A DEMAND-AWARE HEURISTIC FOR VALUE-SPACE PARTITIONING AND REPARTITIONING

Wladimir Livolis de Alcantara Cabral

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: José Ferreira de Rezende
Antônio Augusto de Aragão Rocha

Rio de Janeiro
Maio de 2019

A DEMAND-AWARE HEURISTIC FOR VALUE-SPACE PARTITIONING AND REPARTITIONING

Wladimir Livolis de Alcantara Cabral

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. José Ferreira de Rezende, Ph.D.

Prof. Antônio Augusto de Aragão Rocha, Ph.D.

Prof. Rosa Maria Meri Leão, Ph.D.

Prof. Daniel Cardoso Moraes de Oliveira, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
MAIO DE 2019

Cabral, Wladimir Livolis de Alcantara

A Demand-Aware Heuristic for Value-Space Partitioning and Repartitioning/Wladimir Livolis de Alcantara Cabral. – Rio de Janeiro: UFRJ/COPPE, 2019.

XII, 53 p.: il.; 29, 7cm.

Orientadores: José Ferreira de Rezende

Antônio Augusto de Aragão Rocha

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2019.

Bibliography: p. 51 – 52.

1. Key-Value Store. 2. Value-Space Partitioning. 3. Horizontal Scalability. I. Rezende, José Ferreira de *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Dedico o presente trabalho a
todos que direta ou
indiretamente contribuíram para
que eu chegasse até aqui.

Agradecimentos

Agradeço primeiramente aos meus pais e avós que sempre foram muito fundamentais em minha formação moral, escolar e acadêmica. Agradeço às amizades feitas durante minha jornada acadêmica. Agradeço ainda aos professores com quem tive contato durante essa jornada, em especial meus orientadores.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

UMA HEURÍSTICA PARA PARTICIONAMENTO E REPARTICIONAMENTO DE UM ESPAÇO DE VALORES

Wladimir Livolis de Alcantara Cabral

Maio/2019

Orientadores: José Ferreira de Rezende
Antônio Augusto de Aragão Rocha

Programa: Engenharia de Sistemas e Computação

Neste trabalho, apresentamos uma nova heurística para particionamento de um banco de dados NoSQL de chave-valor baseado em seu espaço de valores. Nossa heurística leva em consideração a distribuição de operações de busca e atualização no intuito de particionar o espaço de valores em regiões mutuamente exclusivas e exaustivas que são contatadas de maneira justa por tais operações. A distribuição de cada uma dessas operações pode mudar com o tempo e, por isso, fazemos uso de uma versão do algoritmo de *Greenwald-Khanna* (um algoritmo de *data stream* bem conhecido), baseado em janelas deslizantes, no intuito de sempre ter disponível um resumo para encontrar quantis (que são os pontos onde o espaço de valores é particionado) e, então, realizar reparticionamentos de modo que as regiões ainda sejam contatadas de maneira justa. Nós realizamos experimentos variando a fração de buscas e atualizações, bem como suas distribuições, com o objetivo de avaliar o desempenho de nossa heurística e também compará-la com outras soluções. Os resultados mostram que, conforme a fração de buscas e atualizações muda, bem como suas distribuições, as regiões ainda são contatadas de maneira justa, além de não impor um número demasiado de mensagens a serem enviadas para as máquinas associadas a essas regiões.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

