Data locality aware partitioning schemes for large-scale data stores

Muhammet Orazov

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Examination Committee

Chairperson: Prof. Doutor Pedro Manuel Moreira Vaz Antunes de Sousa
Supervisor: Prof. Doutor Luís Eduardo Teixeira Rodrigues
Members of the Committee: Prof. Doutor João Manuel Pinheiro Cachopo
Prof. Doutor Johan Montelius

July, 2013

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Luís Rodrigues, for giving me this unique opportunity to work under his supervision, and for providing constant motivation, inspiration and encouragement.

I would like to thank Paulo Romano for his insightful discussions and comments which enlightened me to look for different angles in order successfully solve the main problems of this work.

I am grateful to João Paiva and Nuno Diegues for their both theoretical and practical support during the execution of this work and preparation of this thesis.

Manuel Bravo has been a great colleague for me during this work. I have especially enjoyed the discussions and brainstorming I had with him which made our tea breaks most pleasant.

Last, but no means the least, I would like to thank my parents Gülnabat and Kakajan, my sisters Göwher and Altyn, and my brother Süleÿman for their immense love, support and continuous encouragement.

Lisbon, July, 2013

Muhammet Orazov

European Master in Distributed Computing, EMDC

This thesis is part of the curricula of the European Master in Distributed Computing (EMDC), a joint program among Royal Institute of Technology, Sweden (KTH), Universitat Politecnica de Catalunya, Spain (UPC), and Instituto Superior Técnico, Portugal (IST) supported by the European Community via the Erasmus Mundus program.

My track in this program has been as follows:
First and second semester of studies: IST
Third semester of studies: KTH
Fourth semester of studies: IST

For my parents, sisters
and brother.

Resumo

Os sistemas de armazenamento chave-valor caracterizam-se por exibirem elevado desempenho e escalabilidade. No entanto, a sua interface é bastante limitada, só permitindo aceder aos objectos através da sua chave primária. Trabalhos recentes corrigem esta limitação, permitindo o acesso a dados por atributos secundários, em particular através da projecção dos objectos em espaços multi-dimensionais. Estas soluções pecam, no entanto, por exigirem um complexo trabalho de configuração. Este trabalho contribuiu para o desenvolvimento de técnicas de configuração automática para este tipo de sistemas, estudando este problema para o caso concreto do HyperDex. Através de uma análise pormenorizada do funcionamento do HyperDex, derivamos um modelo analítico que captura o seu desempenho. Com base neste modelo, desenvolvemos uma metodologia que permite fazer a configuração automática do HyperDex. Finalmente, avaliamos extensivamente tanto a metodologia como o modelo analítico, recorrendo a padrões de carga que simulam situações reais de utilização de sistemas de armazenamento chave-valor.

Abstract

Key-value stores are highly scalable storage systems that can offer extremely good performance. For these reasons key-value stores are the backbone of several large-scale data processing systems. However, their interface is rather restrictive since it only allows to access objects through their keys. To overcome this limitation, recently proposed systems have developed mechanisms for storing data in multiple dimensions mappings in order to allow searching for objects in 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 thesis makes two main contributions towards the automatic configuration of such multi-dimensional key-value stores: First, from a detailed description of the inner workings of these systems, we derive a model which describes the behavior of queries in multi-dimensional spaces. We then use this model to predict real throughputs of the system for complex workloads. Using these results, we propose 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.

Palavras Chave
Keywords

Palavras Chave

Sistemas de armazenamento chave-valor

NoSQL

Posicionamento de dados

Localidade no acesso aos dados

Espaços multi-dimensionais

Particionamento de dados

Keywords

Key-value Stores

NoSQL

Data Placement

Data Locality

Multi-Dimensional Space

Partitioning

Contents

List of Figures

List of Tables

1 Introduction

Key-value stores have emerged as highly-scalable alternatives to classic distributed databases. By exposing simpler interfaces, allowing objects to be accessed only by a given key, it has been possible to implement a new generation of distributed stores that are significantly faster than SQL-based systems. Notable examples of key-values stores are BigTable (Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, & Gruber 2008), Dynamo (DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, & Vogels 2007), and Cassandra (Lakshman & Malik 2010).

Yet, accessing an object solely by a single key is rather restrictive. To understand the limitation, consider for instance a website for booking hotel rooms. 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 been mostly based on creating indexes on top of the underlying key-value store (Aguilera, Golab, & Shah 2008; Bentley 1975; Crainiceanu, Linga, Gehrke, & Shanmugasundaram 2004). 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 (Vilaca, Oliveira, & Pereira 2011; Escriva, Wong, & Sirer 2012; Ganesan, Yang, & Garcia-Molina 2004; Shu, Ooi, Tan, & Zhou 2005). Among these, HyperDex has gathered a unique set of characteristics that makes it a very appealing solution to the problem.

1.1 Motivation

The main idea of HyperDex is to implement *hyperspace hashing*, which behaves similarly to consistent hashing techniques (Karger, Lehman, Leighton, Panigrahy, Levine, & Lewin 1997; Stoica, Morris, Karger, Kaashoek, & Balakrishnan 2001; Rowstron & Druschel 2001). 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 of all 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 spaces that best suits his application. As we shall see in this thesis, we argue that deciding on the optimal solution for such configuration is far from trivial since the number of different configurations increases exponentially with addition of an extra object attributes. Thus, finding the most efficient configuration for a given application becomes *NP-Hard* problem.

In this thesis we study hyperspace hashing, and in particular HyperDex, both from an analytical and experimental perspectives. The objective is to predict the performance of HyperDex in a given application. As a result, we can then implement the prototype system in order to assist the developers in optimizing the configuration of hyperspaces to improve the performance of a application.

1.2 Contributions

This work addresses the problem of data placement in Internet scale key-value stores. More precisely, the thesis analyzes the *hyperspace hashing* used in HyperDex, models the performance of HyperDex and evaluates the accuracy of the proposed modeling. As a result, the thesis makes the following contributions:

- Provides a detailed analysis of the HyperDex operations;
- Build a model that allows to predict the HyperDex performance;
- Proposes a technique to estimate the best *hyperspace* configuration for a given application.

1.3 Results

The results produced by this thesis can be enumerated as follows:

- A experimental assessment of the performance model of HyperDex;
- A prototype that derives the most efficient configurations for HyperDex deployment;
- An experimental evaluation of the implemented prototype using the real world application workloads.

1.4 Research History

In the beginning, the primary focus of this thesis was to study several practical systems that solve locality aware data placement problems in distributed key-value stores and the techniques used to model the solutions of them. As a result of that study, we expected to be able to produce a novel locality data placement model that would improve an existing one. However, during our research a new system, HyperDex, was introduced(Escriva, Wong, & Sirer 2012). We found this system very interesting because it provides new data placement technique called *hyperspace hashing* which can be extended to support relational API. We investigated the details of the hyperspace hashing and realized that the configuration of hyperspace is non-trivial task as we show in the following chapters of this work. This motivated us to focus on HyperDex analysis and configuration.

During my work, I benefited from the fruitful collaboration with several members of the GSD team, namely Luís Rodrigues, Paulo Romano, João Paiva and Nuno Diegues.

1.5 Structure of the Document

The rest of this document is organized as follows. Chapter 2 provides an introduction to the different technical areas related to this work. Chapter 3 introduces HyperDex system and Chapter 4 analyses the performance of HyperDex. Then the chapter 5 presents a prototype implementation that derives the top-k efficient configurations for a given workload and provides the results of the experimental validation of the system. Finally, Chapter 6 concludes this document by summarizing its main points and future work.

2 Related Work

This thesis addresses the problem of data placement in large-scale key value stores. This chapter surveys the state of the art in data placement techniques for this type of storage systems. First we cover the main data placement techniques. Then we describe some relevant practical systems implementing different data partitioning and data placement techniques.

2.1 Data Placement in Key-value Stores

Key-value stores are a class of storage systems that, as the name implies, allow to store and retrieve data using a unique identifier that is associated with each data object (the data object *key*)(DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, & Vogels 2007; Lakshman & Malik 2010). Typically, the interface of key-value stores consists of two main operations, namely the *put()* and *get()* operations: clients issue *put(key)* operations in order to store object in the system; or *get(key)* commands to retrieve objects associated with keys. However, it is interesting to provide support for additional functionality, such as performing queries on the key-value store by data attributes. In the thesis, we are particularly interested in systems that can support complex queries.

We are interested in large-scale systems, able to store very large data sets that cannot be physically stored in a single machine. More precisely, we assume that the key-value store is implemented by a (potentially very large) set of cooperative data servers. Each server is only required to locally store a subset of all data objects. A fundamental problem in distributed key-value stores is then how to distribute the data objects among the existing servers. This is called the data placement problem. Data placement must consider the following aspects:

- First, data placement must be made such that the physical storage constraints of each individual servers are not violated.

- Data placement should take into consideration locality of data access by the application, such that application requests can be served by contacting as few servers as possibly, to avoid overloading the network.

- Data placement should also promote good load-balancing among the available servers, in particular, concurrent request by different clients should be able to be processed in parallel by different servers whenever possible.

- Finally, the mapping of data item into servers should by stored in a compact and easy to consult manner, such that the server responsible for maintaining a given key can be retrieved efficiently.

2.2 Data Placement Techniques

In this section we mainly consider techniques that can be used to support static data placement. We consider that data placement is static if the mapping among data objects and servers does not have chance of change very infrequently (for instance, in response to changes in the membership of the server pool). In opposition, dynamic data placement techniques may migrate data from one server to another frequently to react to dynamic changes in the workload. Dynamic data placement is discussed at the end of the section.

As noted above, an important aspect in data placement is to ensure that the mapping between keys and servers can be stored and retrieved efficiently. Therefore, many approaches for static placement use deterministic functions that allow any node to derive locally this mapping, just based on the object key and information about the available servers.

2.2.1 Consistent Hashing

Consistent hashing(Karger, Lehman, Leighton, Panigrahy, Levine, & Lewin 1997; Lewin 1998) is a static data placement technique that uses a hash function to map an object key into a given server. For this purpose, servers are assigned an unique identifier in a server identifier space (these identifiers should be picked uniformly at random). Furthermore, servers are organised in a logical ring, according to the order of their identifiers. Then, a deterministic hashing function, such as SHA-1(Eastlake & Jones 2001), maps key values into identifiers in the server identifier space. Finally, each server becomes responsible for the identifier range between its position and that of its predecessor node.

Consistent hashing is remarkable efficient, even if the membership of the system is dynamic: the failure or join of a node only affects its neighbouring nodes in the circular identifier space. This significantly reduces the bandwidth usage of the system and simplifies reconfigurations, since any change in data placement is local and does not involve global coordination.

However, since consistent hashing depends on a hash function for data partitioning, it has two main drawbacks. First, nodes are assigned to a identifier space randomly, and placement is oblivious of the

network locality properties. Second, placement does not take into consideration the workload, which may results in sub-optimal load distributions if different objects are accessed with different frequencies.

2.2.2 Multi-dimensional Hashing

Unlike consistent hashing, where a one-dimensional identifier space is used, CAN(Ratnasamy, Francis, Handley, Karp, & Shenker 2001) uses a d-dimensional Cartesian coordinate space. This virtual coordinate space is dynamically partitioned into zones and each node is responsible for a specific zone. In order to store a key-value pair, the key is deterministically mapped onto a point P in the coordinate space using a hash function. Then the node which is responsible for a zone where P lies stores the key-value pair. Data retrieval works in a similar way. The request messages are greedily forwarded to the nearest neighbouring zone towards the destination. Each CAN node maintains $O(d)$ state; therefore, it does not increase with number of nodes in the system. However, the cost of message routing in CAN is $O(dN^{1/d})$. Multi-dimensional hashing provides more control on the relative data placement on objects, given that different object attributes can be used to define a multi-dimensional key.

2.2.3 Hyperspace Hashing

Hyperspace Hashing(Escriva, Wong, & Sirer 2012) is a data placement strategy that aims at supporting complex queries, that can be based on data attributes other than just the key itself. To accomplish this functionality, Hyperspace Hashing creates an Euclidean space, hyperspace, where attributes of an object define the axes of this space. Subsequently, it maps each data object to a deterministic coordinate in this multi-dimensional space. As in other hashing techniques, in Hyperspace Hashing the location of data item is determined by taking the hash of object attributes which results in a position along the corresponding axis.

The multi-dimensional hyperspace is mapped to the storage servers as follows. The hyperspace is divided into a pre-defined fixed number of non-overlapping regions. Then these regions are deterministically mapped to servers, which will store the objects whose coordinates reside within the region. Therefore, in order to insert or delete an object, clients compute the coordinate of the object by hashing its attributes, determine the region where it falls, and finally contact the storage server responsible for that region.

2.2.4 Space Filling Curves

Similarly to Hyperspace Hashing, in Space Filling Curves (SFC)(Sagan 1994), the attributes or keywords of an objects create a d-dimensional space. This adds flexibility to the queries, allowing searching for

objects using their keywords, in a similar way to Hyperspace Hashing. However, unlike Hyperspace Hashing, SFC-based systems convert the multi-dimensional space to an unidimensional space ($f : N^d \rightarrow N$). The d-dimensional space is viewed as a d-dimensional cube partitioned into sub-cubes, which is mapped onto a line in a deterministic way. Using this mapping, a point in the cube can be described by its spatial coordinates, or by a length along the line which passes once through each point (sub-cube) in the volume of the cube. Similarly to CAN, in SFC-based systems each server is mapped to a point in the d-dimensional space. Then, the server is subsequently mapped to a point in the space-filling line. Keys are attributed to servers using an approach that is similar to the one used in consistent hashing.

