance when using each oracle for all the variants of the workloads.

In every case we highlight that our automatic configuration results in the best performance. The difference in performance, when comparing with the heuristics, is up to one to two orders of magnitude. Nevertheless, the heuristic oracles should not be neglected: even the approach with *dominant*, for instance, is on average 5× better than *no-subspace*. Still, there is a large room for improvement to explore, as our approach unveils.

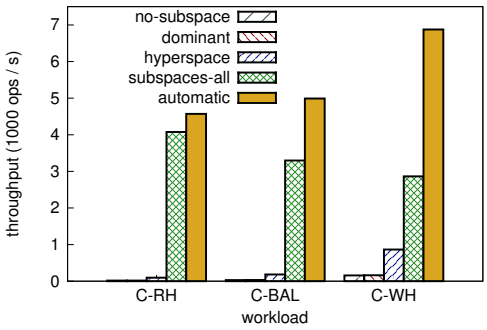
If we look at the *hyperspace* heuristic, instead, it explores the fact that the workloads are highly varied and achieve 2.4× higher throughput than *dominant*. However, we can see that it is *subspaces-all* that fares best among the heuristics. This happens because it creates many subspaces, which increases the likelihood that every search operation will have



(a) Workload A.



(b) Workload B.



(c) Workload C.

Figure 8. Performance of the different oracles in various workloads.

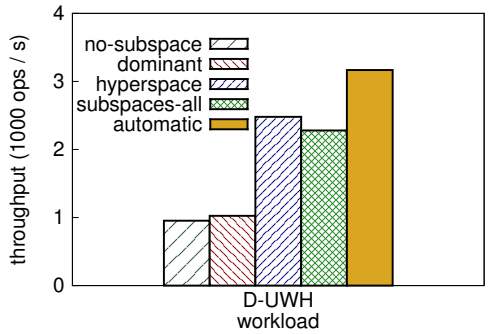


Figure 9. Performance of the oracles in a very write-intensive workload.

Table 2. Difference between estimated and real ranking.

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
C-RH	0.37	0.226	0.460
C-BAL	0.91	0.012	0.115
C-WH	0.75	0.001	0.012

a very efficient way to be performed. Still, we can see that our system can achieve gains up to 31% over that heuristic.

The main disadvantage of *subspaces-all* is that it also duplicates a lot of data, which may be infeasible to have, depending on our hardware deployment. Still, even if we can allow that, this has a downside for the performance of updates. To highlight this, we show in Fig. 9 a workload with 99% update operations. In that case, we can see that our approach is 40% better than *subspaces-all*, which is even surpassed by the *hyperspace* approach because the latter creates only one subspace, which favors updates as these need to possibly change every subspace that exists.

Overall we believe that identifying and quantifying the performance of the oracles is in itself a relevant contribution. Still, if one wants to predict performance in absolute terms, for instance to calculate whether service level agreements will be fulfilled, our system can provide that information quite accurately as shown previously.

5.5 Finding the Optimal Configuration

Finally, we have conducted further benchmarking to assess the ranking function implemented by the oracle based on the analytical model. This is important to understand whether our algorithm is actually trying to estimate the performance of the best configurations. In other words, is our automatic approach considering and correctly predicting the performance for the optimal configuration?

To answer this question, we present Table 2 where we capture the difference between the ranking of configurations estimated by our model and the ranking that results from executing each configurations in a real system. To measure this difference we use a standard metric relying on Kendall's τ coefficient [18], which expresses the agreement between two rankings. This coefficient varies in the interval $[0, 1]$ where the accuracy is better when closer to 1. The results in Table 2 indicate that there is a high correlation between the rankings predicted by our system and the real rankings, since most results are above 0.70.

Kendall's τ coefficient is not expressive enough to capture a subtlety of the rankings produced by our system. In fact, while our system may render a ranking different from that of the real measurements, this is typically caused by the predicted values being close to each other. This means that in practice, the effect of choosing one option over the other is very limited. To better express this measurement, we have also counted the number of pairs of elements which have

a different relative ordering in the two rankings, and then multiply Kendall's τ coefficient 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).

As we can see in Table 2, most ranks have a distance of 0; among the 60 classified configurations for workloads A and B, only 4 have $\overline{\tau}$ over 2%; and none is above 14%. Thus, although our model is not perfect in estimating absolute throughputs, the errors do not significantly affect the accuracy of the system.

As mentioned in Section 4, for the read-heavy variant of workload C, our model cannot accurately predict the throughput of some configurations, which explains the poor ordering reflected in Table 2. Finally, we highlight that the *automatic* oracle was able to correctly identify the optimal configuration in 8 of the 9 workloads in Fig. 8 — to assess this, we had to manually analyze the possible configurations for each workload. In fact, for workload B, the case where the *automatic* choice was suboptimal (B-RH), the selected configuration was the second best, and it only yielded a loss of 6% when compared with a perfect prediction.

6. RELATED WORK

Key-Value stores [5, 10, 19] provide highly scalable and performing alternatives to classic relational DBMS [5]. To achieve this, they are typically based on consistent hashing [17]. To provide richer semantics than simple operations based on the key of the object, traditional approaches either flood the network with queries [6], or insert the object multiple times in the system, one for each attribute (or keyword) of the object [22, 4]. 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 [23]. Unlike hyperspace hashing,[15], 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. Hyperspace hashing, 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.