A DEMAND-AWARE HEURISTIC FOR VALUE-SPACE PARTITIONING AND REPARTITIONING

Wladimir Livolis de Alcantara Cabral

May/2019

Advisors: José Ferreira de Rezende
Antônio Augusto de Aragão Rocha

Department: Systems Engineering and Computer Science

In this work, we present a novel heuristic for partitioning a NoSQL key-value store based on its value-space. Our demand-aware heuristic takes into account the *updates* and *search queries*' distribution in order to partition the value-space into mutually exclusive and exhaustive regions that are fairly contacted by these operations (updates and searches). The operations' distributions might change with time and thus we make use of a sliding window based variation of the Greenwald-Khanna algorithm - a well-known data stream algorithm - in order to always have a summary available for finding quantile points (the value-space is partitioned at these points) and then to perform repartitioning so that regions are still fairly contacted. We also executed experiments varying the fraction of *searches* and *updates*, as well as their distributions, in order to evaluate the performance of our heuristic and compare it with other solutions. The results show that, as the fraction of searches and updates varies, as well as their distributions, regions are still contacted fairly and do not impose a higher number of messages to be sent to the machines associated to these regions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last years there has been an increasingly adoption of NoSQL solutions mainly because of its flexibility when compared to traditional RDBMSes.[1] This is because RDBMSes require that a database administrator design the whole database schema beforehand, which means one must define the structure of the data before working with it. Even a small change in a single table could mean having to change many other parts of the system. This operation could cost lots of money to a company. On the other hand, NoSQL databases have dynamic schema for unstructured data, which means one can insert data without a predefined schema. One can even add fields to data anytime, and different data can have different structure. Moreover, RDBMSes are usually only vertically scalable, which means in order to increase the capacity as the amount of data and users grow, the single server's hardware must be upgraded, as Figure 1.1 shows. This approach can become prohibitively expensive. NoSQL databases, on the other hand, are horizontally scalable (see Figure 1.2), which means it can increase the capacity by adding more inexpensive servers and then replicate and/or partition the database across servers.

NoSQL databases can store data in many different ways. One of the most common ways of storing data by a NoSQL database is using a key-value data store. In a database like this, data is stored as a collection of key-value pairs and a key is used as the unique identifier of a record. The key can be of any type and the value can not only be of any type, but also can be another key-value pair or even a collection of key-value pairs. Each record can have its own fields. Figure 1.3 shows an example of a key-value store which contains information on android devices (like latitude and longitude) on each of its records. Each record is uniquely identified by the unique identifier of an android device. Operations of update and search can be performed over this key-value store. For instance, one may want to search for all the records whose latitude and longitude satisfy some criteria. Or one may want to update the

[1] "NoSQL Market by Type and Application - Global Opportunity Analysis and Industry Forecast." Available at: `https://www.alliedmarketresearch.com/NoSQL-market`

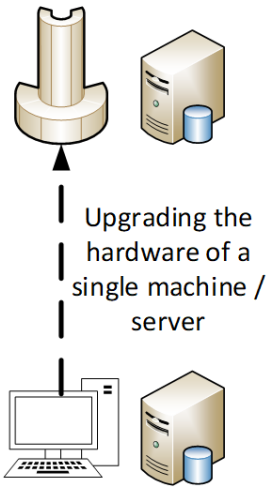


Figure 1.1: Vertical Scalability

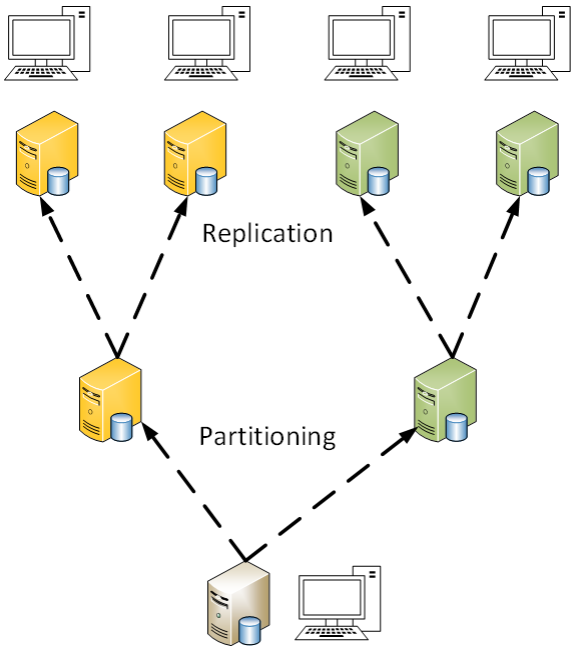


Figure 1.2: Horizontal Scalability

values of latitude and longitude of a given record.