The main advantages of Space Filling Curves are that they preserve locality and allow to efficiently determine subsets of nodes which are responsible for nodes which match a query. If two objects have similar attributes or their keywords are near in lexicographical order, they are called local data items. Objects that are local in multi-dimensional space will be mapped closely in the 1-dimensional space and therefore, be stored in the same or topologically near nodes.

2.2.5 Bloom Filters

In some cases, the data placement that results from the use of consistent hashing may not provide the desired level of locality or load-balancing. As a result, it can be interesting to use alternative techniques that allow to encode arbitrary mappings among data items and data servers. As noted before, the challenge is to encode such mapping in an efficient way.

Bloom filters(Bloom 1970) are efficient probabilistic data structures which can be used to check if an object is mapped to a given node. These structures consist of a bit array of size m. A value is inserted in the filter by setting k different bits in this array as a result of applying k different independent hash functions to the key. The bloom filter can be queried by checking if all the k positions are set to 1. Note that it is possible that all bits associated with a given key are set to 1 not because the key has been inserted, but as a result of the insertion of multiple other distinct keys. Therefore, although bloom filters are a space efficient data structure to check existence of a data item, they may result in false positives when querying for an object.

2.2.6 Probabilistic Associative Array

The Probabilistic Associative Array (PAA)(J. Paiva & Rodrigues 2013) is a space efficient mapping data structure which is able to store arbitrary associations among keys and nodes in an distributed system. The PAA uses multiple Bloom filters internally for tracking the inserted data items. Therefore, it may provide false positives while looking up the keys. The PAA has been designed for a system that use a combination of consistent hashing and arbitrary placement. Objects that are not included in the

PAA are placed using consistent hashing. To avoid checking all the bloom filters for all objects, PAA uses Decision Tree classifiers to assert in an object is mapped by the PAA. To be effective, PAA requires users to associate additional semantic information. e.g a data type, with the keys to be stored in the PAA. Since PAA are space efficient they can be easily disseminated among the nodes with little communication overhead.

2.2.7 Optimal Placement

Consistent hashing and its variants rely on large numbers and on the statistic properties of hash functions to distribute items evenly among servers. However, in many cases, access to data items is not uniformly distributed; some items can be hot-spots while other items can be seldom accessed. Also, some items may be correlated and always accessed together. As a results, the placement that results from the use of consistent hashing may be suboptimal.

Optimal placement can can be derived by assigning a "cost" to each placement decision. This cost is application specific and depends on the characteristics of the data set and of the workload. Assuming that costs can be assigned to individual placement decision, the problem of optimal placement becomes an optimisation problem that can be formulated as an Integer Linear Programming (ILP)(Papadimitriou & Steiglitz 1998) problem. In order to formulate ILP, cost functions such as access latency, load balance, or access frequency, should be defined. Also, constraints such as replication degree, storage capacity or load capacity must be specified.

For example, to derive a data placement that yields the minimum access cost, one can design the ILP model as follows: the placement of the data items to nodes is modeled using binary matrix X, in which $X_{ij} = 1$ if the object i is assigned to a node j, and $X_{ij} = 0$ otherwise. We then further formulate an ILP problem which strives to minimize the total cost of accessing all data items across all nodes subject to two constraints: (a) the number of replicas of each data object should be equal to given replication degree, and (b) each node should be assigned more items than its capacity, they have finite storage capacity. Then the cost function can be formulated as following:

$$\sum_{j \in N} \sum_{i \in O} \overline{X}_{ij}(cr^r r_{ij} + cr^w w_{ij}) + X_{ij}(cl^r r_{ij} + cl^w w_{ij}) \tag{2.1}$$

with the following constraints:

$$\forall i \in O : \sum_{j \in N} X_{ij} = d \wedge \forall j \in N : \sum_{i \in O} X_{ij} \leq C_j \tag{2.2}$$

The parameters used in formulation are listed in Table 2.1

N	the set of nodes j in the system
O	the set of objects i in the system
X	a binary matrix in which $X_{ij} = 1$ if the object i is mapped to node j, and $X_{ij} = 0$ otherwise
r_{ij}, w_{ij}	the number of read, write accesses performed on an object i by node j
cr^r, cr^w	the cost of remote read, write access
cl^r, cl^w	the cost of local read, write access
d	the replication degree, that is the number of replicas of each object in the system
C	the capacity of node j

Table 2.1: Parameters used in ILP formulation

Unfortunately, ILP is a NP-Hard(Garey & Johnson 1979; Hochbaum 1996) problem. Finding an optimal solution using ILP is not feasible in large-scale distributed systems because the number of decision variables are proportional to number of objects. Therefore, most systems design approximation algorithms to solve ILP by reducing the number decision variables. In the following sections we present two approximation heuristics which derives solution to the ILP problem.

2.2.8 Distributed Replication Group

In this model, a group of servers is created dedicating some storage for the object replicas. A server is responsible to handle the requests coming from its clients and also from other servers present in the group. A user request is first received by the local node. If the requested object is stored locally, it is returned to the requesting user immediately, thereby incurring a minimal access cost. Otherwise, the requested object is searched and fetched from other nodes in the group, at a potentially higher access cost. If the object cannot be located anywhere in the group, it is retrieved from an origin server, which is assumed to be laying outside the group with the highest access cost.

We can formulate an ILP problem for the *distributed replication group*, given the access cost if the object is stored locally (cl), the cost when the object fetched from a node within the replication group (cg), and access cost when the object accessed remotely from the source (cr). Typically, $cl \leq cg \leq cr$. Using these costs, it is possible to derive an objective function aiming to minimize the access time at each server over all objects as follows:

$$\sum_{j \in N} \left(\sum_{i:X_{ij}=1} (cl^r r_{ij} + cl^w w_{ij}) + \sum_{i:X_{ij'}=1} X_{ij'} (cg^r r_{ij} + cg^w w_{ij}) \sum_{i:X_{ij}=0 \wedge X_{ij'}=0} \overline{X}_{ij} (cr^r r_{ij} + cr^w w_{ij}) \right) \tag{2.3}$$

with the following constraints:

$$\forall i \in O : \sum_{j \in N} X_{ij} = d \wedge \forall j \in N : \sum_{i \in O} X_{ij} \leq C_j \tag{2.4}$$

where j' is a node inside the replication group. The rest of the parameters are defined as in Table 2.1.

The first term of the objective function represents accessing object stored locally in node j. The second term represents the access cost if the object is not stored locally at node j, but stored in one of the nodes j' in the replication group. The third term represents the cost related to accessing object from the source. The distributed replication group model is efficient if there is high proximity among the server nodes.

2.2.9 Greedy Heuristics

Another approximation towards the solution of the ILP problem is to formulate it as 0/1 Knapsack problem(Kellerer, Pferschy, & Pisinger 2004). In this approach, object access rates are expressed as cost function and object weights assumed to be unit size. The capacity of Knapsack is equal to an integer value which is the capacity of a node. Given the objects replicated on the other nodes, the optimal data placement for a node can be found in the following way: (i) first objects are ordered in decreasing order of their cost function, (ii) then a node selects most valuable objects until its capacity is full. Since object sizes are unit and capacity of a node is integral, this greedy approximation finds an optimal solution to the 0/1 Knapsack problem; as a result to the ILP problem.

2.2.10 Graph Partitioning

One of the goals of data placement can be to ensure locality in data accesses. In particular, if the application code is structured using transactions, atomic sequences of data operations that are executed in sequence, it can be beneficial to ensure that all objects accessed by a given transaction reside in the same data partition. By spreading the partitions into different nodes, it is possible to improve the throughput of the queries since different nodes could process different transactions in parallel, without mutually interfering with each other.

In order to find the balanced partitions for the key-value store, its workload can be transformed into graph representation. The vertices of the graph will represent the data items and the edges will be connected between two vertices that maps to the items accessed by a given transaction. Consequently this graph representation will describe both the servers and queries over it. After its representation is finished, the graph is split into k non-overlapping partitions so that the overall cost of cut edges is minimized considering the partition weights are equally balanced. Partition weights are the sum of

the node weights which can be the number of times a data item is accessed. However, $k - way$ graph partitioning is an NP-complete problem. There has been sheer research and development yielding efficient heuristics that solves the graph partitioning problem using distributed parallel implementations(Karypis & Kumar 1998; Karypis & Kumar 1995; Kernighan & Lin 1970; Khandekar, Rao, & Vazirani 2009).

2.2.11 Dynamic Data Placement

Dynamic data placement consists of changing the placement of some or all data items in response to changes in the workload. For instance, a given item may suddenly become an hot-spot and more replicas of that item may be created to have more processing power to serve incoming requests. Dynamic data placement can be performed periodically or can be triggered by some specific events, for instance, when one server becomes overloaded. Dynamic data placement is more effective in systems that have a richer set of configuration opportunities and, in particular, in systems that support arbitrary placement.

Unfortunately, dynamic data placement comes at multiple costs. First, a new placement must be derived. Second, data items must be physically copied from their previous locations to the new locations, and this can consume significant time and network resources. Finally, the new mapping must be propagated to all nodes in the system, such that the new locations of data items can be easily obtained.

2.3 Example Systems

In this section we illustrate the use of some of the techniques surveyed above, by describing some relevant concrete systems that address the problem of data placement in key-value stores.

2.3.1 URSA

Ursa(You, Hwang, & Jain 2011) is a system that illustrates how integer linear programming can be used to derive data placement in key-value stores. To avoid the scalability limitations of ILP, Ursa only attempts to optimize the placement of data items that have been previously identified as hot-spots. The main goal of the system is to reduce the latency in remote data access, by placing data items close to the clients. However, it balances the benefits of moving objects closer to the clients with the costs involved in data migration. Therefore, a data item is not relocated if the costs of migrating the item are larger than the benefits that can be achieved by the relocation. To avoid doing an exhaustive search over the cost/benefits of all possible relocation options, Ursa starts first looking for a target new location in the same rack, then a neighbouring rack and so on. That is, if a maximal migration cost r_{max} of a single object is known a priori, any destination node with the cost higher than r_{max} could be discarded.

Since r_{max} is only known after solving an optimization problem, Ursa starts searching for a target node with the lower bound estimation r and run ILP model. If a feasible solution exists halt the searching; otherwise increase the r by Δr and continue running ILP model.

2.3.2 AutoPlacer

Autoplacer(J. Paiva & Rodrigues 2013) is a system that implements dynamic data placement for distributed key-value stores. The system can be seen as a highly parallel and optimized version of the Ursa system described before. The key features of Autoplacer are the following. The system runs periodically, where in each period it attempts to optimise the placement with regard to the results of the previous optimisation. For each optimisation round, Autoplacer identifies the top-k hotspots, i.e. the objects generating most remote operations for each node of the system. Autoplacer identifies hotspots using Space-Saving Top-k algorithm(Metwally, Agrawal, & El Abbadi 2005) that is executed in parallel by all data servers. Autoplacer only derives a new placement for those items in the top-k list. The new placement is derived by running an optimisation task which finds the appropriate placement for the hotspots. This tasks is decomposed in several parallel subtasks, that can be executed concurrently by different servers, thus speeding the process of deriving the new placement. The optimal location for the items in the top-k list can be arbitrary. To store the mapping between items and servers, Autoplacer uses the Probabilistic Associative Array (PAA) data structure that was briefly introduced earlier in the text.

2.3.3 DRP

The Distributed Greedy Replication (DGR) algorithm(Zaman & Grosu 2011) is an approximation algorithm designed to solve the Distributed Replication Group problem previously introduced in section 2.2.8. The algorithm takes several parameters as inputs, as described next: The first parameter, $\mathbf{r}$ is the matrix of request rates. Each server s_i knows the request rates r_{ij}, $j = 1, ..., n$, of its local users for all objects. Then $\mathbf{r_i} = (\mathbf{r_{i1}}, \mathbf{r_{i2}}, ..., \mathbf{r_{in}})$ denotes the vector of request rates of users at node s_i, and therefore, $\mathbf{r} = (\mathbf{r_1}, \mathbf{r_2}, ..., \mathbf{r_m})^{\mathbf{T}}$, $m \times n$ matrix represents the request rates of all the objects at all servers. The second parameter, $\mathbf{c} = (\mathbf{c_1}, ..., \mathbf{c_m})$ is m-vector of node capacities. The next three parameters, t_r, t_g, and t_l, are the access cost of data items from remote, group and local replicas.

Moreover, the algorithm also uses two additional parameters, the *insertion gain* and the *eviction cost*. The insertion gain, ig_{ij} represents the increase in overall gain the system would undergo if it replicates data object o_j in server s_i. The highest achievable insertion gain is for the object which does not have a replica in the replication group. If it is replicated inside the group, insertion gain reduces to the local gain. Otherwise it is zero. Similarly, the eviction cost, ec_{ij}, is the decrease in the gain the system will

experience if the object o_j is evicted from server s_i. The eviction cost is highest for the objects that have only one replica in the group because evicting this object will make all the servers have remote access cost.

The algorithm works by execution a sequence of phases; in each phase the insertion gain and the eviction cost are calculated in order to characterize each local decision of whether to replicate or evict the object from the server. In making these decisions the algorithm considers the effect of replicating the objects on the overall system gain. The algorithm stops when overall system gain converges.

