Data locality aware partitioning schemes for large-scale data stores

(extended abstract of the MSc dissertation)

Muhammet Orazov
Dissertation submitted in partial fulfillment of the requirements for
the European Master in Distributed Computing Programme
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

***Abstract*—Key-value stores are widely recognized as scalable systems with good performance, and are the backbone of several large-scale storage deployments. However, their interface is rather restrictive since it only allows to access objects through their keys. To address this problem, recently proposed systems have developed mechanisms for storing data by mapping it into multiple dimensions, in order to allow searching for objects using their secondary attributes. These solutions, however, pose another serious problem: that of configuring the system such that it may take the best advantage of these multi-dimensional mappings.**

This work makes two main contributions on configuring such multi-dimensional key-value stores: First, from a detailed description of the inner workings of how operations are mapped to multiple dimensions, we derive a model which describes the behaviour of operations in these systems. We then use this model to predict real throughputs of the system for complex workloads. We also contribute with a generic architecture which allows to automatically adapt the configuration of the multiple dimensions in order to obtain the maximum possible throughput for a given workload.

I. Introduction

The ongoing trend towards key-value stores has been driven mostly by a concern with scalability and performance. These systems typically adopt simplistic interfaces, allowing objects to be accessed only by a given key. This simplification has led to a new generation of systems significantly faster and more scalable than classic distributed databases. This generation has been materialized in known systems such as BigTable [1], Dynamo [2], and Cassandra [3].

Yet, accessing an object solely by a single key is rather restrictive. Consider a website for booking hotel rooms to understand the limitation. It is easy to conceive that the system must support searches for hotels in a given location and within a reasonable price that the customer is willing to pay. Therefore, it is imperative to obtain the objects, which represent the hotels, by other attributes rather than their primary keys.

Solutions to this problem have mostly suggested creating indexes on top of the underlying key-value store [4], [5], [6]. This naturally entails some overheads in the normal execution of the system, which remains agnostic of the concern to locate objects through secondary attributes. Alternatively, other approaches have explored multi-dimensional mappings built-in the key-value stores and peer-to-peer systems [7], [8], [9], [10]. Among these, HyperDex [8] has gathered a unique set of characteristics that makes it a very appealing solution to the problem.

The main idea of HyperDex is *hyperspace hashing*, which behaves similarly to consistent hashing techniques [11], [12], [13]. Briefly, an object with a set of attributes $\mathcal{A}$ can be mapped to an Euclidean space with $|\mathcal{A}|$ dimensions (i.e., its cardinality). By hashing the values the attributes of an object, one can find the coordinates in that space where the object lies. Then sets of points in the space can be assigned to servers, effectively sharding the data.

Using the idea, HyperDex provides a rich API with support for range queries and partial searches on any set of attributes composing objects. This promising system is meant to solve the long-existing trade-off between efficiency of partial searches without exacerbating the cost of insertions and modifications. However, it is still up to the programmer to define the configuration of subspaces that best suits his application. This poses a limitation on the throughput of HyperDex, since, as we shall see in this thesis, this configuration has a strong effect in the performance of the system. In our experiments, differences in configuration result in differences of up to 47 times in throughput.

The task of selecting the optimal configuration is far from trivial. On one hand, the number of possible configurations grows exponentially with the number of attributes considered, making exhaustive testing a tedious or possibly impossible task. On the other hand, to the best of our knowledge, there exists no method for off-line predicting how different configurations affect the performance of the system in order to avoid such testing. Hence, selecting the best configuration requires the programmer to have a strong knowledge of the inner workings of HyperDex and hyperspace hashing.

In this work we study hyperspace hashing in detail, and in particular HyperDex, both from an analytical and experimental perspectives. The objective is to predict the performance of HyperDex in a given application and running environment. As a result, we can then assist the programmer in optimizing the configuration of hyperspaces to improve the performance of the system. To achieve this goal, this

work makes two main contributions: a) a predictive model of the performance of HyperDex, which allows to rank configurations for a given workload with up to 92% of accuracy; and b) a generic architecture for automatically configuring HyperDex to adapt to any given workload in order to maximize throughput.

The remaining of the document is structured as follows. In Section II, we briefly overview the related work. Section III presents an in-depth description of HyperDex and the trade-offs tied to its different possible configurations. Using this knowledge, in Section IV we derive a model of HyperDex, which is then validated in Section V. In Section VI, we present the architecture of our solution for automatically configure HyperDex and evaluate its performance against that of static heuristics. Finally, Section VII concludes the document.

II. Related Work

Key-Value stores [1], [2], [3] provide high-performance alternatives to store data. As shown in [14], [15], [1], their throughput can scale linearly with the deployment size, making them particularly well suited for large-scale deployments. To achieve this scalability, these systems are typically based on consistent hashing [11], [12], [13], which can efficiently partition data by the system nodes, simply by hashing the objects' keys and node identifiers.