In order to achieve horizontal scalability, different key-value databases implement different ways of partitioning. A naive approach would be replicating the whole database at all servers (replicate-at-all). Other approach could be using consistent hashing to map keys to servers (query-all). The problem with these two approaches is that they cannot handle update and search operations in a balanced way. If there are only search operations, replicating content balances server load and thus, replicate-at-all approach seems to be a good idea. On the other hand, if there are only update operations, using the second approach seems to be a better option as using consistent hashing would distribute fairly the keys among servers. However, the first approach handles poorly updates as one single operation would have to be propagated to all servers, and the second approach handles poorly searches, as a single search would have to query all servers.

android-eff75fc277ea2f1a	{(Lat, 38.582526),(Long, 73.300640),...}
android-3fdac0d173c62f73	{(Lat, 45.859412),(Long,-95.463381),...}
android-184874f991044cb0	{(Lat, 30.012031),(Long,108.469067),...}
⋮	⋮
android-990a86f1a695d0cb	{(Lat,-17.455473),(Long,-45.607556),...}

Figure 1.3: A key-value store containing information on android devices on each of its records.

HyperDex[1] uses subspace partitioning to create many lower-dimensional hyperspaces called subspaces. Each subspace is built using a unique subset of attributes as its axes. A subspace is then divided into disjoint regions, and is assigned to a unique subset of servers. Then, each region is assigned to a server. If there is a search, it will never contact more than the number of servers assigned to a single subspace. If there is an update, it will contact up to 2 servers for each existing subspace (assuming the object to be updated has all the attributes defined). The problem with this approach is that the subspaces are built upfront and some servers may be contacted more times than others. This problem should not arise if the operations' distributions[2] are known in advance. Moreover, it requires $O(2^m)$ servers for a m dimensional hyperspace, which means HyperDex requires a unique $O(2^m)$-sized subset of servers for each existing subspace.

CNS[2] uses value-space partitioning to create $\sqrt{n}$ disjoint regions based on the values of attributes, where n is the total number of servers. Each region is assigned to a unique $\sqrt{n}$-sized subset of servers, and they are created in such a way they are all fairly contacted. For this purpose, the knowledge on the operations' distributions is used to do the partitioning. An update contacts up to $2\sqrt{n}$ machines and a search contacts up to $\sqrt{n}$ servers. The downside of this approach is that, in order to create the regions, it must check multiple partitioning configurations to keep the one that maximizes an objective function for each iteration of the algorithm. This process is highly compute-intensive. Moreover, if $\sqrt{n}$ (i.e. the number of regions) is not a power of 2, some regions are bigger than others (they cover twice the value space the others cover), which means some regions may be more contacted than others.

Both HyperDex and CNS solutions present another problem to be considered - the operations' distributions might change. Even making use of the prior knowledge on the operations' distributions to do the partitioning does not prevent the case that some regions are more contacted than others. This is because the operations' distributions may change and make the current partitioning configuration not valid (if its aim is to provide regions that are contacted fairly). Thus, we find that it is imperative to support repartitioning in order to consider the most recent operations' distributions.

We want to devise a solution for the horizontal scalability problem on a key-value store, that partitions its value-space into regions that are contacted in a balanced way. We also want to be able to repartition the value-space so that regions are still

[2]We call operations' distributions the distribution of load of each type of operation (search and update) across the servers. Each operation may impose load by contacting one or more servers as we will see in the next chapters. We will call these contacts generated by operations into servers as touches at these servers. It is important to consider the distribution each type of operation follows when contacting servers in order to organize the key-value store in such a way these servers are always fairly contacted.

contacted fairly.

In this sense, the objective of this work is to contribute in three ways. Our first contribution is the proposal of a demand-aware heuristic for value-space partitioning of a key-value store. It partitions the value-space into disjoint regions taking into account the operations' distributions so that regions are contacted fairly. The heuristic has an initial configuration for the regions, and after observing a certain amount of search and/or update operations, it divides the value-space into disjoint regions in a way they are contacted in a balanced way (and thus, the servers associated to these regions).

Our second contribution is the application of a data stream algorithm - the Greenwald Khanna algorithm - to assist in the partitioning. The use of this data stream algorithm enables a much less compute-intensive partitioning. It also enables repartitioning, as a summary is always available for finding quantiles (the points where we partition the value-space). Repartitioning is important because the operations' distributions might change.

Our third contribution is the evaluation of our heuristic. We executed experiments varying the fraction of searches and updates (and their distributions) in order to evaluate the performance of our heuristic and compare it with other approaches like replicate-at-all, query-all, HyperDex and CNS. The results show that our heuristic creates regions that are always contacted fairly, while not imposing a higher number of messages to be sent to the servers (because of update and search operations, and repartitioning messages) as the fraction of searches and updates varies, as well as their distributions.