2.3.4 Distributed Selfish Replication

Distributed Selfish Replication (DSR)(Laoutaris, Telelis, Zissimopoulos, & Stavrakakis 2006), is a distributed algorithm, that is run in a decentralized manner by multiple nodes, in order to create replicas of popular data items. The algorithm uses Game Theory to derive a replica placement strategy that is a Nash equilibrium(Nash 1951; Osborne & Rubinstein 1994). Therefore, contrary to most of the systems considered in this chapter, DSR does not assume that nodes belong to the same administrative domain and are forced to execute a pre-defined algorithm; instead, they can deviate from the algorithm if this allows the nodes to optimize their profit. The algorithm requires each node to know only its local demand pattern and the objects selected for replication by remote nodes, but not the remote demand patterns of other nodes.

2.3.5 Schism

Schism(Curino, Jones, Zhang, & Madden 2010) is an approach to partition the data set among multiple nodes such that aims at supporting load balancing and data locality in the access patterns. For this purpose, it assumes that data accesses are structured in transactions, where each transaction accesses a number of correlated items. The primary goal of Schism is to minimize the number of distributed transactions.

Towards this goal, Schism performs first a workload-driven graph-based partitioning phase, and then an explanation and validation phase. Schism represents a database and its workload using a graph. Data items are mapped to nodes of the graph, and edges are created between items accessed by the same transactions. Afterwards, graph partitioning algorithms are applied in order to find balanced partitions that minimize the weight of cut edges. Consequently, this operation approximately minimizes the distributed transactions and balances the load evenly across the nodes. After the partitioning phase, Schism performs an explanation phase. This phase attempts to find a compact model that captures the (item, partition) mappings produced by the partitioning phase. To perform this task, Schism uses

decision trees (a machine learning classifier) since they produce understandable rule-based output that can be translated directly into predicate-based range partitioning.

2.3.6 Everest

Everest(Narayanan, Donnelly, Thereska, Elnikety, & Rowstron 2008) is a system that relocates data items in a distributed storage system in order to avoid bottleneck in the access to disk. Everest offloads bursty I/O workloads from a primary storage volume to temporary short term persistence stores. Everest runs as a middleware between application and base volume which is the block storage such as single disk, array of disks or solid-state storage. It redirects all read and write requests to the base volume and monitors the performance of that volume. If the load of the volume increases beyond a predefined threshold, Everest off-loads the writes into a virtual store which is created by pooling idle bandwidth and spare capacity on existing data volumes. When the load peak diminishes, Everest lazily reclaims the data in the background from the virtual store to the base volume. Everest is designed to handle short-term peaks only. Long-term changes in load should be addressed by provisioning the storage system that suits the new workload patterns.

2.3.7 Flat Datacenter Storage

Flat Datacenter Storage (FDS)(Nightingale, Elson, Hofmann, Suzue, Fan, & Howell 2012) is a datastore that uses a novel network design that reduces the differences between remote and local data accesses in a data center. In this model there are no concept of "local" disks; every storage is considered as remote even if they are co-located with the computation. By spreading data over disks uniformly at a relatively fine grain, FDS statistically multiplexes I/O over all of the disks in a cluster. Thus, it eliminates the need to satisfy locality constraints in the storage systems.

In FDS, data is stored in blobs. Each blob is named by byte sequence of 128-bit GUID. FDS uses a metadata server that stores the location of the data blocks. However, it is role is simple and limited; metadata collects the list of active servers and distributes it to clients. This list is called the tract locater table (TLT). To read or to write to a specific tract number i from a blob with GUID g, a client selects an entry from TLT by computing an index as foloows: Tract_Locater=(Hash(g) + i) mod TLT_Length. Since tractserver lookup is deterministic, readers and writes rendezvous as long as they have the same TLT. Because TLT changes only when there is a cluster reconfiguration or failures it can be cached by clients for a long time.

2.3.8 HyperDex

HyperDex (Escriva, Wong, & Sirer 2012) is a distributed key-value store where objects can be queried using not only the key of an object but also other attributes of an object. Therefore, HyperDex aims at adding more flexibility to the type of queries supported by key-value stores without hindering their performance. Towards this goal, HyperDex combines two techniques: (a) hyperspace hashing, in which objects with multiple attributes are mapped into a multidimensional hyperspace; (b) space partitioning, which consists in dividing the hypersape in regions that can be mapped on physical servers. Furthermore, for fault-tolerance, regions may be replicated using chain replication(van Renesse & Schneider 2004).

Hyperspace hashing represents each table inside the datastore as a multidimensional space, or hyperspace, where each dimensional axis corresponds directly to one attribute of a table. An object is mapped to a deterministic coordinate in this space by hashing each of its attribute values to a location along the corresponding axis. Servers in the datastore are mapped and responsible for a one of the volumes of hyperspace, which are determined by dividing hyperspace into non-overlapping regions. Each server stores the objects that fall within the corresponding region. Moreover, in order to perform search using several attributes, a search operation needs to specifies a set of attributes and values that they must match. Afterwards, servers whose regions intersect with the search operation are identified and only those servers are contacted to execute the search operation.

Space partitioning decreases the number of servers that are responsible to maintain the hyperspace inside the datastore. Although space construction significantly minimizes the number of servers needed to query when searching, increasing a dimension exponentially grows the hyperspace volume. HyperDex divides the tables with many attributes into multiple lower-dimensional hyperspaces, called subspaces. Each of these subspaces uses a subset of attributes as its dimensional axes and mapped to a servers which stores the objects that has an attribute mapping to region of this subspace. Therefore, by using partitioning, HyperDex decreases the number of servers necessary to maintain a table.

Since subspace partitioning mapping creates more replicas for each object and the position of an object is determined by its contents, objects will mapped to several servers and they will change whenever they are updated. HyperDex arranges object replicas into value-dependent chain whose members are deterministically chosen based on the object's hyperspace coordinate. The head of chain is called the point leader, a server who is responsible for a subspace where the hash of an object's key maps. The subsequent servers are determined by hashing attribute values for each of remaining subspaces. The point leader of an object is responsible for dictating the total order of updates to that object. Each update flows from the point leader through the chain, and remains pending until an acknowledgement of that update is received from the next server in the chain. When an update reaches the tail, the tail sends an acknowledgement back through the chain in reverse so that all other servers may commit the pending update and clean up transient state.

In HyperDex, there is an external component called the coordinator, which is responsible for dividing the hyperspace into regions and assigning regions to servers. The main goal of the coordinator is to maintain the hyperspace and disseminate it to servers and clients. When the servers fail or new servers are joining, the coordinator is responsible for the updated hyperspace mapping. The coordinator is a logically centralized component that is replicated for fault-tolerance.

2.3.9 Data Droplets

DataDroplets(Vilaca, Oliveira, & Pereira 2011) is a distributed key-value store which implements a correlation-aware data placement strategy. The DataDroplets adopts a simple data model where each data object is organized into disjoint collections of items identified by strings, i.e., each data objects consists of a unique identifier and set of free-form string tags. DataDroplets aims to achieve efficient data placement for applications where data objects are arbitrarily related and can be retrieved using their tags (such as Twitter). Towards this goal, DataDroplets uses Space-Filling Curves (SFC) as a mapping function in which tags define the dimensions of the multidimensional space. This makes objects with the same tags to be placed in the logically close nodes; therefore, improves the locality of the requests which uses the tags as a query. DataDroplets employs dynamic load redistribution on membership changes. That is, periodically, a randomly chosen node contacts its successor in the ring structure to perform mutual load balancing(Karger & Ruhl 2004).

2.3.10 Squid

Squid(Schmidt & Parashar 2004) is a peer-to-peer information discovery system that supports complex queries containing partial keywords, wildcards or ranges. Squid strives to ensure that all existing data items matching a given query can be returned with bounded cost, in terms of number of nodes contacted and messages exchanged to perform the query. Similar to DataDroplets, Squid builds on top of structured overlay network 2.2.1, for its data lookup mechanism. Moreover, it uses Space Filling Curves (SFC) as an indexing scheme in order to map the data objects to nodes using keywords. As a results, given the query, only those items that match all the keywords in query are retrieved.

2.4 Discussion and Comparison

Table 2.2 captures the main features of the systems surveyed in this chapter. We can classify these systems into two main categories; systems using static data placement techniques and systems using dynamic data placement techniques. The latter can be seen as an evolution of the former, where data placement is recomputed periodically or when a significant change in the workload is detected.

Systems	Techniques	Topology Aware Replication	Replica Lookup	Architecture
Ursa	ILP	✓	Central	Centralized
AutoPlacer	ILP + Consistent Hashing	✓	Local + PAA	Decentralized
DRP	ILP + Dist. Replication Group	✓	Local	Decentralized
Schism	Graph Partitioning	×	-	Centralized
DSR	Game Theory + Greedy Approach	×	Local + Bloom Filters	Decentralized
Everest	-	×	Local	-
FDS	Consistent Hashing	×	Local	-
HyperDex	Hyperspace Hashing	×	Coordinator	Decentralized
DataDroplets	Space Filling Curves + Consistent Hashing	×	Local	Decentralized
Squid	Space Filling Curves + Consistent Hashing	×	Local	Decentralized

Table 2.2: Comparison of existing systems

We identify two main classes of systems that use static data placement techniques:

- Systems that make use of space filling curves in order to cluster related data objects in the set of nodes which improves the data locality. These systems are based on consistent hashing techniques for data lookup methods and therefore are highly scalable.

- Systems that make use of hyperspace hashing technique. These systems provide flexible data retrieval methods. Clients can search for data items using secondary attributes other than primary key of the data object. However, the replica lookup method is centralized. That is global membership of servers accessed from a fault-tolerance centralized component, namely coordinator.

Concerning the systems that make use of dynamic data placement techniques, we can identify three main categories:

- Systems that perform topology aware replication. That is replicate most popular objects closer to their clients. Moreover, balance the load of hot-spot nodes to the nearest nodes which will incur less migration costs.

- Systems that use different data lookup mechanisms. Some of them use lightweight data structures such PAA and Bloom filters for data lookup methods. This improves scalability of the system; however there is extra bandwidth cost related to dissemination of these data structures to the other nodes. On the other hand, some of these systems employ central directory for data lookup methods which introduces extra costs along the critical execution path of data access operations.

- Centralized vs decentralized systems. The former systems depend on the centralized node for analysis of the cost functions and decisions of the data placement techniques such as Integer Linear Programming. The latter systems calculate the cost functions locally and disseminate the decisions among themselves incurring extra communication steps.

Summary

In this chapter we introduced the related work in the area of data placement for key-value stores. We started by introducing the key data placement techniques that are required to understand the rest of the document. Then we briefly surveyed some of the most relevant practical systems that are related to our work.

3 The HyperDex System

In the previous chapter we gave the survey of the state of the art practical systems that solve the data placement problem. Among them HyperDex(Escriva, Wong, & Sirer 2012) provides a unique set of interesting features, including support for complex queries. Therefore, we have decided to study data locality-aware partitioning schemes for HyperDex.

In this chapter we describe the HyperDex in more detail. Then we will present different configuration options that can be used when deploying HyperDex. Finally we show that different configurations can result in different performance results.

3.1 HyperDex Description

HyperDex is a high-performance, scalable, consistent, and distributed key-value store that provides rich API. When using HyperDex, clients can perform range queries and partial searches using the secondary attributes of the data objects without building secondary indexes or contacting all the system nodes. The main idea is that, by using Hyperspace Hashing, the system can deterministically determine what is the smallest set of nodes which may contain data matching any given query. After determining that set, the client sends a message to each node containing the query, which is processed locally by each of them and combined in the client into a reply.

3.1.1 Hyperspace Hashing

Hyperspaces are the building blocks 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 into 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 only determine which regions are matched by the query, and contact their corresponding nodes to request the objects.

3.1.2 Subspace Partitioning

Hyperspace hashing facilitates efficient search by significantly reducing the number of servers to contact for each partially specified query. However, an hyperspace increases in volume exponentially with each additional secondary 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, given that some regions of this volume would be sparse and others more populated.

To avoid the problem above, a strategy that is typically followed (and that is also used in HyperDex) consists in dividing the volume in many small logical regions, and then distribute 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 easiest way to perform this division is divide each axis in d segments, and let each virtual region be defined by a different segment in each dimension. This raises the additional problem of managing an extremely large set of logical regions. In fact, using this approach, the number of regions for a hyperspace with N dimensions is proportional to $O(d^N)$, where d is the number of equally divided intervals along one dimension of hyperspace.

Unfortunately, this division is oblivious to the query profile, and may result in sub-optimal performance, as many servers may need to be contacted to satisfy a query. To illustrate further why this may be problematic, consider that $N = 10$, this means that even if the search query contacts only 10% of the regions, it still needs to send around 100 search requests and possibly contact 100 different servers.

The solution for this "curse of dimensionality" (Bellman 2003) 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 $O(d^{|\mathcal{A}|})$, when using subspaces they become proportional to $O(|\mathcal{S}| \times d^{|\mathcal{S}_i|})$, where d is the number of equally divided intervals for each dimension of subspace.

3.2 HyperDex Configuration

The subspace configurations of hyperspace in HyperDex must be performed manually. Moreover, the hyperspace configuration is static, in the sense that is it needs to be configured before the deployment.

Consider, for the following discussion, an application containing hotels information where each hotel data is characterized by a primary key and secondary attributes: city, star-rating, and price. For this data set, HyperDex would create a three dimensional hyperspace where attribute city constitutes the x-axis, attribute stars constitutes the y-axis and price attribute constitutes the z-axis. Figure 3.1 illustrates this configuration schema.