To provide richer semantics than simply operations based on the key of the object, some approaches either flood the network with queries [16], [17], or insert the object multiple times in the system, one for each attribute (or keyword) of the object [18], [19], [20]. Both strategies are particularly inefficient either due to the network usage or redundancy involved. To be able to deterministically determine which nodes own a given object and contact as few nodes as possible, other approaches make use of space filling curves [7], [21], [22], [10]. This technique consists in mapping a multi-dimensional space to a uni-dimensional line which is then used as a ring in consistent hashing while preserving the locality of the multi-dimensional space. However, unlike HyperDex [8], this approach suffers from the curse of dimensionality: 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 work, 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 [23], [24], [25] 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.* [24] 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.

III. The HyperDex System

One of the main goals of HyperDex is to search by items using their secondary attributes without building secondary indexes or contacting all system nodes. The main idea is that using Hyperspace Hashing, the system can deterministically determine the smallest set of nodes which may contain data matching a given query. This allows the client to send the query only to the relevant nodes, which process it locally and return the matched objects.

A. Hyperspace Hashing

Hyperspaces are the building block of hyperspace hashing, which in its turn is at the core of HyperDex. Generally, an hyperspace is an N-dimensional Euclidean space where each dimension encompasses any possible value output by a hashing function. A particular case of such a space is the widely known Distributed Hash Table (DHT), which is a 1-dimension space and objects are placed in it by computing a hash function over some attribute associated with the object (normally, its key).

Hyperspace hashing is a technique that uses such hyperspaces, by associating an attribute of an object with each dimension (of an hyperspace). Consider a set of attributes $\mathcal{A}$ such that $|\mathcal{A}| = N$. Consider also an N-dimensional space, such that each dimension i of the space is associated with attribute $\mathcal{A}_i \in \{\mathcal{A}_1, ..., \mathcal{A}_N\}$. Then hyperspace hashing locates an object in the space by hashing each attribute $\mathcal{A}_i$ of the desired object and assigning the result to the value of dimension i. This creates a set of N coordinates that correspond to a point in the space. Finally, the technique works by partitioning the space in non-overlapping regions that are distributed to servers. Therefore an object is known to belong to a given server simply when the corresponding coordinates in the space are within a region owned by the server. Conversely, to determine which nodes match a given query, the system can simply determine which regions are matched by the query, and contact their corresponding nodes to request the objects.

B. Subspace Partitioning

So far we have assumed a single hyperspace with as many dimensions as searchable attributes. However, creating one hyperspace with all attributes of an object suffers from the "curse of dimensionality" [26]: the volume of the space increases exponentially with each additional attribute. Hence, data will be scattered by the hyperspace in a sparse way. Should this large volume be partitioned into a small number of regions, such as the number of servers, this would very likely result in an unbalanced partitioning of data as some regions of this volume would be sparse and other more populated. Consequently, it becomes extremely difficult to partition the volume in a small number of regions, equal to the number of available servers, such that each server gets a similar load. Therefore, the strategy that is typically followed (and that is also used in HyperDex) consists in dividing the volume in many "small" logical regions, and then assign

these logical regions among the available servers. If the size of each region is small, and the number of regions is large, when these regions are shuffled among the physical servers, each server statistically gets a similar load.

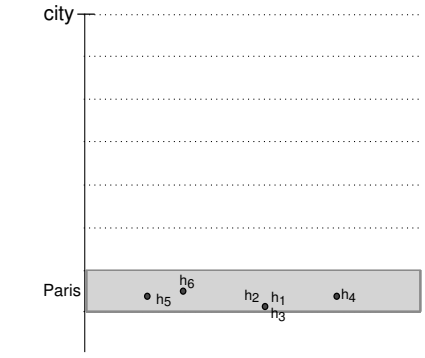
The solution for this problem is instead to resort to partitioning a hyperspace into several lower-dimensional hyperspaces called *subspaces* such that each subspace refers only to a subset of all the attributes of the object. The introduction of subspaces accounts for a considerable complexity increase in the configuration of HyperDex. This means that the programmer of an application must configure the set of subspaces to be used in HyperDex, which we denote by $\mathcal{S}$. We represent each subspace $\mathcal{S}_i \in \mathcal{S}$ by $\langle \mathcal{A}_1, ..., \mathcal{A}_k \rangle$, where $\mathcal{A}_i$ are attributes of the object of that subspace. Notice that while the number of regions in a single hyperspace would be proportional to $O(2^{|\mathcal{A}|})$, when using subspaces they become proportional to $O(|\mathcal{S}| \times 2^{|\mathcal{S}_i|})$.

Herein we shall use as examples a data set of hotels characterized by: a primary key; state; city; address; postal code; price; telephone (and others). We show two possible subspaces, their regions (possibly distributed to different servers), and some hotels corresponding to the result of a query. Consider a query where we search for hotels in Paris: using the subspace in Fig. 1a we only need to contact one region, whereas in Fig. 1b we need to contact three regions. If we instead search for hotels in Paris costing 120 euros we need only to contact one region regardless of which subspace we have.