The remainder of this work is organized as follows. The chapter 2 presents a background review and related works. The chapter 3 describes our heuristic in detail. The chapter 4 describes the experiments we performed, and presents their results and some discussions. And, the chapter 5 presents our conclusions.

Chapter 2

Background Review and Related Works

In this chapter, we review some approaches on partitioning a key-value data store such as replicate-at-all, query-all, HyperDex and CNS. We also review quantiles and the GK algorithm.

2.1 Key-Value Store Partitioning Solutions

2.1.1 HyperDex

HyperDex uses subspace partitioning to create multiple lower-dimensional hyperspaces, called subspaces, from a higher-dimensional hyperspace. The term *hyperspace* here means a D-dimensional Euclidean space with $D > 3$ where its dimensions are defined by the attributes of the objects stored in the key-value store. Typically, HyperDex partitions a D-dimensional hyperspace into multiple m-dimensional subspaces where $D >> m$. Each of these subspaces uses a unique subset of object attributes as its dimensional axes, and is assigned to a unique subset of machines (or nodes). Each attribute axis of a subspace is then partitioned, dividing the subspace into disjoint regions (or blocks), and a machine is assigned to each region.

Next we give an example of how this partitioning is done. The Figure 2.1 shows one of the four 3-dimensional subspaces generated from a 12-attribute hyperspace. Each subspace is mapped to a unique subset of 4 nodes (from a datacenter consisting of 16 nodes). Each dimension (attribute axis) is divided into 2 partitions, generating 2^3 blocks per subspace. Each block is assigned to one of the nodes associated to the subspace that contains the block (in this example, 2 blocks are assigned to each node).

When there is a search, it picks the subspace with the subset of attributes that best matches the search attributes in order to contact the fewest machines. If it is a

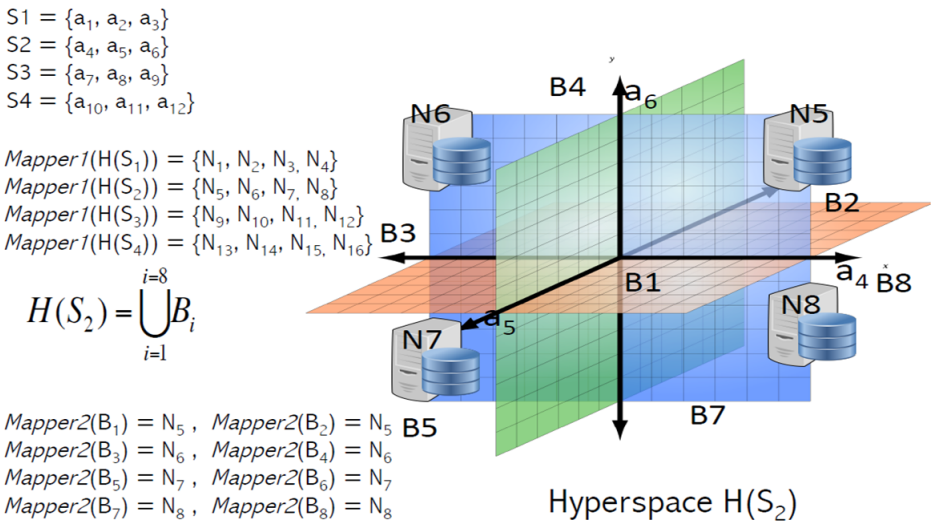


Figure 2.1: Example of a HyperDex subspace (Source: CNS article)

D-dimensional subspace, a search query specifying Q attributes will need to contact 2^{D-Q} regions and thus, this same amount of machines (assuming each dimension is split into 2 parts).

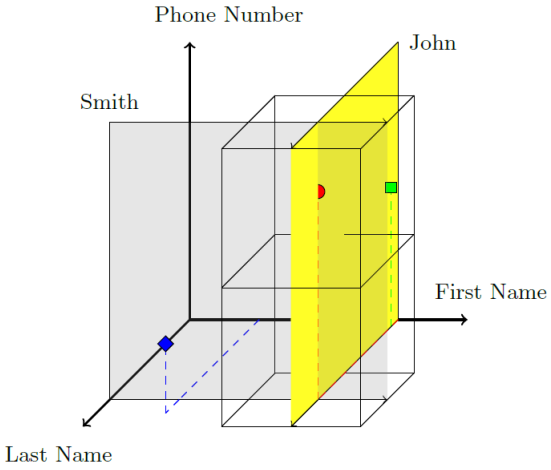


Figure 2.2: Resolving a search for two attributes (Source: HyperDex article)