Other related approaches [9, 21, 30] aim at maximizing efficiency of distributed data stores by relying on autonomic techniques to enhance data locality and load balancing. To this end, these approaches transparently perform a detailed characterization of the data access patterns generated by applications, and accordingly derive optimized data placement strategies in an automatic fashion. These techniques assume that the underlying data store adopts a mono-dimensional hash-based data distribution function [17], and therefore cannot be straightforwardly applied to hyperspace hashing. On the other hand, since hyperspace hashing, just like consistent hashing, distributes data across nodes in a random way, the resulting placement may be suboptimal given the locality patterns exhibited by applications. We argue that extending these techniques to operate in synergy with hyperspace hashing would be an interesting research avenue, which might lead to further improve efficiency of hyperspace hashing.

Alternative approaches to support efficient queries over secondary attributes in distributed key-value stores rely on scalable solutions to build and update indexes that are distributed over a large set of machines. Unlike hyperspace hashing, these techniques do not determine the placement of replicas of data using multidimensional hashing schemes. Conversely, these solutions assume that the placement of data is governed by an orthogonal placement policy (e.g., consistent hashing) and build indexes over secondary attributes using distributed tree-like data structures [2, 28, 25, 14]. Despite being designed to maximize efficiency and scalability, also these approaches clearly incur costs to maintain and query distributed indexes, which vary also depending on the consistency semantics that they ensure (ranging from eventual consistency [10] to classic 1-copy serializability [3] and including intermediate consistency semantics [1]). Probably due to the recency of these mechanisms, to the best of our knowledge, no performance models for these distributed indexing solutions have appeared in the literature, yet.

On the other hand, the idea of generating a predictive model of the performance a key-value store in order to decide on its best configuration is not a new one. Works such as [26, 8, 13] 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.* [8] 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.

Finally, our work is also related to the vast literature in the area of performance modelling of (distributed) database systems, which include a large number of approaches based on queuing theory [31, 7, 11], and, more recently, on black-box machine learning methodologies [24, 29, 12].

7. CONCLUSIONS AND FUTURE WORK

In this work we address the problem of how to effectively configuring a recently proposed technique for indexing data in distributed NoSQL data stores, namely hyperspace hashing. Subtle changes in the configuration process were shown to lead to drastic performance losses, motivating the need to automate 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 [23]. 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, CMU-PT/ELE/0030/2009 and PEPITA (PTDC/EEI-SCR/2776/2012).

8. REFERENCES

[1] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, 1999. AAI0800775.

[2] M. K. Aguilera, W. Golab, and M. A. Shah. A Practical Scalable Distributed B-tree. *Journal Proc. VLDB Endowment*, 1(1):598–609, Aug. 2008.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

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

[5] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[6] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of SIGCOMM*, 2003.

[7] B. Ciciani, D. M. Dias, and P. S. Yu. Analysis of replication in distributed database systems. *IEEE Transactions on Knowledge and Data Engineering*, 2(2), 1990.

[8] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. Paulo, J. Pereira, and R. Vilaça. MeT: workload aware elasticity for NoSQL. In *Proceedings of EuroSys*, 2013.

[9] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. In *Proc. of the 36th VLDB*, Singapore, Sept. 2010.

[10] 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 *Proceedings of the Symposium on Operating Systems Principles*, SOSP, pages 205–220, 2007.

[11] P. Di Sanzo, B. Ciciani, F. Quaglia, and P. Romano. A performance model of multi-version concurrency control. In *Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE International Symposium on*, pages 1–10, Sept 2008.

[12] P. Di Sanzo, D. Rughetti, B. Ciciani, and F. Quaglia. Auto-tuning of cloud-based in-memory transactional data grids via machine learning. In *Proceedings of the 2012 Second Symposium on Network Cloud Computing and Applications*, NCCA '12, pages 9–16, Washington, DC, USA, 2012. IEEE Computer Society.

[13] 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.

[14] N. Diegues and P. Romano. STI-BT: A Scalable Transactional Index. In *Proc. International Conference on Distributed Computing Systems (ICDCS)*, 2014.

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

[16] 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.

[17] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of STOC'97*, 1997.

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

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

[20] M. Orazov. Data locality aware partitioning schemes for large-scale data stores. Master's thesis, Instituto Superior Tecnico, Universidade de Lisboa, July 2013.

[21] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues. AutoPlacer: scalable self-tuning data placement in distributed key-value stores. In *Proceedings of the 10th International Conference on Autonomic Computing*, ICAC'13, San Jose, CA, USA, June 2013. USENIX.

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

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

[24] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proc. of the International conference on Autonomic computing (ICAC)*, 2010.

[25] B. Sowell, W. Golab, and M. A. Shah. Minuet: A Scalable Distributed Multiversion B-tree. *Journal Proc. VLDB Endowment*, 5(9):884–895, May 2012.

[26] B. Trushkowsky, P. Bodík, A. Fox, M. Franklin, M. Jordan, and D. Patterson. 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.

[27] R. van Renesse and F. Schneider. Chain replication for supporting high throughput and availability. In

Proceedings of OSDI'04, 2004.

[28] S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu. Efficient B-tree Based Indexing for Cloud Data Processing. *Journal Proc. VLDB Endowment*, 3(1-2):1207–1218, Sept. 2010.

[29] J. Xu, M. Zhao, J. A. B. Fortes, R. Carpenter, and M. S. Yousif. On the use of fuzzy modeling in virtualized data center management. In *Proc. of the International conference on Autonomic computing (ICAC)*, 2007.

[30] G.-W. You, S.-W. Hwang, and N. Jain. Scalable load balancing in cluster storage systems. In *Proc. of the 12th Middleware*, Lisbon, Portugal, 2011.

[31] P. S. Yu, D. M. Dias, and S. S. Lavenberg. On the analytical modeling of database concurrency control. *J. ACM*, 40(4):831–872, Sept. 1993.