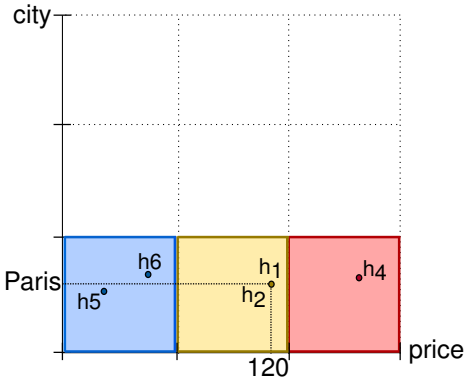
Note that it is important that the number of regions remains reasonable, regardless of how many dimensions we define a subspace to have. Therefore, as we shall see, HyperDex partitions each dimension of a subspace $\mathcal{S}_i$ in a way such that the resulting number of regions $\mathcal{R}_i$ is as close as possible to some default configured value $\mathcal{R}_{def}$. Ideally $\mathcal{R}_i$ should be close to the number of servers in the cluster to be used. Assuming that $\mathcal{R}_{def} = 8$, we can see that subspace $\langle city \rangle$ is partitioned such that it has 8 regions. However, subspace $\langle city, price \rangle$ is partitioned into 9 regions because there is no combination of integer partitions for the two dimensions that results in 8 regions.

In Fig. 2 we present an experiment that illustrates how much the performance of HyperDex can vary according to its configuration. The difference in throughput ranges from $8\times$ to $47\times$ when changing the dominance of search queries and modifications (Ready Heavy, Balanced and Write Heavy). The low throughput configuration was simply chosen as an hyperspace with all the attributes of the objects being mapped to dimensions. The best configuration, instead, is a complex configuration of subspaces with different dimensions and sizes.

In fact, as we shall see in the evaluation of our work, it is non trivial to understand the peculiarities of hyperspace hashing that affect performance. This strongly motivates the need understand the inner-workings of HyperDex to be able to predict its performance in an automatic fashion.



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



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

Figure 1: Two subspaces with different configurations to index hotels.

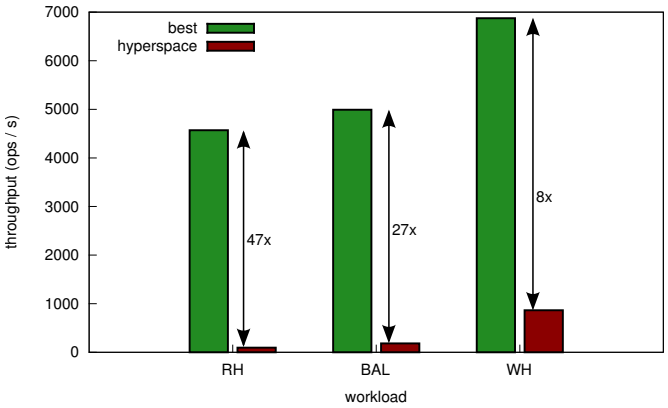


Figure 2: Performance of HyperDex when using a single hyperspace against a more complex configuration with subspaces. The results are shown for three variants of the same workload.

C. Description of Operations

In this section we describe in more detail the operations for accessing data in HyperDex. Throughout our presentation we shall use examples from a data and query sets of a hotel booking website, which we use in the evaluation of our solution in Section VI-B.

1) Search Operation: We consider queries defined in a similar fashion as subspaces, i.e., query $\mathcal{Q}_i$ is the set of attributes that the query accesses. Using as example Fig. 1b, a search query $\mathcal{Q}_i = \langle city, price \rangle$ results in contacting only one region — for instance the yellow region is contacted if querying for Paris and 120. For this, the client begins by choosing which subspace to use, from those configured. The configuration of the system, along with the mapping of regions to servers, is provided by a centralized fault-tolerant coordinator. Assuming the client logic chooses the subspace $\langle city, price \rangle$, it then hashes the values Paris and 120. This obtains the coordinates in both dimensions of the space, resulting in the decision to contact only the yellow region. Conversely, if $\mathcal{Q}_i = \langle city \rangle$, then the client can locally determine that it must contact the blue, yellow and red regions.

Contacting a region for a search implies sending a message to the server responsible for that region (by using the configuration provided by the coordinator). The contacted server filters the local data and returns only the results that are relevant to the search. The number of servers contacted (analogously, number of regions) varies as the queries are fully or only partially defined; for instance, when searching only by Paris in the subspace $\langle city, price \rangle$ all three coloured regions are contacted. Note that HyperDex maintains a full copy of each objects in every configured subspace.