The Figure 2.2 shows a search for the name "John Smith". This search is a query for $first\_name$ = "$John$" and for $last\_name$ = "$Smith$". The query for $first\_name$ is represented by the yellow hyperplane that passes through all points in which $first\_name$ = "$John$". The query for $last\_name$ is represented by the grey hyperplane that passes through all points in which $last\_name$ = "$Smith$". Their intersection form a line that corresponds to all phone numbers for people whose name is "John Smith". This line passes through two regions of this hyperspace that are represented by the two cubes. The search for the name "John Smith" needs to contact only these two regions and thus, the machines associated to these regions.

An update needs to contact up to two regions (and thus, two machines) - the one where the object was stored and the one where it will now be stored - per existing subspace, as each subspace stores a full copy of each object (the object attributes that serve as the subspace's dimensional axes and the other ones). This is because storing the full copy of objects optimizes search. If, instead, each subspace were to store only the object attributes that serve as its dimensional axes, an update would not need to contact all the subspaces anymore; however, a search would now need to contact up to all the existing subspaces. It happens that an update when contacts a subspace, also contacts up to two regions (and thus, two machines); a search, however, may contact much more than two of its regions (and thus, much more than two of its machines). It means that it might be a better approach to optimize searches.

The main criticism of this approach is that, in general, an m dimensional hyperspace requires $O(2^m)$ machines (assuming each dimension is split into 2 parts). This is too many machines even when reducing the dimension $D >> m$ of the original hyperspace by creating multiple m-dimensional subspaces, as it requires multiple unique $O(2^m)$-sized subsets of machines (each to be assigned to a single subspace). Moreover, the fact that the subspaces are built upfront makes it possible for some machines to be more contacted than others, as it does not take into account the operations' distributions. There is, however, the case of knowing the operations' distributions in advance, in which case this problem would not arise if the distributions do not change. Unfortunately, this is not always the case that these distributions do not change.

2.1.2 Replicate-at-all

Instead of splitting the key-value store and distributing its partitions across multiple servers, this solution replicates the entire key-value store and stores this replica at all servers (see Figure 2.3). Thus, a search query would only need to contact one of the servers, and one could implement a round-robin scheduling in order to distribute the search operations fairly across the machines. On the other hand, a single update would have to be propagated to all machines.

This approach is ideal for scenarios in which there are only search operations or when the amount of searches is much greater than that of updates. However, there is another major problem with this approach. If the key-value data store grows inadvertently, the only way to scale is vertically scaling (even deferring updates), as adding more servers does not help.

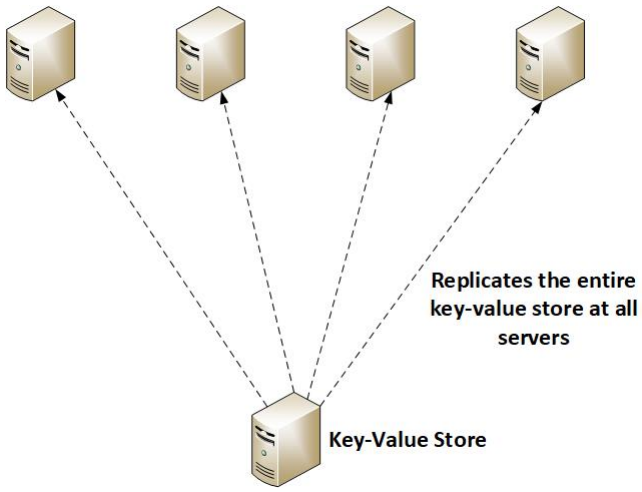


Figure 2.3: Replicating an entire key-value store across all machines

2.1.3 Query-all

The path to solve the horizontal scalability problem is to divide data across multiple servers (machines). One could try to use a classical hash function (for example, the linear congruential function $x \mapsto ax+b \ (mod\ n)$, where n is the number of machines) to map object keys to servers. This way objects would be evenly distributed across machines. The problem with this approach is that whenever n changes, almost every object has to be moved because it is now hashed to a different machine.