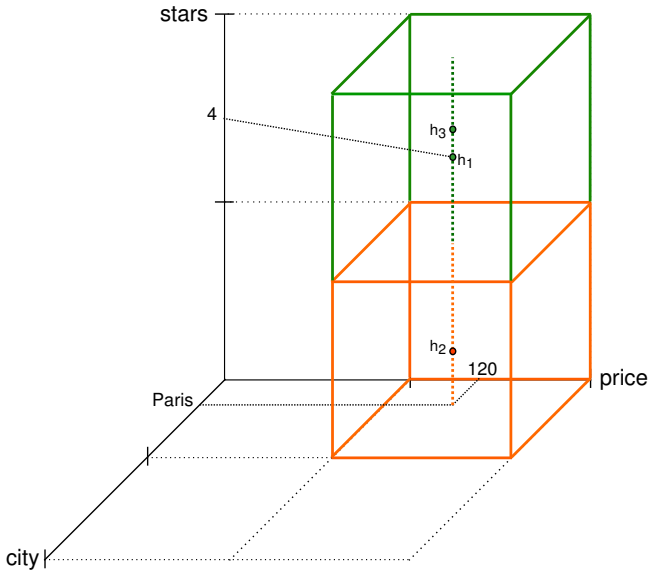
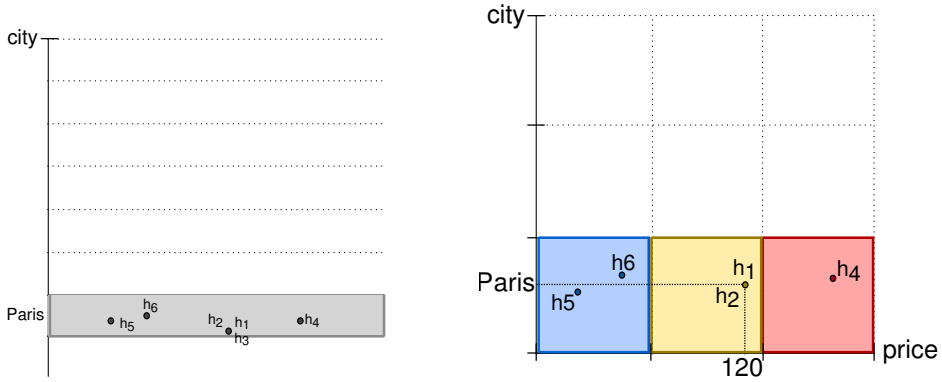


Figure 3.1: A three dimensional hyperspace with attributes $\langle city, price, stars \rangle$.

However, as mentioned previously we can divide the hyperspace into smaller *subspaces*. In Figure 3.2 we show two possible subspaces. In the Figure 3.2a we illustrate a subspace with only one attribute city, $\mathcal{S}_1 = \langle city \rangle$; whereas in Figure 3.2b we present a subspace with two attributes city and price, $\mathcal{S}_2 = \langle city, price \rangle$.

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



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

Figure 3.2: Two subspaces with different configurations to index hotels.

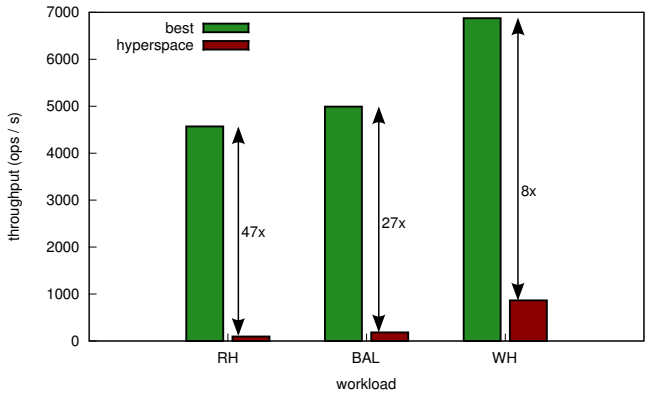


Figure 3.3: Performance of HyperDex when using a single hyperspace against a more complex configuration with subspaces. The results are shown for three variants of the same workload.

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 $\mathcal{S}_1 = \langle city \rangle$ is partitioned such that it has 8 regions. However, subspace $\mathcal{S}_2 = \langle city, price \rangle$ is partitioned into 9 regions because there is no combination of integer partitions for the two dimensions that results in 8 regions.

HyperDex always implicitly creates a 1-dimensional subspace for the key of objects. 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 region of 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. Note that the primary key subspace counts as a subspace on its own.

Notice that each subspace $\mathcal{S}_i \in \mathcal{S}$ can comprise any combination of attributes $\langle \mathcal{A}_1, ..., \mathcal{A}_k \rangle$, and a hyperspace can constitute any combination of those subspaces. That means the number of different hyperspace configurations is proportional to $O(2^{2^{|\mathcal{A}|}})$. Thus, given a application workload finding the most efficient performance configuration becomes *NP-Hard* problem.

In Figure 3.3 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× to 47× when changing the dominance of different operations (Read 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 to understand the inner-workings of HyperDex to be able to predict its performance in an automatic fashion.

In the following sections we describe search and modify operations of the HyperDex.

3.2.1 Search Operation

We now describe how queries are processes in HyperDex. 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 Figure 3.2b, 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 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. This may return only a subset of the data available locally respecting that region if the query was only partially defined with regard to the subspace, for instance when searching only by city in subspace $\langle city, price \rangle$. 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.

3.2.2 Modify Operation

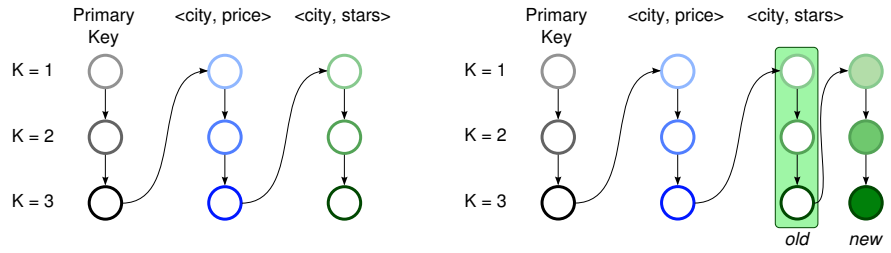
We now describe the modification of existing objects in the HyperDex key-value store. For this the object is fetched (a simple get operation) through the primary key, and later inserted with attributes hav-

ing their corresponding values modified. Consequently, a read operation only uses the (always available) primary key subspace (that only has one dimension), and is unaffected by the subspace configurations. An insertion can thus be seen as 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 considered 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, apart from fault tolerance level, of data.

HyperDex organizes these replicas using chain replication (van Renesse & Schneider 2004) 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. Figure 3.4 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 Figure 3.4a. 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. Figure 3.4b 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.



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

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

Figure 3.4: The chain of servers resulting from two different modification operations in the same configuration of HyperDex.

3.3 HyperDex Performance

In this section we show that different hyperspace configurations yield different performance results for the same query workloads by experimental evaluation.

In the following tests every query that we perform uses values that lead to the result set being empty. This simplification serves the purpose of neglecting the component of the cost of a search that lies in sending the results to the client. This is an orthogonal issue that we shall address in future work. To understand why this simplification eases the reasoning over the experiments we ran the following three experiments always with $\mathcal{S} = \langle city \rangle$: $\mathcal{Q}_1 = \langle city \rangle$; $\mathcal{Q}_2 = \langle city, addr \rangle$; and $\mathcal{Q}_3 = \langle city \rangle$, where the first two queries search for existing values whereas the latter searches for non-existing cities.

The result is that $\mathcal{Q}_1$ returns on average 50 hotels, $\mathcal{Q}_2$ returns 1 hotel, and $\mathcal{Q}_3$ naturally returns 0 hotels. For this reason, the throughput of $\mathcal{Q}_2$ is on average the same as $\mathcal{Q}_3$, and in turn both have 50% more throughput than $\mathcal{Q}_1$ in our evaluation environment. This indicates that there are variable costs associated with the number of results returned by the queries. We leave the modeling of such costs to future work, and hereon assume the simplification for which our synthetic benchmarks never return results.

Let's assume we have the same database with hotels information. Each hotel has primary key and several secondary attributes: *city*, *stars*, *price*, *address*, *postcode* and *telephone number*. Our workload has three queries: $\mathcal{Q}_1 = \langle city \rangle$, $\mathcal{Q}_2 = \langle city, address \rangle$ and $\mathcal{Q}_3 = \langle city, address, postcode \rangle$. Notice that these queries affect only the search operation. We will present the impact of modifications in the following chapters.

In Table 3.1 we show that throughput of the system significantly changes for a different subspace configuration given the queries defined above.

Therefore it is crucial to configure the HyperDex that gains best performance results for a given application. In the following chapters we are going to analyse the search and modify operations in detail

$\mathcal{S}_i$	$Throughput(ops/sec)$		
	$\mathcal{Q}_i = \langle city \rangle$	$\mathcal{Q}_i = \langle city, address \rangle$	$\mathcal{Q}_i = \langle city, address, postcode \rangle$
$\langle \emptyset \rangle$	12	14	11
$\langle city \rangle$	3118	3018	3152
$\langle city, address \rangle$	199	3165	3210
$\langle city, address, postcode \rangle$	88	581	3511
$\langle city, address, postcode, tel \rangle$	46	201	828

Table 3.1: Measured throughput for different queries and different configurations.

and propose a prototype model that results top-k efficient hyperspace configurations for a given workload.

Summary

In this chapter we described the HyperDex system. Firstly we presented main building blocks of HyperDex, hyperspace hashing and space partitioning. Then we described how to configure hyperspace for the system and described the execution of the *search* and *modify* operations. Finally, we showed how different configurations impact the system performance by experimental evaluation.

In the following chapter we present a model that captures the expected performance of HyperDex, with the ultimate goal of automating the HyperDex configuration.

4 Modeling HyperDex Performance

As we have illustrated in the previous chapter, deriving most efficient configuration of the HyperDex can be a challenging task. In this chapter we perform an analysis of the HyperDex operations in detail and derive a performance model model that can be used to predict best HyperDex configurations for a given workload.

We start by making the analysis of search and modify operations in HyperDex. Then we provide a model to predict the performance of each operation in isolation and also of a combination of these operations. Finally we assess the accuracy of our model using synthetic benchmarks.

4.1 General Assumptions for the Performance Model

In this chapter, when we perform the analysis of HyperDex, we make the following assumptions. First, we assume that the system should be configured to sustain the maximum throughput. Second we assume that the system is deployed in a data center with enough network bandwidth such the communication is not the bottleneck of the system operation.[1] As a result, we assume that the bottleneck of the system is the combined CPU/disk usage at each server.

In this scenario, the optimal configuration of HyperDex should attempt to minimize the number of servers involved in processing a given request. This happens because every time an additional server has to process a request, it consumes local resources that could be used to process other requests, reducing the overall throughput.

4.2 Modeling Search Operations

Under the assumptions above, the overall throughput of a read-dominated workload in HyperDex is inversely proportional to the number of regions contacted by each query.

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

Herein we consider a generic search query $\mathcal{Q}_i$. The worst possible performance for a search $\mathcal{Q}_i$ happens whenever $\nexists_{\mathcal{S}_i \in \mathcal{S}} : \mathcal{Q}_i \cap \mathcal{S}_i \neq \emptyset$. Since no subspace contains (at least) one attribute being searched, then the query must contact (i.e., all) $\mathcal{R}_i$ regions 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.

On the other hand, 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 $\mathcal{Q}_i$. This is due to fact that 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.

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:

$$\mathcal{CR}^{exp}(\mathcal{Q}_i) = \sqrt[|\mathcal{S}_i|]{\mathcal{R}_i}^{|E|} \quad \text{such that: } E = \mathcal{S}_i \setminus \mathcal{Q}_i \tag{4.1}$$

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 3.1.2, 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 (4.1)) 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.

The expected throughput of a search query $\mathcal{Q}_i$ that uses some subspace $\mathcal{S}_i$ is obtained by:

$$T^{exp}(\mathcal{Q}_i) = \frac{1}{cost(\mathcal{Q}_i)} \qquad \text{where: } cost(\mathcal{Q}_i) = \sqrt[|\mathcal{S}_i|]{\mathcal{R}_i}^{|E|} \times \frac{\mathcal{O}}{\mathcal{R}_i} \times \beta \tag{4.2}$$

We note that Equation 4.1 is consistent with the results stated before:

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

4.3 Modeling Modify Operations

From the description of modify operations that we have included in Section 3.2.2, it is possible to infer that the cost of those operations is proportional to the length of the chain. In this section we use this insight to model the cost of a modification.

We start by addressing the function that calculates the length of the chain. 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 Figure 3.4a 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 Figure 3.4b. 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 we capture the length of the chain of servers for some modification $\mathcal{Q}_i$ by the following function:

$$length(\mathcal{Q}_i) = \mathcal{K}(1 + |\mathcal{N}| + 2|\mathcal{M}|) \tag{4.3}$$

Using Equation 4.3 and a value for a measured maximum throughput T_{max} (obtained when $length(\mathcal{Q}_i) = 1$, e.g. by modifying an object in an hyperspace configured only with the key subspace), one would expect to be able to derive the expected throughput of a query $\mathcal{Q}_i$ as:

$$T^{exp}(\mathcal{Q}_i) = \frac{T_{max}}{length(\mathcal{Q}_i)} \tag{4.4}$$

There are, however, two additional considerations that we must take into account to correctly model the expected throughput of a modification $\mathcal{Q}_i$. So far we have only mentioned how HyperDex modifies the data; but a modification assumes a previous fetch of the data item being modified. In every experiment

we do so by using a read by the primary (unique) key, which need only to contact one server. Therefore we need to add one extra server to the denominator of the equation.