Since the client has access to the configuration provided by the coordinator, it can make sure that it will contact the subspace $\mathcal{S}_i \in \mathcal{S}$ that is most suitable for a given query. This is achieved by iterating through $\mathcal{S}$ and selecting the subspace which yields the smallest number of regions.

2) Modification Operation: In this section we analyse the modification of existing objects in the Key-Value Store. For this we assume that the object is fetched (a simple get operation) through the primary key, and later inserted with attributes having their corresponding values modified. Consequently, a read operation only uses the (always existing) primary key subspace (that only has one dimension), and is unaffected by the subspace configurations. An insertion can thus be seen as a particular case of the modify: the insert does not require fetching first, and every attribute is modified, albeit no server previously owned it.

So far we have seen that a given search operation is conducted by resorting to a single subspace $\mathcal{S}_i$, i.e., the one among those in $\mathcal{S}$ that best suits the search query. When considering modifications instead, HyperDex must ensure that the modified object is updated in every subspace. This is because each subspace has a full copy of every object, and thus can be seen as an extra replication degree of data. Note that the primary key subspace counts as a subspace on its own. In addition to this, HyperDex can be configured to have a given fault tolerance level $\mathcal{K} = f + 1$ where f failures are tolerated. This guarantees that a subspace is always available (up to f faults) because the corresponding $\mathcal{K}$ replicas are guaranteed to be assigned to different servers. So, each object is replicated $(|\mathcal{S}| + 1) \times \mathcal{K}$ times.

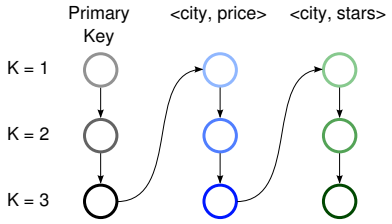
In fact, HyperDex organizes these replicas using chain replication [27] to ensure strong consistency of concurrent searches and modifications across the different subspaces. As in typical chain replication, servers are organized in a linear chain and forward requests to their successor. When the tail is reached, the inverse path is used to send acknowledgements that confirm the operation. Since the coordinates of the object in the subspaces depend on their content, HyperDex makes use of *value-dependent chaining*, the object's chain membership depends on its attributes' values. Consequently, when an attribute is modified, the object's position in all subspaces that contain that attribute must be changed to reflect this modification. Hence, new nodes may be required to enter the chain to reflect this change in position. Fig. 3 shows two examples of the resulting chain for different modifications. In both cases the configuration is the same, with the primary key subspace, two additional subspaces, and $\mathcal{K} = 3$.

Consider the modification $\mathcal{Q} = \langle tel \rangle$, meaning that it changes the telephone of a given hotel, shown in Fig. 3a. In this case the copies of that hotel have to be updated with the new telephone, which implies contacting the 3 replicas in each subspace. Fig. 3b a modification $\mathcal{Q} = \langle stars, tel \rangle$ that also changes the stars of the given hotel. This results in a more complex chain because the attribute $stars$ is in a dimension of one subspace. By changing the value of the $stars$, the given hotel changes its position in the subspace $\langle city, stars \rangle$, which is very likely to belong to a different region than the one it previously belonged to. This is why we label the two sub-chains of that subspace as old and new: to move the hotel to the correct region, the old servers delete it from their storage and the new servers insert it. Conversely, the servers in the sub-chain of the primary key and $\langle city, price \rangle$ merely update the values of the attributes of the hotel, and remain as owners of the hotel for those corresponding subspaces. Notice that in subsequent operations involving the modified objects, its chain will not include the nodes in the old part of the chain, they are only included in the chain operation which moves the object in the subspace.

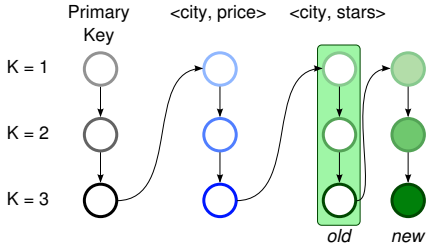
IV. Modelling HyperDex Performance

In the following we use the insights of the previous section on the inner workings of HyperDex to derive an analytical model that captures its performance. For this, we individually study the search and modification operations, and assess the validity of the corresponding model. In the following analyses we always assume scenarios with peak throughput, meaning the servers' processors are fully utilized and the network resources are not restraining the performance [1]. We also assume the objects in the data-set are uniformly distributed by the regions of subspaces, a factor that only depends on the hashing function.

[1]In deployments where the network is the bottleneck, such a solution cannot fully exploit the benefits of distribution, and thereof the optimal solution may converge to use as few servers as possible.



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



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

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

A. Modelling Searches

Under the assumptions of our analysis, i.e. that the application is saturated for obtaining peak throughput, then the performance of searching in HyperDex is a function of the number of regions contacted by the query. Herein we consider a generic search query $\mathcal{Q}_i$.