Consistent Hashing[3] solves this problem by representing the hash key space as a ring, called hash ring, and placing objects and machines in this ring in such a way changing n does not affect all the objects. The hash function maps each machine to a specific place on the ring. Then, it also maps object keys to places on the ring. An object can be placed in a position that was previously assigned to a machine. In such a case, this object is mapped directly to that machine. When it does not happen - when it is placed where there is no machine - it is mapped to the closest machine that is located in a greater address than that of the object. Because objects might be non-uniformly distributed across servers in a hash ring, more than one position in the ring may be assigned to a single server.

Figure 2.4 shows a hash ring with machines ($M0$ and $M1$) and objects (from $OBJ0$ to $OBJ3$) distributed across it. In this example the ring is ordered in a clockwise fashion. Thus, each object is mapped to the closest machine in a clockwise fashion. Objects $OBJ1$ and $OBJ2$ are both mapped to the machine $M0$. Similarly, objects $OBJ0$ and $OBJ3$ are both mapped to the machine $M1$. If we were to insert a machine $M2$ between $OBJ1$ and $OBJ2$, only the $OBJ2$ would have to be moved. If the second node of machine $M1$ were to be removed (the second one clockwise), only $OBJ3$ would have to be moved. Now, consider the scenario in which the second node $M1$ does not exist and the second node $M0$ also does not exist (in a clockwise

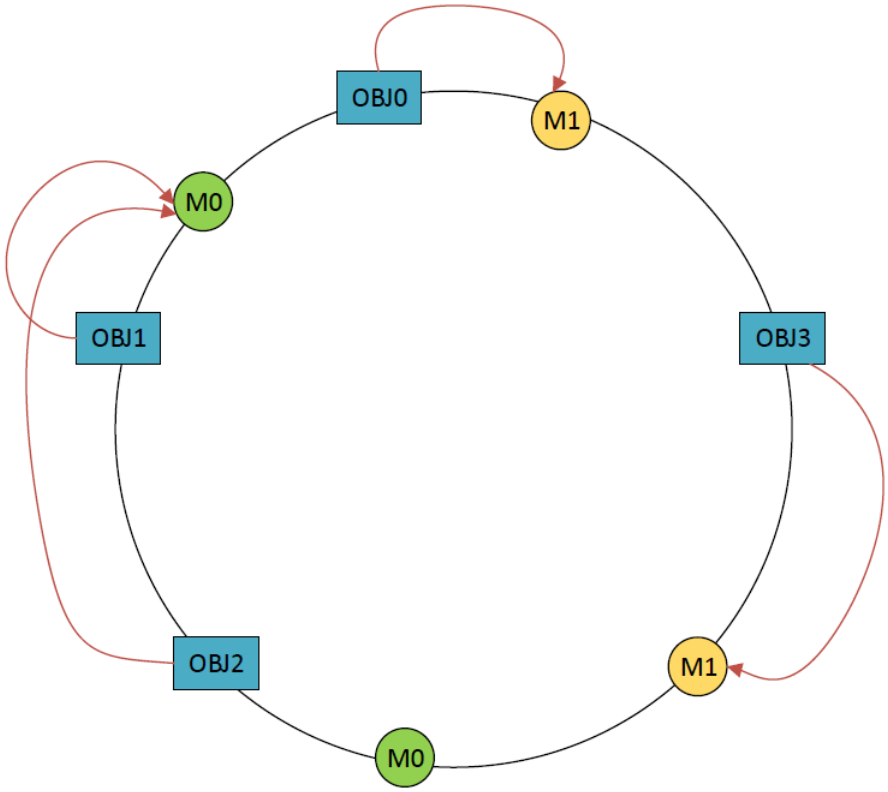


Figure 2.4: Objects and machines distributed across the hash ring

fashion). In this case, all the objects but $OBJ3$ would be mapped to machine $M1$ (which represents 75% of all objects). Assigning more than one position in the ring (i.e. adding more nodes) for each machine prevents that.