In addition to this, recall that we assume the bottleneck of the system to be the servers' local processing. This fact leads Equation 4.4 to return an inaccurate prediction. This is due to the operation of overwriting an existing object in an HyperDex server's persistent storage being considerably faster than that of deleting or creating it. Hence, $2|\mathcal{M}| \times K$ nodes incur in an extra cost, which we encapsulate in a correcting factor α. This factor, similarly to β, is dependant on the hardware configuration and HyperDex implementation, and must be estimated from a running system. Based on these two corrections, a more accurate estimation of the throughput of the system for a given query $\mathcal{Q}_i$ can be obtained from Equation 4.5.

$$T^{exp}(\mathcal{Q}_i) = \frac{T_{max}}{1 + \mathcal{K}(1 + |\mathcal{N}| + 2\alpha|\mathcal{M}|)} \tag{4.5}$$

In the following sections we quantify both β and α costs.

4.4 Modeling Hybrid Workloads

In this section we model hybrid workloads, with a mix of search and modify operations, where each operation occurs with some probability p_i.

So far we have only considered configurations where $\mathcal{S}$ contains one single subspace. When there is more than one subspace in the configuration, HyperDex performs pre-processing to determine which subspace allows for contacting the fewer number of regions for a given search query. Hence, the query performance depending on several subspaces should be equal to that of the most fit subspace for the search query.

Similarly when there are several subspaces the modification operation changes the region of a subspace whose attributes match the query attributes; otherwise it only overwrites.

We now consider combined queries for a single operation where there may exist several search or modify queries $\mathcal{Q}$, where 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 (Equations (4.2), (4.5)) of each query as in the following equation:

$$T^{exp}(\mathcal{Q}_i) = \frac{1}{\sum_{i=0}^{|\mathcal{Q}^{\mathcal{S}}|} (cost(\mathcal{Q}_i) \times p_i)} \tag{4.6}$$

Finally given the complex workloads where search and modify operations are intermixed with some probability p_i, we can model the overall cost in the same fashion using the linear combination of the each operation cost. Thus we can predict the overall system throughput when we have mixed workloads using the following equation:

$$T^{overall} = \frac{1}{\sum_{i=0} \frac{p_i}{T^{exp}(\mathcal{Q}_i)}} \tag{4.7}$$

4.5 Assessing the Accuracy of the Model

In this section we assess the accuracy of our model. First we describe our experimental setup. Next we present our synthetic benchmarks. Finally we address the accuracy of the equations (4.2) and (4.5), first estimating the correction factors β and α and then assess the throughput estimated by our model by comparing them with measured throughput results.

4.5.1 Experimental Setup

As a benchmarking hardware, we used 9 virtual machines, running on Xen. Each machine is equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) E5506 processors and 40 GB Ram, running Linux 2.6.32-33-server and interconnected via a private Gigabit Ethernet. For our experimental study we used two benchmarks: synthetic benchmark and real hotels benchmark. In this chapter we present the results of synthetic benchmarks. Moreover, 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.

4.5.2 Experimental Results

In the following sections 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 sections.

Search Operation

We now present some examples of searches and respective configurations. These examples will be useful in the following sections, as we demonstrate the accuracy of the previous equations. We use abstract identifiers for the subspaces and queries as we shall instantiate different experiments for each example.

The following are the descriptions of the each example presented in Table 4.1.

Description	$\mathcal{Q}$	$\mathcal{S}$	$\lvert\mathcal{O}\rvert$	$\lvert\mathcal{R}\rvert$	E
1 - *query == subspace*	$\langle a, b\rangle$	$\langle a, b\rangle$	256000	256	$\lvert\mathcal{S} \setminus \mathcal{Q} = \emptyset\rvert = 0$
2 - *query* $\cap$ *subspace* $= \emptyset$	$\langle a, b\rangle$	$\langle c, d\rangle$	256000	256	$\lvert\mathcal{S} \setminus \mathcal{Q} = \langle c, d\rangle\rvert = 2$
3 - *query* $\not\subseteq$ *subspace*	$\langle a, b\rangle$	$\langle b, c\rangle$	256000	256	$\lvert\mathcal{S} \setminus \mathcal{Q} = \langle c\rangle\rvert = 1$
4 - *query* $\not\subseteq$ *subspace*	$\langle a, b\rangle$	$\langle b, c, d\rangle$	256000	256	$\lvert\mathcal{S} \setminus \mathcal{Q} = \langle c, d\rangle\rvert = 2$
5 - *query* $\not\subseteq$ *subspace*	$\langle a, b\rangle$	$\langle b, c, d, e\rangle$	256000	256	$\lvert\mathcal{S} \setminus \mathcal{Q} = \langle c, d, e\rangle\rvert = 3$

Table 4.1: Search example scenarios

1. In this scenario search query will touch $(\sqrt[2]{256})^0 \times \frac{256000}{256} = 1000 = \frac{|\mathcal{O}|}{|\mathcal{R}|}$ objects.

2. In this example search query will touch $(\sqrt[2]{256})^2 \times \frac{256000}{256} = 256000 = |\mathcal{O}|$ objects.

3. This search query will touch $(\sqrt[2]{256})^{2-1} \times \frac{256000}{256} = 16000 = 16 * \frac{|\mathcal{O}|}{|\mathcal{R}|}$ objects. Since there are 256 regions, each attribute (axis of the subspace) is partitioned into $\sqrt[2]{(256)} = 16$ regions. So, since only one of the attributes is unnecessary and for the other one the query is fully specified, it will touch precisely those 16 regions. The formula confirms this: $(\sqrt[2]{256})^{2-1} = 16$.

4. This search query will touch $(\sqrt[3]{256})^{3-1} \times \frac{256000}{256} = 40317 = 40 * \frac{|\mathcal{O}|}{|\mathcal{R}|}$ objects. Because there are 256 regions, each attribute is partitioned into $\sqrt[3]{(256)} = 6.35$ regions. Additionally, two of the attributes are unnecessary and for the other one the query is fully specified, it will touch 6.35×6.35 regions.

5. In this scenario the search query will touch $(\sqrt[4]{256})^{4-1} \times \frac{256000}{256} = 64000 = 64 * \frac{|\mathcal{O}|}{|\mathcal{R}|}$ objects. Since there are 256 regions, each attribute is partitioned into $\sqrt[4]{(256)} = 4$ regions. Moreover, three of the attributes are unnecessary and for the other one the query is fully specified, it will touch $4 \times 4 \times 4$ regions.

Now we address the accuracy of Equation (4.2). For this we shall estimate the correction factor β using actual measurements, and then assess our throughput estimations by comparing them with increasingly complex search scenarios. Since all tests were executed in the same hardware configuration,

and β depends only on said configuration, we expect that if Equation (4.2) holds, then we should be able to estimate a constant value for β across all experiments.

By easily manipulating Equation (4.2), we can obtain an estimation of β as long as we have available some real throughput measurements. Table 4.2 presents the throughput and estimated β for different configurations that fit the previous five examples. In the following experiments we number each scenario accordingly to the example that it matches for ease of understanding. The experiments are divided in three sub-tables as we use queries with a large number of attributes specified.

In these results, the estimated β has an average value of $3.31 \times 10^{-7} \pm 5$, where very few experiments fall off over $3.31 \times 10^{-7} \pm 5\%$. In fact, the scenarios where β is most different from the average are scenarios with very small throughput. Therefore we interpret the variance on β as fluctuations. An additional explanation for some errors lies in the partition of spaces in regions. Looking at configuration 4 in the first sub-tables, it is the only one there whose $\sqrt[|\mathcal{S}_i|]{|\mathcal{R}_i|}$ does not result in an integer result (as $\sqrt[3]{256}$ is not an integer value). This causes the 3 axis of the subspace to be partitioned into 7, 7, and 6 regions respectively, while our model uses a formula which accepts non-integer values for simplification. In the future we shall more adequately fit our model to this particularity, but for now, since the results demonstrate the error introduced by this parameter is small, it shall not be deemed as too problematic.

Description	$\mathcal{S}$	$\mathcal{Q}$	**Throughput**	**Est.** β
1 - $\mathcal{Q} = \mathcal{S}$	$\langle city \rangle$	$\langle city \rangle$	3118	3.22×10^{-7}
2 - $\mathcal{Q} \cap \mathcal{S} = \emptyset$	$\langle \emptyset \rangle$	$\langle city \rangle$	12	3.29×10^{-7}
3 - $\mathcal{Q} \subset \mathcal{S} \wedge E = 1$	$\langle city, addr \rangle$	$\langle city \rangle$	199	3.15×10^{-7}
4 - $\mathcal{Q} \subset \mathcal{S} \wedge E = 2$	$\langle city, addr, post \rangle$	$\langle city \rangle$	88	2.96×10^{-7}
5 - $\mathcal{Q} \subset \mathcal{S} \wedge E = 3$	$\langle city, addr, post, tel \rangle$	$\langle city \rangle$	46	3.40×10^{-7}
1.1 - $\mathcal{S} \subset \mathcal{Q}$	$\langle city \rangle$	$\langle city, addr \rangle$	3018	3.33×10^{-7}
1.2 - $\mathcal{S} \subset \mathcal{Q}$	$\langle city, addr \rangle$	$\langle city, addr \rangle$	3165	3.17×10^{-7}
2 - $\mathcal{Q} \cap \mathcal{S} = \emptyset$	$\langle \emptyset \rangle$	$\langle city, addr \rangle$	14	2.85×10^{-7}
3 - $\mathcal{Q} \subseteq \mathcal{S} \wedge E = 1$	$\langle city, addr, post \rangle$	$\langle city, addr \rangle$	581	2.99×10^{-7}
4 - $\mathcal{Q} \subseteq \mathcal{S} \wedge E = 2$	$\langle city, addr, post, tel \rangle$	$\langle city, addr \rangle$	201	3.12×10^{-7}
1.1 - $\mathcal{S} \subset \mathcal{Q}$	$\langle city \rangle$	$\langle city, addr, post \rangle$	3152	3.18×10^{-7}
1.2 - $\mathcal{S} \subset \mathcal{Q}$	$\langle city, addr \rangle$	$\langle city, addr, post \rangle$	3210	3.13×10^{-7}
1.3 - $\mathcal{Q} = \mathcal{S}$	$\langle city, addr, post \rangle$	$\langle city, addr, post \rangle$	3511	3.28×10^{-7}
1.4 - $\mathcal{Q} = \mathcal{S}$	$\langle city, addr, post \rangle$	$\langle city, addr, post, tel \rangle$	3672	3.14×10^{-7}
1.5 - $\mathcal{Q} = \mathcal{S}$	$\langle city, addr, post, tel \rangle$	$\langle city, addr, post, tel \rangle$	3150	3.19×10^{-7}
2 - $\mathcal{Q} \cap \mathcal{S} = \emptyset$	$\langle \emptyset \rangle$	$\langle city, addr, post \rangle$	11	3.51×10^{-7}
3 - $\mathcal{Q} \subseteq \mathcal{S} \wedge E = 1$	$\langle city, addr, post, tel \rangle$	$\langle city, addr, post \rangle$	828	3.03×10^{-7}

Table 4.2: Real throughput and estimated β for configurations where we query for one random value.

Using this estimated β we show our estimated throughput (using Equation (4.2)) and the respective measured throughput for each of the previous experiments in Figure 4.1. We used the same identifier (albeit non unique, but in the same sequence) as in Table 4.2, which we recall points to the example on which the scenario matches. These results show a very accurate prediction, with an absolute average error of 5.6% and a standard deviation of 4%.

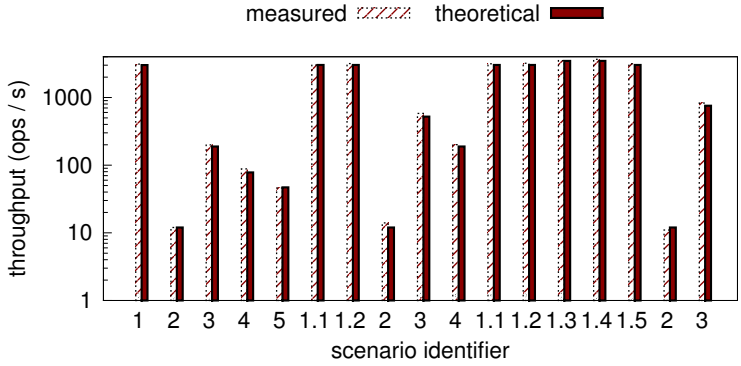


Figure 4.1: Comparison of estimation vs measured throughput for searches using Equation (4.2) for the scenarios in Table 4.2.

We also highlight another important experimental result. Recall hypothesis we derived, which summarized states that having $E = 0$ for some search query leads to the optimal performance, regardless of the characteristics of the subspace chosen. This flexibility allows performing a search over a subspace with different dimensions, as long as they are all included in the query — in other words, as long as there are no *extra* dimensions that are useless for the query.

In fact, if we analyze the results in Table 4.2 and Figure 4.1, we can confirm that given our experimental data. Namely, the experiments 1.1 and 1.2 (in the third sub-table) get approximately the same throughput (around 3150 ops/s), albeit the subspace used is different for the same query. But regardless of the difference in the subspace, it never has useless dimensions, thus confirming empirically our theoretical analysis.

As we mentioned in section 4.4 when there is more than one subspace in the configuration, HyperDex performs pre-processing to determine the most fit subspace for the query.

We conducted another set of experiments to verify this hypothesis. Table 4.3 shows the throughput of different queries in configurations with more than one subspace. The value of β was estimated using the subspace most fit for the query, i.e., the one that produces the least number of contacted regions according to Equation (4.1). Since β is close to the expected value for our hardware and network configuration, we confirm our hypothesis on this matter.