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

Proof: Since no subspace contains (at least) one attribute being searched, then the query must contact $\mathcal{R}_i$ 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. ■

Hypothesis 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 $\mathcal{Q}_i$.

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

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

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

Proof: 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 $\mathcal{R}_i$, each subspace dimension is split in $\sqrt[|\mathcal{S}_i|]{\mathcal{R}_i}$ partitions (as previously explained in Section III-B, this value is an approximation). As a result, the number of regions contacted is the product of this number of partitions $|E|$ times, because 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 throughput obtained for a given search. For this, we define the cost of a single query to be proportional to the product of the estimated 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:

$$\begin{aligned} T^{exp}(\mathcal{Q}_i) &= \frac{1}{cost(\mathcal{Q}_i)} \\ \text{where: } cost(\mathcal{Q}_i) &= \sqrt[|\mathcal{S}_i|]{\mathcal{R}_i}^{|E|} \times \frac{\mathcal{O}}{\mathcal{R}_i} \times \beta \end{aligned} \tag{2}$$

We note that Equation 1 is consistent with the results stated in Hypotheses 1 and 2. For this, consider the two following extreme cases:

- When $\mathcal{S}_i \cap \mathcal{Q}_i = \emptyset$, then $E = \mathcal{S}_i \setminus \mathcal{Q}_i = \mathcal{S}_i$. So, the number of regions contacted is $\sqrt[|\mathcal{S}_i|]{\mathcal{R}_i}^{|\mathcal{S}_i|} = \mathcal{R}_i$
- When $\mathcal{S}_i \subseteq \mathcal{Q}_i$, then $E = \mathcal{S}_i \setminus \mathcal{Q}_i = \emptyset$. So, the number of regions contacted is $\sqrt[|\mathcal{S}_i|]{\mathcal{R}_i}^{|\emptyset|} = 1$

Finally, we consider more complex 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)) of each search query as in the following equation:

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

B. Modelling Modifications

From the description of modifications in HyperDex the intuition is that the cost of those operations is proportional to the length of the chain. In this section we use that intuition to model the cost of a modification.

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

Proof: There is always a part of the chain proportional to the product of the number of subspaces ($|\mathcal{S}|$) and the replication degree ($\mathcal{K}$). Recall that we always have 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. There, we can see that there are some additional servers in the chain — the subspaces that are modified lead to two sub-chains instead of just one. We now capture more formally the number of these additional servers. We define $\mathcal{S} = \mathcal{N} \cup \mathcal{M}$, where $\mathcal{N}$ is the set of not modified subspaces, and $\mathcal{M}$ is the set of subspaces that have at least one dimension whose attribute is modified by modification $\mathcal{Q}_i$, i.e., $\mathcal{N} = \{\forall_{\mathcal{S}_i \in \mathcal{S}} : \mathcal{Q}_i \cap \mathcal{S}_i = \emptyset\}$ and $\mathcal{M} = \{\forall_{\mathcal{S}_i \in \mathcal{S}} : \mathcal{Q}_i \cap \mathcal{S}_i \neq \emptyset\}$. Then the proposed $length$ function follows from these observations. ■

Yet, this approach to model the modifications is considering that every server performs a similar effort. As pointed out earlier, we assume the bottleneck of the system to be the servers' local processing. Due to this, we need to carefully assess any difference between processing a modification operation in the servers.

Hypothesis 5: The cost of a modification has to be weighted by a corrective factor α.

Proof: Such a differences in processing effort may exist depending on whether a modification $\mathcal{Q}_i$ changes an attribute that is mapped to a dimension of some subspace. Using the previous example in Fig. 3b, a subspace that is not modified merely needs to update the local copy of the object, by 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.

We experimentally assessed considerable differences in the costs of those operations. The conclusion is that DELETE and WRITE have the same cost on average, which is approximately 50% more expensive than that of OVERWRITE. Consequently, we introduce a correction factor α to account for that difference. This factor is proportional to the number of subspaces that are modified, i.e., $|\mathcal{M}|$. This factor, similarly to β, is dependant on the hardware configuration and HyperDex implementation, and must be estimated from a running system. ■

Finally, as pointed out earlier, the modification always conveys a fetch operation to obtain the object (by its primary key). Therefore we model this by adding one server to the output of the $length$ function. We additionally consider a parameter T_{max} to capture the maximum throughput achievable by the hardware deployment in study. This parameter depends on the hardware configuration and can be obtained with a simple scenario such as when $length(\mathcal{Q}_i) = 1$, e.g. by modifying an object in an hyperspace configured only with the key subspace. We then obtain the following estimation for the throughput of a modification query $\mathcal{Q}_i$:

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

V. Assessing the Model Accuracy

Predicting the throughput of query sets which combine search and modify operations can be obtained by combining the costs of each query in weighted average, similar to that used for predicting the combined throughput of different types of search queries (Equation (3)).