As we have seen, consistent hashing distributes evenly the objects across machines (see Figure 2.5). If we were to use it in our key-value store, we could perform updates in constant time, $O(1)$, by contacting only one machine. This is because the machine where the object is located is immediately found by hashing the object key. Since changing the values of the object attributes does not move the object from a machine to another, only one machine is contacted due to an update operation. However, as it distributes the objects based solely on its keys, there is no way of knowing where an object was placed based on its attributes. Thus, a single search query would have to go over all the machines in order to find the objects that meet its criteria - this is why we call this approach as *query-all*. This approach is ideal for scenarios in which there are only update operations or when the amount of updates is much greater than that of searches.

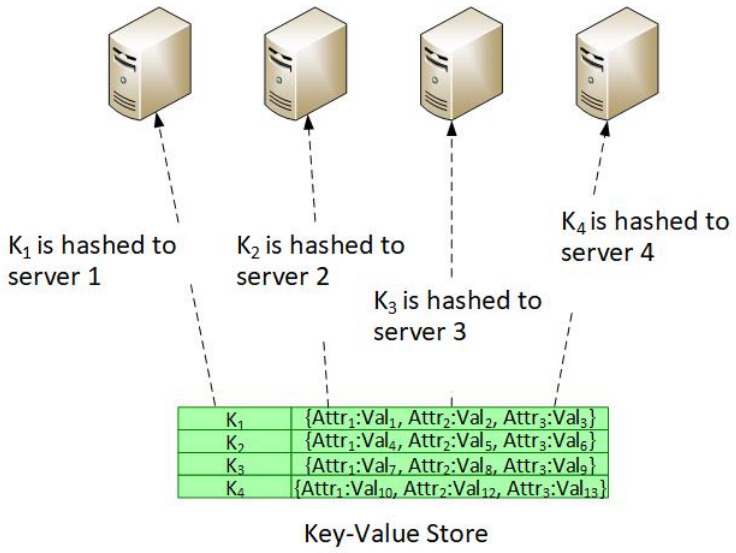


Figure 2.5: Using consistent hashing to uniquely map keys to servers

2.1.4 Global Name Service and Contextual Notification Service

It is a common sense that mobile devices are the most popular way to access the Internet today[3]. In addition, there are more mobile devices than people around the world[4]. However, the Internet continues to provide poor infrastructure support for mobility. The Internet's conflation of identity and location, i.e., the use of an IP address to both identify and localize a device on the network, implies that whenever a device changes its location, it also changes its identity. Moreover, the DNS's heavy reliance on TTL-based caching is unsuitable in high mobility scenarios as it increases update propagation delays, load on name servers, and client-perceived latency. The authors of Auspice GNS propose a Global Name Service (GNS) [4, 5] that replicates authoritative name servers in a globally geo-distributed manner based on demand and load. Wherever a name is popular, Auspice places a replica of that name-record at that location in a way it does not create a load imbalance. In comparison to DNS, this replication policy significantly reduces the reliance on passive caching, and its placement policy reduces the client-perceived latency (as opposed to DNS's static placement). Auspice GNS also supports arbitrary names, as opposed to DNS, that works with hierarchical names. The authors argue that structure of names should not be restricted as names carry application-specific semantics.

The GNS is designed as a massively geo-distributed key-value store. Each name record of this key-value store is associated with a globally unique identifier (GUID) that is the record's primary key. A name record contains an associative array of key-value pairs, wherein each key K_i is a string and the value V_i may be a string, a primitive type, or recursively a key-value pair. Next there is an example of a name

record for a GUID X.

$$\{X : \{IPs : [\{IP : 23.55.66.43, plan : Unlimited\}, \{IP : 62.44.65.75, plan : Limited\}],$$
$$geoloc : \{[lat, long], readWhitelist : [Y, Z]\}, multihome\_policy : Unlimited\}\}$$

When an endpoint Y wants to start a communication, it performs a query - a keyword-based search - seeking the GUIDs of the endpoints that meet the search criteria. This query is sent to a Contextual Notification Service (CNS)[5][2] (also a key-value store like GNS), which then returns the proper GUIDs (if they exist). Say one of the GUIDs returned by CNS is the one associated to the endpoint X. Once Y knows its GUID, Y can send a lookup request to GNS in order to retrieve the IP of X's phone, as the Figure 2.6 illustrates.

The Figure 2.7 shows how Auspice GNS handles mobility between two endpoints. Whenever Y wants to communicate with X again, it just has to directly query GNS for X's up-to-date IP, since X's GUID is already known. If X moves after Y's query but before a connection has been established, Y has to query GNS again. After a connection has been established, if either endpoint moves one at a time, the connection can be re-synchronized without relying upon the GNS. If both endpoints move simultaneously, one or both endpoints must query the GNS and then re-synchronize the connection.