Once again we show the estimated throughput using Equation (4.2), but this time for the scenarios in Table 4.3. Figure 4.2 depicts an accurate prediction, whose average absolute error is of 4.6% with a standard deviation of 4.5%. This also strengthens our conclusions about the availability of multiple subspaces and the respective choice for a search.

We now consider more complex workloads where there may exist several search queries $\mathcal{Q}$, where

Description	S	Q	**Throughput**	**Est.** β
1.1 - $\mathcal{Q} = \mathcal{S}$	$\langle city \rangle, \langle city, addr \rangle$	$\langle city \rangle$	2948	3.41×10^{-7}
3.1 - $\mathcal{Q} \subset \mathcal{S} \wedge E = 1$	$\langle city, addr \rangle, \langle addr, post \rangle$	$\langle city \rangle$	184	3.41×10^{-7}
3.2 - $\mathcal{Q} \subset \mathcal{S} \wedge E = 1$	$\langle city \rangle, \langle city, addr \rangle$	$\langle addr \rangle$	162	3.88×10^{-7}
3.3 - $\mathcal{Q} \subset \mathcal{S} \wedge E = 1$	$\langle city, addr \rangle, \langle addr, post \rangle$	$\langle addr \rangle$	180	3.48×10^{-7}
1.2 - $\mathcal{Q} = \mathcal{S}$	$\langle city \rangle, \langle city, addr \rangle$	$\langle city, addr \rangle$	2922	3.44×10^{-7}
1.3 - $\mathcal{Q} = \mathcal{S}$	$\langle city, addr \rangle, \langle addr, post \rangle$	$\langle city, addr \rangle$	3047	3.30×10^{-7}
5.1 - $\mathcal{S} \subset \mathcal{Q}$	$\langle city \rangle, \langle city, addr \rangle$	$\langle city, addr, post \rangle$	2922	3.44×10^{-7}
5.2 - $\mathcal{S} \subset \mathcal{Q}$	$\langle city, addr \rangle, \langle addr, post \rangle$	$\langle city, addr, post \rangle$	3113	3.23×10^{-7}
5.3 - $\mathcal{S} \subset \mathcal{Q}$	$\langle city \rangle, \langle city, addr \rangle$	$\langle city, addr, post, tel \rangle$	2946	3.41×10^{-7}
5.4 - $\mathcal{S} \subset \mathcal{Q}$	$\langle city, addr \rangle, \langle addr, post \rangle$	$\langle city, addr, post, tel \rangle$	3233	3.11×10^{-7}

Table 4.3: Real throughput and estimated β for configurations with two subspaces in each configuration.

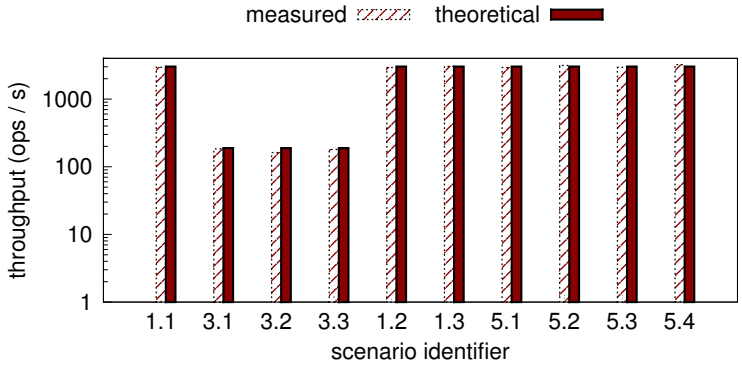


Figure 4.2: Comparison of estimation vs measured throughput for searches using Equation (4.2) for the scenarios in Table 4.3.

each query $\mathcal{Q}_i$ occurs with some likelihood p_i. Where we prove the accuracy of the Equation (4.6).

Table 4.4 shows the measured throughput and estimated β for scenarios where we perform mixed queries. In each test, each query in $\mathcal{Q}^S$ had the same probability, i.e. $\forall p_i \in \mathcal{Q}^S : p_i = \frac{1}{|\mathcal{Q}^S|}$.

This time we note that the estimated β varies more (due to the more complex workloads). Still, the average β from all workloads in this experiment (not only the ones presented in Table 4.4 is of 3.26×10^{-7}, and the standard deviation is once again of approximately 10%.

We additionally present the same scenarios in Figure 4.3, by showing our prediction. Note that we still used the initially estimated β of 3.31×10^{-7}. As a result, we obtain an average absolute estimation error of 5.28% and a standard deviation for the error of 3% (for all workloads, not only the ones presented). These results attest the accuracy of the linear combination of costs for the search queries.

$\mathcal{S}$	$\mathcal{Q}$	**Throughput**	**Est.** β
$\emptyset$	$\mathcal{Q}_1 = \langle city \rangle, \mathcal{Q}_2 = \langle city, addr \rangle$	14	2.88×10^{-7}
$\langle city \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2$	2870	3.50×10^{-7}
$\langle city, addr \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2$	360	3.28×10^{-7}
$\langle city, addr, post \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2$	141	3.22×10^{-7}
$\langle city, addr, post, tel \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2$	81	3.10×10^{-7}
$\langle city \rangle, \langle city, addr \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2$	3263	3.08×10^{-7}
$\langle city, addr \rangle, \langle addr, post \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2$	327	3.61×10^{-7}
$\emptyset$	$\mathcal{Q}_1, \mathcal{Q}_3 = \langle city, addr, post \rangle$	14	2.79×10^{-7}
$\langle city \rangle$	$\mathcal{Q}_1, \mathcal{Q}_3$	3169	3.17×10^{-7}
$\langle city, addr \rangle$	$\mathcal{Q}_1, \mathcal{Q}_3$	364	3.24×10^{-7}
$\langle city, addr, post \rangle$	$\mathcal{Q}_1, \mathcal{Q}_3$	166	3.07×10^{-7}
$\langle city, addr, post, tel \rangle$	$\mathcal{Q}_1, \mathcal{Q}_3$	93	3.17×10^{-7}
$\langle city \rangle, \langle city, addr \rangle$	$\mathcal{Q}_1, \mathcal{Q}_3$	3434	2.92×10^{-7}
$\langle city, addr \rangle, \langle addr, post \rangle$	$\mathcal{Q}_1, \mathcal{Q}_3$	390	3.03×10^{-7}
$\emptyset$	$\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3$	12	3.15×10^{-7}
$\langle city \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3$	3090	3.25×10^{-7}
$\langle city, addr \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3$	551	3.06×10^{-7}
$\langle city, addr, post \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3$	222	3.03×10^{-7}
$\langle city, addr, post, tel \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3$	110	3.29×10^{-7}
$\langle city \rangle, \langle city, addr \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3$	3315	3.03×10^{-7}
$\langle city, addr \rangle, \langle addr, post \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3$	552	3.06×10^{-7}
$\emptyset$	$\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4 = \langle city, addr, post, tel \rangle$	12	4.55×10^{-7}
$\langle city \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4$	3149	4.25×10^{-7}
$\langle city, addr \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4$	655	3.41×10^{-7}
$\langle city, addr, post \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4$	296	3.00×10^{-7}
$\langle city, addr, post, tel \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4$	142	3.36×10^{-7}
$\langle city \rangle, \langle city, addr \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4$	3173	4.22×10^{-7}
$\langle city, addr \rangle, \langle addr, post \rangle$	$\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4$	655	3.41×10^{-7}

Table 4.4: Real throughput and estimated β for configurations where we query for random values for three and four random attributes.

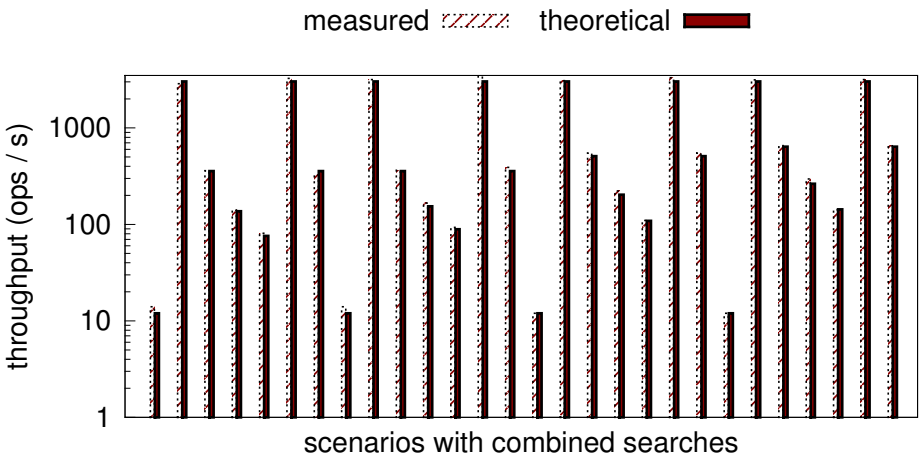


Figure 4.3: Comparison of estimation vs measured throughput for workloads with combined searches using Equation (4.6) for the scenarios in Table 4.4.

Modify Operation

We present some examples in Table 4.5 representative of different cases that will be helpful to confirm the accuracy of Equation (4.5) presented in the section 4.3.

Example	$\mathcal{Q}$	$\mathcal{S}$	$\mathcal{N}$	$\mathcal{M}$	**Chain Length**
1 -	$\langle city \rangle$	$\emptyset$	$\emptyset$	$\emptyset$	$1 + \mathcal{K}$
2 -	$\langle city \rangle$	$\langle addr \rangle$	$\langle addr \rangle$	$\emptyset$	$1 + 2\mathcal{K}$
3 -	$\langle city \rangle$	$\langle city \rangle$	$\emptyset$	$\langle city \rangle$	$1 + 3\mathcal{K}$
4 -	$\langle city \rangle$	$\langle city \rangle, \langle addr \rangle$	$\langle addr \rangle$	$\langle city \rangle$	$1 + 4\mathcal{K}$
5 -	$\langle city, addr \rangle$	$\langle city \rangle, \langle addr \rangle$	$\emptyset$	$\langle city \rangle, \langle addr \rangle$	$1 + 5\mathcal{K}$
6 -	$\langle city \rangle$	$\langle city \rangle, \langle addr \rangle, \langle post \rangle$	$\langle addr \rangle, \langle post \rangle$	$\langle city \rangle$	$1 + 5\mathcal{K}$
7 -	$\langle city, addr \rangle$	$\langle city \rangle, \langle addr \rangle, \langle post \rangle$	$\langle post \rangle$	$\langle city \rangle, \langle addr \rangle$	$1 + 6\mathcal{K}$

Table 4.5: Modify example scenarios

1. In this example, there is no subspace configuration, so the chain length becomes, $1 + \mathcal{K} \times (1 + 0 + 2 \times 0) = 1 + \mathcal{K}$.

2. In the second example, there is a subspace and modification does not affect it, so the chain length becomes, $1 + \mathcal{K} \times (1 + 1 + 2 \times 0) = 1 + 2\mathcal{K}$.

3. In next example, modifications changes only a single subspace, so the chain length becomes, $1 + \mathcal{K} \times (1 + 0 + 2 \times 1) = 1 + 3\mathcal{K}$.

4. In this example, there are two subspace configurations and only one is modified, so the chain length is equal to, $1 + \mathcal{K} \times (1 + 1 + 2 \times 1) = 1 + 4\mathcal{K}$.

5. Next example has two subspaces and both of them are modified, chain length becomes, $1 + \mathcal{K} \times (1 + 0 + 2 \times 2) = 1 + 5\mathcal{K}$.

6. This example has thee subspaces and only one is modified, so the chain length becomes, $1 + \mathcal{K} \times (1 + 2 + 2 \times 0) = 1 + 5\mathcal{K}$.

7. In the last example, there are three subspaces and two of them are modified, the chain length is equal to, $1 + \mathcal{K} \times (1 + 1 + 2 \times 2) = 1 + 6\mathcal{K}$.

Now we look into the theoretical (expected) improvement on the performance of modification operations and compare it to the practical (measured) improvement. For this, we use the examples described above. We first verify that simply estimating the cost based on the length of the chain is not enough by using Equation (4.3). Then we show how α can be estimated so that we can use Equation (4.5). Finally we verify that the latter equation correctly estimates the throughput.

In Figure 4.4 we show the average throughput of three trials for each example presented above, and for $\mathcal{K}$ varying from 1 to 3. The dashed bars indicate the measured throughput. Then we also show in the full bars the throughput predicted accordingly to Equation (4.4) using $T_{max} = 98000$[2].

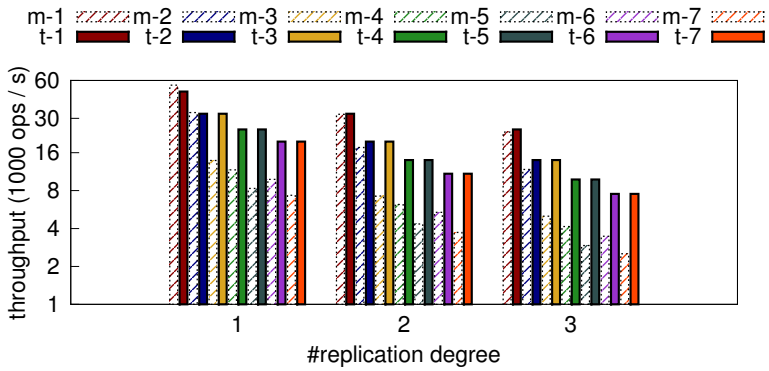


Figure 4.4: Comparison of estimation vs measured configurations using Equation (4.4).