To assess the viability of combining the queries in such way, we have tested the system using a data-set with information about hotels in the USA that is updated regularly. We used a synthetic benchmark, where we performed queries according to two workloads. Each workload is composed by two parts: the *Searches* and the *Modifications*. In order to test different ratios of searches and modifications, for each of the workloads we derive three configurations: a read-heavy configuration (RH), with 90% searches and 10% modifications; a balanced configuration (BAL), with 50% searches and 50% modifications; and finally a write-heavy configuration (WH) with 10% searches and 90% modifications. The following paragraphs describe workloads A and B:

Workload A: This workload simulates situations where users frequently perform very specific searches. So, the *Searches* of the workload are composed by 4 classes of searches, with increasing probability and with increasing number of attributes specified. The *Modifications* of the workload are composed by two modification queries with equal probability, which modify two attributes which are nor the most frequent nor the least frequent in the *Searches* part of the workload.

Workload B: This workload simulates situations where users most frequently perform very broad searches. So, the *Searches* of the workload 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 now the most common one. The *Modifications* of the workload simulate an environment where one of the attributes is frequently updated (e.g. the "price" attribute of a hotel), and a set of other attributes is less frequently updated in the same query (e.g. the address, telephone number and zip code for an hotel).

A. Parameter estimation and Hardware Environment

Recall that we used three parameters in our models that are dependent on the hardware configuration. They have to be estimated for a given system deployment as they are affected by the network and hardware conditions. Then these parameters can be used as inputs to our model, and remain the same independently of the workload.

We propose to use simple scenarios to experimentally assess these hardware-dependent parameters. For space constraints, we give an example of estimating the α parameter required for the modifications; the following conclusions were similarly obtained for the other parameters. This parameter can be assessed with a micro-application configured

with different (yet simple) subspaces and a workload that repeatedly invokes modifications. 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 through Equation (4). We used 24 such simple executions to derive α in our environment. The value that minimized the error in those tests was 2.3, which we shall use in the following experiments. In fact, our estimation is not the best one, as using an extensive array of hundreds of complex workloads reveals that we could lower our final error in 6.5% if α was precisely estimated. As we shall see in the following section, this gross (and easy) estimations have a reduced impact in the final goal of our work to automatically devise the best configuration of HyperDex.

Our hardware deployment consists of 9 servers in a private cluster connected through 1GB ethernet links. HyperDex coordinator was used in a dedicated machine, whereas the other 8 servers executed the daemon that serves requests. We followed the same testing environment of the authors of HyperDex by deploying 1 client process in each of the 8 servers, with each client executing 32 threads performing requests without think time.

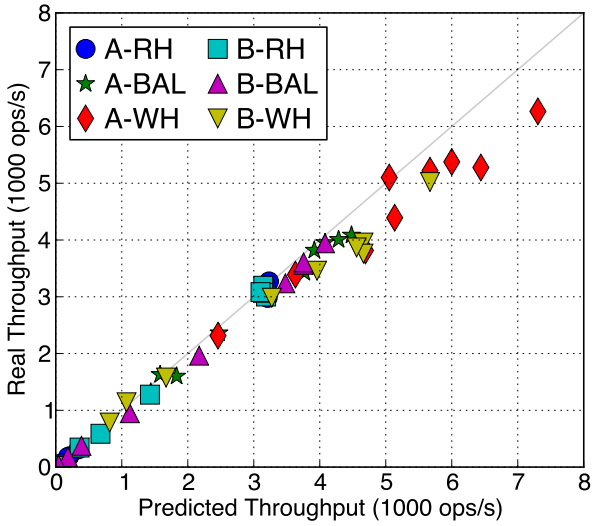


Figure 4: Estimation vs measured throughput in different configurations for each workload.

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

Figure 5: τ coefficient, average and maximum $\overline{\tau}$ distance between estimated and real ranking of configurations.

B. Throughput Estimation

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 present 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 grey line. In fact, these results show that our model is able to predict fairly accurately the real throughput obtained by the system, given that the average error is 9% with a standard deviation of 7%.

VI. Auto-Configuring HyperDex

In the following we use the previous models to estimate the best possible configuration for a given workload. This way we can automatically configure HyperDex without burdening the programmer with this concern. For this, we present the architecture of our solution in Section VI-A. We then assess its accuracy in Section VI-B by comparing our predictions with extensive measurements.

A. Architecture

Our system is composed of three main modules: the *query analyzer*, *optimizer*, and *configuration deployment*. The query analyzer is responsible for monitoring the system to generate a profile of the queries performed. This profile is then fed to the optimizer, which will use the predictive model to determine which configuration best suits the profile. Finally, the configuration deployment module will redeploy the system using the selected configuration.

The query analyzer monitors the servers, in order to generate a log of the queries performed in each of them. These partial logs are then compressed by types of queries, and aggregated into a single log. Then, the query analyzer calculates the probabilities p_i of each type of query $\mathcal{Q}_i$ and produces the set of searchable attributes. Afterwards it pipes this profile to the *optimizer*.

The main goal of *optimizer* is to derive most efficient configuration of HyperDex. To achieve this it first reads the system constraints: replication degree $\mathcal{K}$, the number of expected objects $\mathcal{O}$ in the system, the number of regions $\mathcal{R}$ each subspace is to be partitioned, and the correction factors α and β for the current deployment. The optimizer works in three phases. The first phase of the algorithm is to generate all possible combinations using the set of attributes which was provided by the query analyzer. Then it proceeds to the second phase where the prediction model is queried to obtain a throughput estimation for every possible configuration. In the final phase, all configurations are ranked in decreasing order, and the optimizer returns a random configuration from those which were ranked in first place. In order to update HyperDex's configuration to match the current workload, the

best configuration is piped to the *configuration deployment* module which re-deploys HyperDex accordingly.

B. Evaluation

In this section, we evaluate the proposed system. We begin by evaluating the accuracy of the configuration rankings predicted by our system. In fact, even though the accuracy of the throughput estimation may not be perfect, it can still be the case that the ranking of configurations is correctly assessed. We then compare the best configuration chosen by our system against several baselines.

1) Ranking Configurations: More importantly than predicting the real throughputs, our tool is useful as long as it accurately predicts the configuration which leads to the best performance. So, it should correctly rank different configurations according to their throughput. Figure 5 presents Kendall's τ coefficient [28] for the configuration rankings predicted by our model against the rankings experimentally obtained for each workload. The τ coefficient is an indication of how two rankings differ; it varies in the interval $[0, 1]$ where 1 indicates complete concordance and 0 indicates complete discordance. Hence, the ranking accuracy of our system is better when τ is close to 1. The results presented indicate that, for all workloads, there is a high correlation between the throughput rankings predicted by our system and the real rankings.

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 caused by the values predicted being close to each other. To illustrate this matter, we have applied Kendall's τ distance to our predicted and real rankings. This distance measures the number of pairs of rankings which have a different relative position in the two distributions. We then multiplied this distance by the relative difference in throughput between the out of order pairs, to convey how relevant these ranking errors are. We represent this adjusted distance by $\overline{\tau}$. In this case it is better to have a 0 distance (i.e., the ranking was correctly predicted) or as close to it as possible.

The average and maximum $\overline{\tau}$ of the configurations in each workload are also shown in Figure 5. We note that most ranks have distance zero, only 4 ranks out of the 60 ranked configurations have a value greater than 2% for this metric, and none of them are larger than 14%. This indicates that even though our system may perform some errors while ranking configurations, these errors do not significantly affect the final throughput. Furthermore, we observe that for 5 out of 6 workloads, our system was indeed able to select the best configuration, while for the remaining one (workload B-RH) it selected the second best configuration, which generated a throughput 6% lower than that of the best configuration sampled.

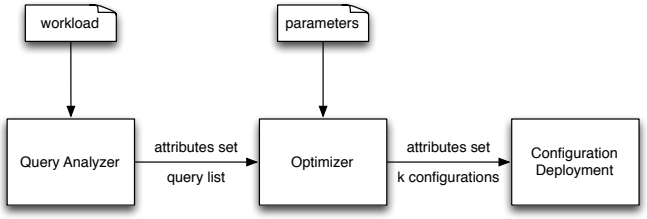


Figure 6: Architecture of the system

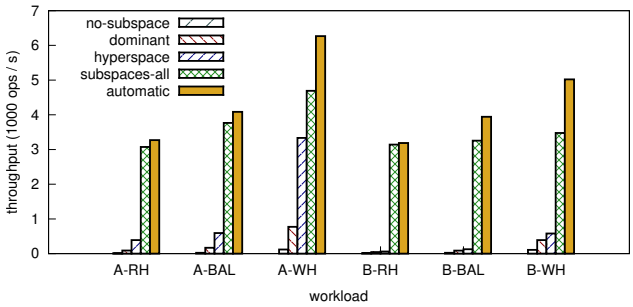


Figure 7: Throughput of configurations selected by our solution and by the baselines

2) Comparison with baselines:: In this section, we choose the best configuration predicted by our tool and compare it with baselines relying on heuristics to select the best configuration: *No-subspace* as equivalent to a regular key-value store; *Hyperspace* using only a single subspace containing all attributes in the queries; *Subspaces-all* where a subspace is configured with one dimension and for each attribute; and *Dominant* where a single subspace us used, containing the most common attribute in the queries.