There we can see that examples 1 and 2 have a close estimation to the measured throughput, with an average error of 3%. However, the other examples are estimated with an average error of 41%. The difference is that the latter examples 3-7 modify attributes that belong to some subspace(s), which is what motivated us to enhance Equation (4.4) to Equation (4.5) in the theoretical model. In fact, we can quickly verify that there is a cost not directly measured by the length of the chain. On one hand examples 5 and 6 were shown to have the same length, but example 6 is measured to have less throughput. Also,

[2]Estimated from three executions that only modify one element without reading it and in a configuration without subspaces.

example 4 is measured to have the same throughput as example 5, but they have the different chain lengths.

To further motivate the usage of the correcting factor for the subspaces modified in Equation (4.5), we conducted an additional experiment. Recalling the previous description of HyperDex by using Figure 3.4b, a subspace that is not modified merely needs to update the data, 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.

Using a single server and client, we used a synthetic benchmark to experimentally assess the relative costs of OVERWRITE, DELETE, and WRITE. We conducted modifications that sometimes modified a subspace and others did not. We then measured the latency server-side of applying the respective changes in the database. The result is that DELETE and WRITE have the same cost on average, which is approximately 50% more expensive than that of OVERWRITE. This is the cost that we seek to take into account with the correction factor α, which is proportional to the number of subspaces that are modified, i.e., $|\mathcal{M}|$.

To estimate alpha we require some experiments. For purpose of demonstration, we can rely on the examples presented in the previous section. By manipulation Equation (4.5), we obtain:

$$\alpha = \frac{T_{max} - T_{real} - |\mathcal{N}|\mathcal{K}T_{real}}{2|\mathcal{M}|\mathcal{K}T_{real}} \tag{4.8}$$

Then we use the experiments for each example as input for the measured throughput (variable T_{real}) and the same estimated maximum throughput of 98000 (variable T_{max}). We show the estimation for α in Table 4.6, where we obtain an average α of 2.66 and a standard deviation of 0.18.

Example	3	4	5	6	7
$K = 1$	2.527	2.693	2.440	2.989	2.586
$K = 2$	2.632	2.695	2.440	2.802	2.662
$K = 3$	2.591	2.772	2.452	3.036	2.657

Table 4.6: Estimation of α according to Equation (4.8) for examples 3-7.

We finally use Equation (4.5), with the calculated α, and show the upgraded estimations in Figure 4.5. As a result, we obtain an absolute average 4.9% error of estimation with regard to the measured values.

4.5.3 Discussion

Our assessment indicates that our model accurately predicts the performance of the HyperDex.

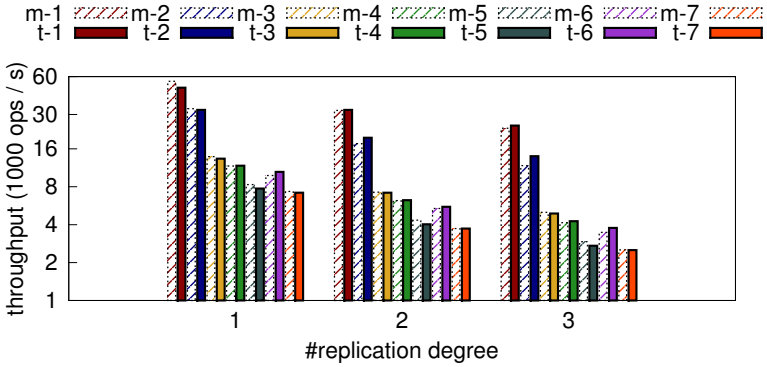


Figure 4.5: Comparison of estimation vs measured configurations using Equation (4.5).

The correction factor for the search operation, β, was calculated an average value of 3.26×10^{-7} with standard deviation of approximately 10%. The deviation is due to fact that regions of subspaces are unevenly partitioned for some scenarios. The experiments indicate that our model for search operations is able to predict the throughput with an average error of 5.28%.

The experiments have shown that the performance of the update operation does not depend only on the size of the chain. This is due to the cost of relocating an object from one region to another (delete and create) is substantially higher than the cost of updating a field without causing a relocation. We capture this cost with parameter α. The average α is calculated to be 2.66 with standard deviation of 0.18%. The experimental evaluation indicates that our model for modify operations is able to predict the performance of these operations with average error of 4.9%

Summary

In this section we proposed the modelling of HyperDex performance. First we analysed and formulated the model for the search and modify operations. Then we have experimentally derived the correction factors β and α for search and modify operations respectively. Finally, we evaluated the accuracy of our model by comparing measured real throughput with the predicted throughput. In the following chapter we discuss that the model can be used to perform the autonomic configuration of HyperDex.

5 Auto-Configuration of HyperDex

In this chapter we use the model described previously to estimate the best possible configuration for a given workload. This makes possible to automatically configure HyperDex, without burdening the programmer or the system administrator with the task of manually tuning the configuration. For this, we first present the architecture of our auto-configurator for HyperDex, and subsequently experimentally assess its performance using a concrete case study.

5.1 Architecture

In this section we describe the architecture of proposed automatic configuration model, dubbed **HyperDexConfigurator**, which is illustrated in Figure 5.1. The system consists of three main components: (a) the *query analyzer* which generates query list from workload description, (b) the *optimizer* calculates the best configuration and (c) the *configuration deployment* which deploys the most efficient configuration according to space specifications of HyperDex. We will describe the details of the components next.

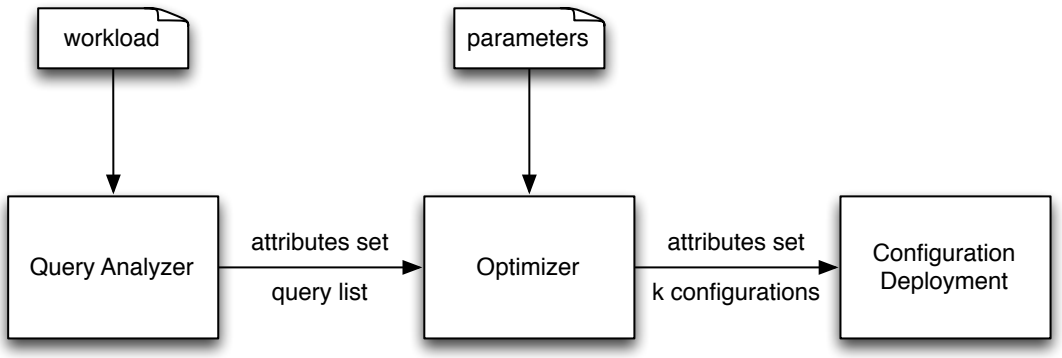


Figure 5.1: HyperDexConfigurator architecture

The *query analyzer* reads a file with all workload specifications. Then it parses the workload in order to analyze the file and creates the queries that access one or more attributes. Furthermore, it calculates the probabilities of each query and produces the unique set of attributes. Afterwards it feeds the query set, with the associated probabilities, and set of attributes to the *optimizer*.

The main goal of the *optimizer* is to derive most efficient k configurations. To achieve this it first reads the system constraints: replication degree $\mathcal{K}$, the number of objects $\mathcal{O}$, the number of regions $\mathcal{R}$ each subspace to be partitioned, correction factors α and β, and the target number k of best configurations.

The execution of the HyperDexConfigurator consists of three phases. The first phase of the algorithm is to generate all possible combinations using unique set of attributes which was provided from *query analyzer*. Then it proceeds to the second phase where throughput estimation is calculated for each of the configuration generated using the modeling we provided in chapter 4. Finally, in the last phase the HyperDexConfigurator clusters all the equally performant configurations and orders them in decreasing order of performance. Then the model performs sampling of configurations where it selects k most efficient ones and k from the rest of configurations in random fashion. In the last phase HyperDexConfigurator uses the most efficient configuration and deploys it into HyperDex in accordance with the space creation rules defined by the system.

5.2 Evaluation

We now evaluate the performance of the HyperDexConfigurator system described in the previous section. For this we use a case study inspired by a web-application that provides informations about hotels, and that maintains a data-set with information such as prices, features, reviews, etc, that is updated regularly.

Using this data-set, we assess the quality of the configurations derived by HyperDexConfigurator in face of two different workloads, as described next. 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 postcode for an hotel).

All the experiments were carried out on the same cluster that has already been described in Section 4.5.1.

5.2.1 Parameter estimation

We start by briefly showing the impact of the possible error in estimating the parameters required by our model. For this, we describe that with the α parameter required for the modifications; the following conclusions were similarly obtained 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 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 (4.5).

As an example, we used 24 such simple executions to derive α in our environment. The value that minimized the error in those tests was 2.3. Figure 5.2 shows the absolute error in predicting the throughput for the workloads presented above: in fact, our estimation is not the best one, but the impact is only of 6.5% in the error. As we shall see in the following sections, this gross (and easy) estimations have a reduced impact in the final goal of our work.

5.2.2 Throughput Estimation

In order to test the efficiency of our system in selecting the best possible configuration out of all existing ones, we tested workloads A and B on our system against 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 5.3, 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 system 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%.

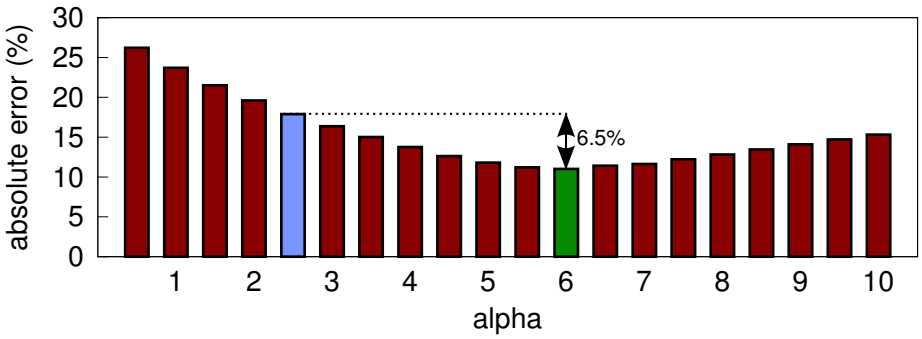


Figure 5.2: alpha error

However, more importantly than predicting the real throughputs, our model is useful as long as it accurately predicts the most performant configuration. This means that it should correctly rank the different configurations according to their throughput. Figure 5.4 presents Kendall's τ coefficient (Kendall 1938) for the rankings predicted 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 while 0 indicates complete discordance. Therefore the ranking accuracy is better when τ is close to 1. The results presented in Figure 5.4 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 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 important these ranking errors are. 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.

In fact, we observe 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 affect significantly the final estimation. Furthermore, we observe that for 5 out of 6 workloads, our system

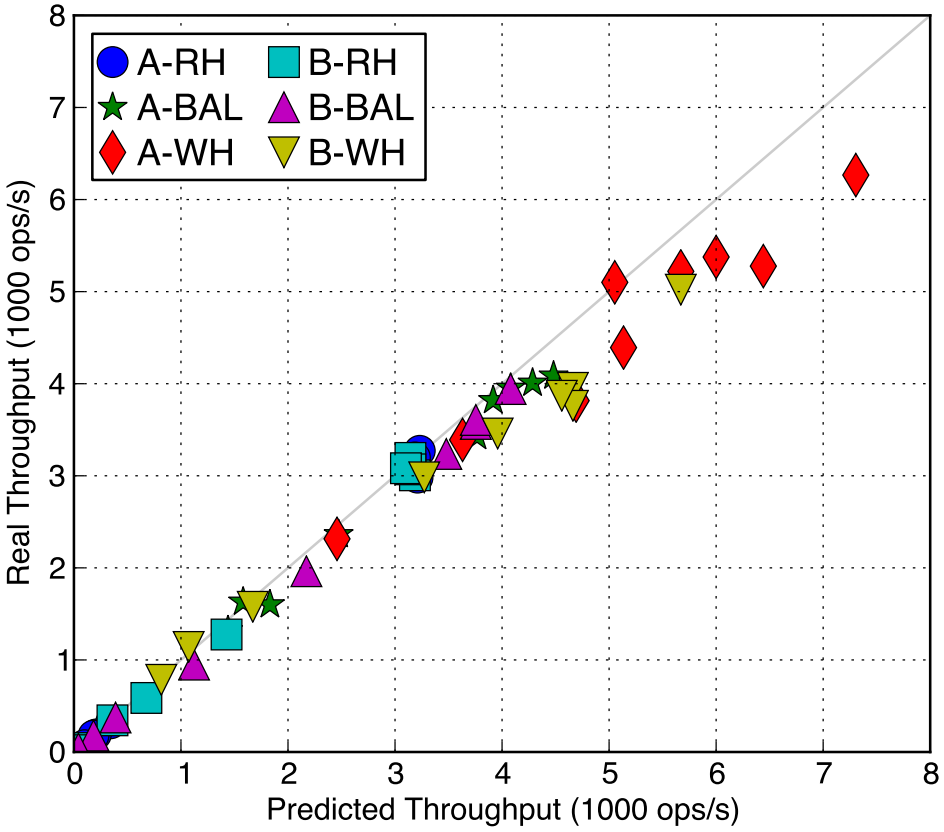


Figure 5.3: Comparison of estimation vs measured throughput of mixed workloads.

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.

5.2.3 Comparative Analysis

Finally, we compare the performance of the configuration that is derived by HyperDexConfigurator with the configurations that can be derived when using heuristics to select the best configuration. We consider the four following heuristics as baselines for comparison:

- *No-subspace* that corresponds to configure HyperDex as a regular key-value store;
- *Hyperspace* that consist in 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 is used, containing the most common attribute in the queries.

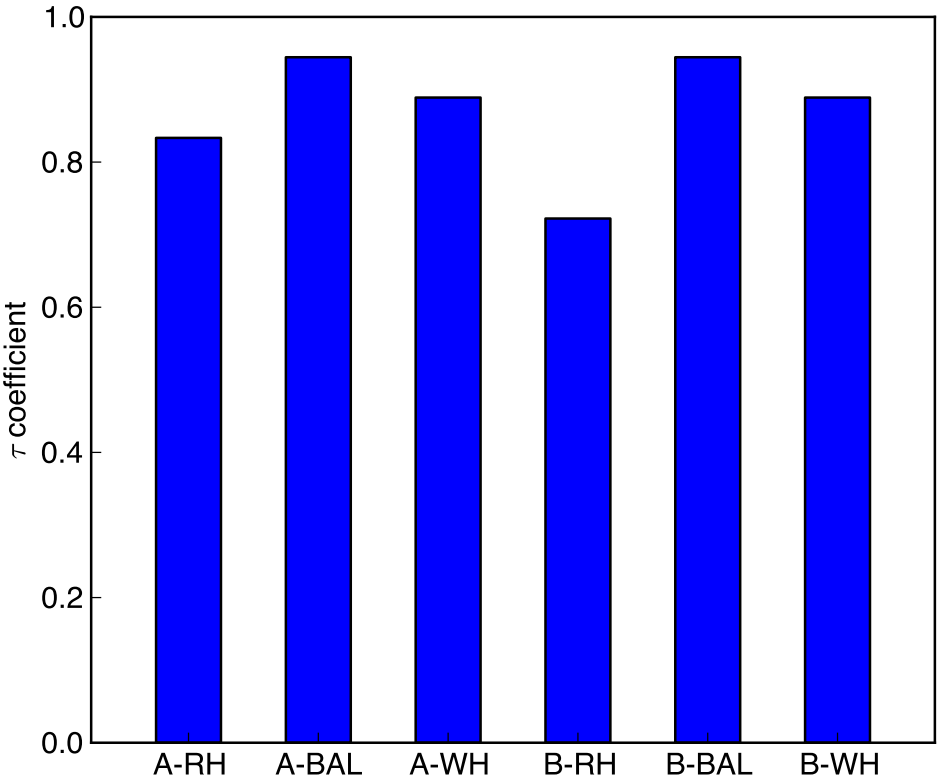


Figure 5.4: τ coefficient between estimated and real ranking of configurations

In Figure 5.6, 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 model consistently captures 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.

However, there is another heuristic, **Greedy Heuristic** which achieves relatively equal performance as our model. Greedy heuristic greedily chooses the most occurring attribute in the queries and creates a subspace with that attribute. Then eliminates all the queries containing the chosen attribute and continues to the select the next most occurring attribute untill no query is left.

We evaluated our model with another workload, workload C, where we have the same four classes of attributes but it has slightly increased percentage of the writes.

In Figure 5.7, we present the performance results obtained by HyperDexConfigurator and heuristics including greedy heuristic. Our model achieves up to 38% improvement over the greedy heuristic and

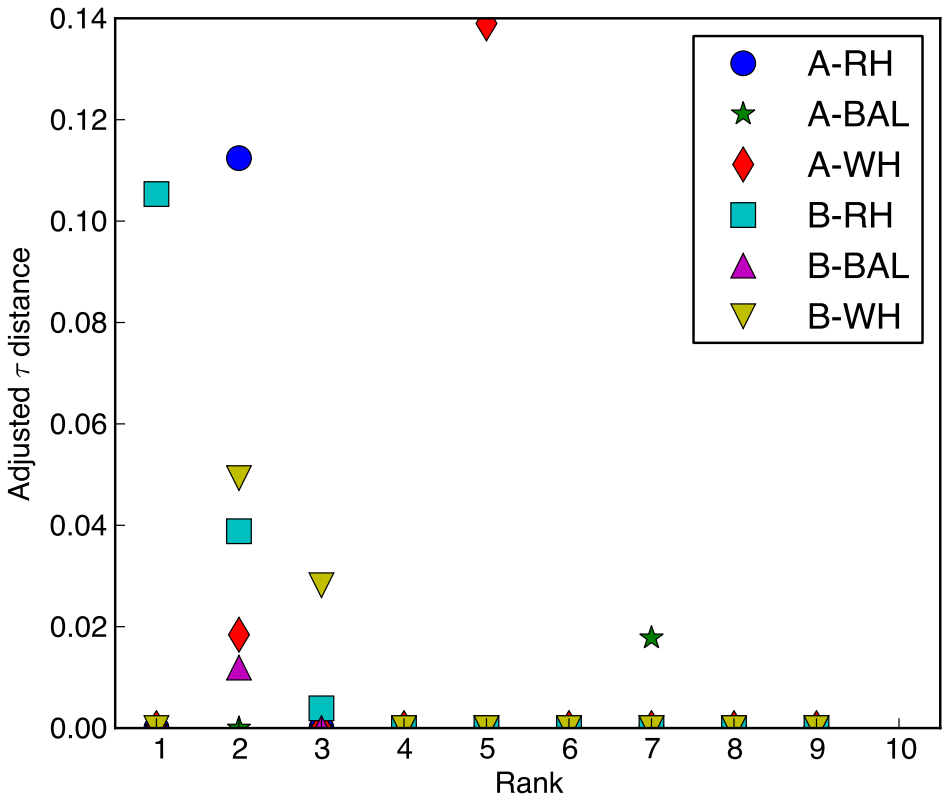


Figure 5.5: τ distance multiplied by the percentage of difference between estimated and real ranking of configurations

27% over the hyperspace heuristic.

Optimizing for every possible case is non obvious, for which reason the automatization provided by our model 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.

Summary

In this chapter we described the architecture of our system, the HyperDexConfigurator, that is able to select the best configuration for HyperDex given a specified workload. We have evaluated our system using real world dataset and two different workloads. Results indicate that our model accurately estimates the real throughput with an average error of 9%. The results also show that our model correctly orders the different configurations according to their throughput and indicate that there is high correlation between throughput rankings estimated by our model and real rankings. Finally, we presented the comparison of the best performance estimated by our system with that of baselines which use different heuristics to

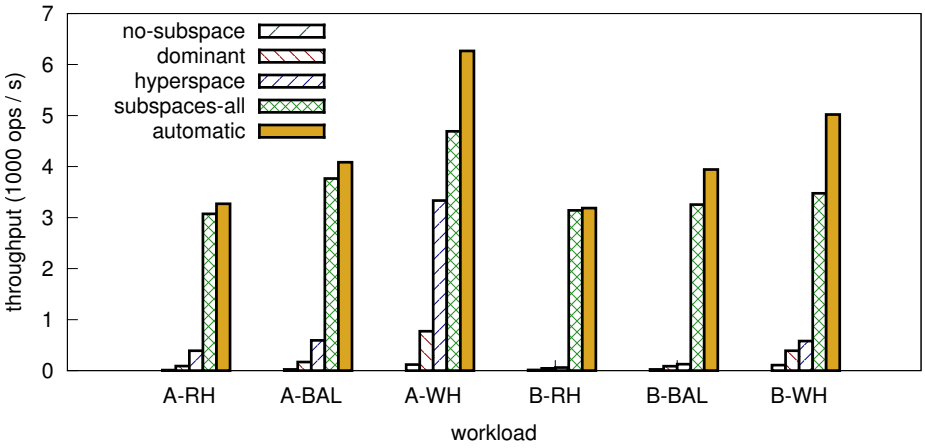


Figure 5.6: Throughput of the configuration selected by HyperDexConfigurator and the configurations selected by the different heuristics

obtain best configuration. The next chapter concludes the thesis and introduces some directions in terms of future work.

Figure 5.7: Throughput of the configuration selected by HyperDexConfigurator and the configurations selected by greedy heuristic

6 Conclusions and Future Work

This thesis proposed a HyperDexConfigurator system that is able to derive an efficient configuration for the HyperDex, given a specific workload. To achieve this goal, we first described the inner workings of the HyperDex. Then we effectively modeled the operations of it by formulating performance costs associated with search and modification operations. Experimental evaluation indicated that our model accurately predicts the throughput of the system for a given workload with less than 10% of error. In addition, our model generates the precise orderings of the configurations. Finally we have shown that the configuration derived by HyperDexConfigurator performs better than configurations derived using different plausible heuristics.

As a future work, we plan to improve the efficiency our system by employing approximation techniques in order to improve the selection procedure. Moreover, it is also important to increase the accuracy of the prediction model so that we can capture more precisely the effect of concurrent operations inside the servers; this can be achieved by employing more complex techniques such as queuing theory. Finally, we plan to to extend the model to dynamically adapt HyperDex configuration, according to runtime changes in the workload.

References

Aguilera, M. K., W. Golab, & M. A. Shah (2008). A practical scalable distributed b-tree. *Proceedings of the VLDB Endowment 1*(1), 598–609.

Bellman, R. (2003). *Dynamic Programming*. Dover Books on Computer Science Series. Dover Publications, Incorporated.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*(9), 509–517.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*(7), 422–426.

Chang, F., J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, & R. E. Gruber (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*(2), 4.

Crainiceanu, A., P. Linga, J. Gehrke, & J. Shanmugasundaram (2004). 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*, pp. 25–30. ACM.

Curino, C., E. Jones, Y. Zhang, & S. Madden (2010). Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment 3*(1-2), 48–57.

DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, & W. Vogels (2007). Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, Volume 41, pp. 205–220. ACM.

Eastlake, D. & P. Jones (2001). Us secure hash algorithm 1 (sha1).

Escriva, R., B. Wong, & E. G. Sirer (2012). Hyperdex: a distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review 42*(4), 25–36.

Ganesan, P., B. Yang, & H. Garcia-Molina (2004). One torus to rule them all: multi-dimensional queries in p2p systems. In *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004*, pp. 19–24. ACM.

Garey, M. R. & D. S. Johnson (1979). *Computers and intractability*, Volume 174. freeman New York.

Hochbaum, D. S. (1996). *Approximation algorithms for NP-hard problems*. PWS Publishing Co.

J. Paiva, P. Ruivo, P. R. & L. Rodrigues (2013, June). Autoplacer: scalable self-tuning data placement in distributed key-value stores. In *In Proceedings of the 10th International Conference on Autonomic Computing (ICAC '13)*, San Jose, CA, USA.

Karger, D., E. Lehman, T. Leighton, R. Panigrahy, M. Levine, & D. Lewin (1997). Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pp. 654–663. ACM.

Karger, D. R. & M. Ruhl (2004). Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pp. 36–43. ACM.

Karypis, G. & V. Kumar (1995). Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0.

Karypis, G. & V. Kumar (1998). Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing 48*(1), 96–129.

Kellerer, H., U. Pferschy, & D. Pisinger (2004). *Knapsack problems*. Springer Verlag.

Kendall, M. G. (1938). A new measure of rank correlation. *Biometrika 30*(1/2), pp. 81–93.

Kernighan, B. & S. Lin (1970). An eficient heuristic procedure for partitioning graphs. *Bell system technical journal*.

Khandekar, R., S. Rao, & U. Vazirani (2009). Graph partitioning using single commodity flows. *Journal of the ACM (JACM) 56*(4), 19.

Lakshman, A. & P. Malik (2010). Cassandra: a decentralized structured storage system. *Operating systems review 44*(2), 35.

Laoutaris, N., O. Telelis, V. Zissimopoulos, & I. Stavrakakis (2006). Distributed selfish replication. *Parallel and Distributed Systems, IEEE Transactions on 17*(12), 1401–1413.

Lewin, D. M. (1998). *Consistent hashing and random trees: Algorithms for caching in distributed networks*. Ph. D. thesis, Massachusetts Institute of Technology.

Metwally, A., D. Agrawal, & A. El Abbadi (2005). Efficient computation of frequent and top-k elements in data streams. *Database Theory-ICDT 2005*, 398–412.

Narayanan, D., A. Donnelly, E. Thereska, S. Elnikety, & A. Rowstron (2008). Everest: Scaling down peak loads through i/o off-loading. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pp. 15–28. USENIX Association.

Nash, J. (1951). Non-cooperative games. *The Annals of Mathematics 54*(2), 286–295.

Nightingale, E., J. Elson, O. Hofmann, Y. Suzue, J. Fan, & J. Howell (2012). Flat datacenter storage. In *Proceedings of the 10th USENIX conference on Operating systems design and implementation*.

Osborne, M. J. & A. Rubinstein (1994). *Course in game theory*. The MIT press.

Papadimitriou, C. H. & K. Steiglitz (1998). *Combinatorial optimization: algorithms and complexity*. Courier Dover Publications.

Ratnasamy, S., P. Francis, M. Handley, R. Karp, & S. Shenker (2001). *A scalable content-addressable network*, Volume 31. ACM.

Rowstron, A. & P. Druschel (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pp. 329–350. Springer.

Sagan, H. (1994). *Space-filling curves*, Volume 2. Springer-Verlag New York.

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

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

Stoica, I., R. Morris, D. Karger, M. F. Kaashoek, & H. Balakrishnan (2001). Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review 31*(4), 149–160.

van Renesse, R. & F. B. Schneider (2004). Chain replication for supporting high throughput and availability. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, Volume 6, pp. 7–7.

Vilaca, R., R. Oliveira, & J. Pereira (2011). A correlation-aware data placement strategy for key-value stores. In *Distributed Applications and Interoperable Systems*, pp. 214–227. Springer.

You, G., S. Hwang, & N. Jain (2011). Scalable load balancing in cluster storage systems. *Middleware 2011*, 101–122.

Zaman, S. & D. Grosu (2011). A distributed algorithm for the replica placement problem. *Parallel and Distributed Systems, IEEE Transactions on 22*(9), 1455–1468.