In Figure 7, we present the throughput results obtained using the configuration selected by our system (denoted by "automatic") and that obtained by the baselines. The results show how our models consistently capture the best configuration; even when comparing with the baseline which achieves the best results, our system achieves throughputs up to 31% larger. Note that the **Subspaces-all** strategy can achieve results close to that of our solution. Yet, its prediction is hampered by the increase in modification likelihood in the system, which is a straightforward consequence of the heuristic being focused only on favouring the search operations: when using a subspace per attribute, all search queries will match a single region on any subspace, leading to a good performance. Optimizing for every possible case is non obvious, for which reason the automatization provided by our tool becomes important and useful. Finally, we also highlight that unlike the presented baselines, our strategy can adapt to workload changes in order to maximize the throughput while the remaining strategies are mostly static approaches.

VII. Conclusions

In this work, we presented a solution for auto-configuring a multi-dimensional NOSQL data store, using a predictive model to predict throughputs and then decide on which configuration generates the highest system throughput.

We have shown that our approach can predict the system throughput with up to 92% of accuracy, can select the best configuration on most cases (and when it does not, the error has a small impact on the system), and can outperform static heuristics for configuring the system.

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 and dynamically change HyperDex according to changes in the workload and the prediction of our tool.

Acknowledgments

Parts of this work have been performed in collaboration with other members of the Distributed Systems Group at INESC-ID, namely, Luís Rodrigues, Paulo Romano, João Paiva and Nuno Diegues.

References

[1] F. Chang et. al., "Bigtable: a distributed storage system for structured data," in *OSDI '06*.

[2] G. DeCandia et. al., "Dynamo: amazon's highly available key-value store," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, 2007.

[3] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, 2010.

[4] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed b-tree," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 598–609, 2008.

[5] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[6] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram, "Querying peer-to-peer networks using p-trees," in *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004*. ACM, 2004, pp. 25–30.

[7] R. Vilaca, R. Oliveira, and J. Pereira, "A correlation-aware data placement strategy for key-value stores," in *Distributed Applications and Interoperable Systems*. Springer, 2011, pp. 214–227.

[8] R. Escriva, B. Wong, and E. Sirer, "Hyperdex: a distributed, searchable key-value store," in *SIGCOMM '12*.

[9] P. Ganesan, B. Yang, and H. Garcia-Molina, "One torus to rule them all: multi-dimensional queries in p2p systems," in *WebDB '04*.

[10] Y. Shu, B. C. Ooi, K.-L. Tan, and A. Zhou, "Supporting multi-dimensional range queries in peer-to-peer systems," in *Peer-to-Peer Computing, 2005. P2P 2005. Fifth IEEE International Conference on*. IEEE, 2005, pp. 173–180.

[11] D. Karger et. al., "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in *STOC '97*.

[12] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[13] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware 2001*. Springer, 2001, pp. 329–350.

[14] S. P. et. al., "When scalability meets consistency: Genuine multiversion update-serializable partial data replication," in *ICDCS '12*.

[15] S. Peluso, P. Romano, and F. Quaglia, "Score: a scalable one-copy serializable partial replication protocol," in *Proceedings of the 13th International Middleware Conference*. Springer-Verlag New York, Inc., 2012, pp. 456–475.

[16] D. Tsoumakos and N. Roussopoulos, "Analysis and comparison of p2p search methods," in *Proceedings of the 1st international conference on Scalable information systems*. ACM, 2006, p. 25.

[17] Y. Chawathe et. al., "Making gnutella-like p2p systems scalable," in *SIGCOMM '03*.

[18] P. Reynolds and A. Vahdat, "Efficient peer-to-peer keyword searching," in *Middleware '03*.

[19] A. Bharambe, M. Agrawal, and S. Seshan, "Mercury: supporting scalable multi-attribute range queries," in *SIGCOMM '04*.

[20] A. T. Clements, D. R. Ports, and D. R. Karger, "Arpeggio: Metadata searching and content sharing with chord," in *Peer-to-Peer Systems IV*. Springer, 2005, pp. 58–68.

[21] C. Schmidt and M. Parashar, "Enabling flexible queries with guarantees in p2p systems," *Internet Computing, IEEE*, vol. 8, no. 3, pp. 19–26, 2004.

[22] A. Andrzejak and Z. Xu, "Scalable, efficient range queries for grid information services," in *Peer-to-Peer Computing, 2002.(P2P 2002). Proceedings. Second International Conference on*. IEEE, 2002, pp. 33–40.

[23] B. Trushkowsky et. al., "The scads director: scaling a distributed storage system under stringent performance requirements," in *FAST'11*.

[24] F. C. et. al., "Met: workload aware elasticity for nosql," in *EuroSys'11*.

[25] D. Didona et. al., "Transactional auto scaler: elastic scaling of in-memory transactional data grids," in *ICAC '12*.

[26] R. Bellman, *Dynamic Programming*, ser. Dover Books on Computer Science Series. Dover Publications, Incorporated, 2003. [Online]. Available: http://books.google.pt/books?id=fyVtp3EMxasC

[27] R. van Renesse and F. Schneider, "Chain replication for supporting high throughput and availability," in *OSDI'04*.

[28] M. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1/2, 1938.