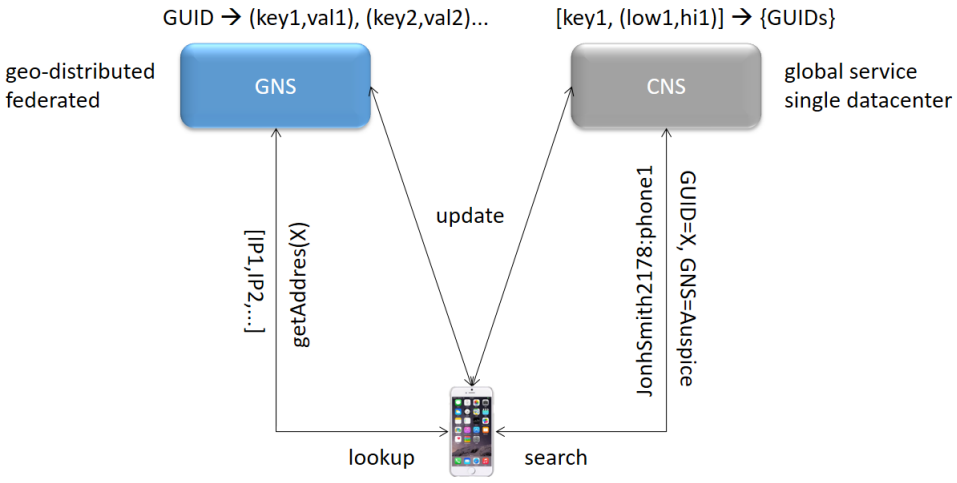


Figure 2.6: GNS and CNS working side by side

[3]"Mobile devices become most popular way to access internet", New York Post. Available at: https://nypost.com/2016/11/03/mobile-devices-become-most-popular-way-to-access-internet/

[4]"There are officially more mobile devices than people in the world", Independent. Available at: https://www.independent.co.uk/life-style/gadgets-and-tech/news/there-are-officially-more-mobile-devices-than-people-in-the-world-9780518.html

[5]CNS paper is expected to be submitted by some of the GNS's authors any time soon.

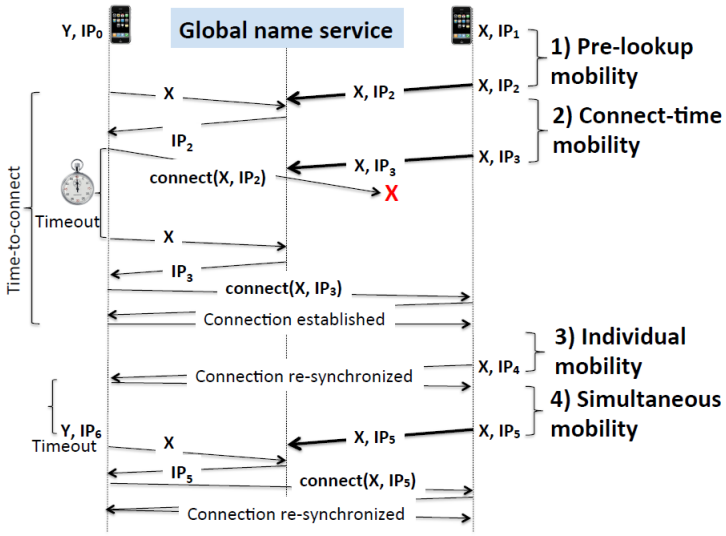


Figure 2.7: Auspice GNS handling mobility between two endpoints (Source: CNS article)

The authors argue that the proposed GNS-CNS separation guarantees privacy since, despite collusion, the CNS or the GNS can only know of the existence of attribute-value pairs but not be able to correlate them to user identities in the absence of external information or side-channels.

Figure 2.8 illustrates other scenarios in which CNS might be useful. Using CNS, a meteorological agency is able to notify people (through their mobile devices) to leave the area they are in (by searching for mobile phones that meet some criteria based on their longitude and latitude) because of a coming hurricane. It can also be used in order to issue security alerts or even epidemic alerts, for example.

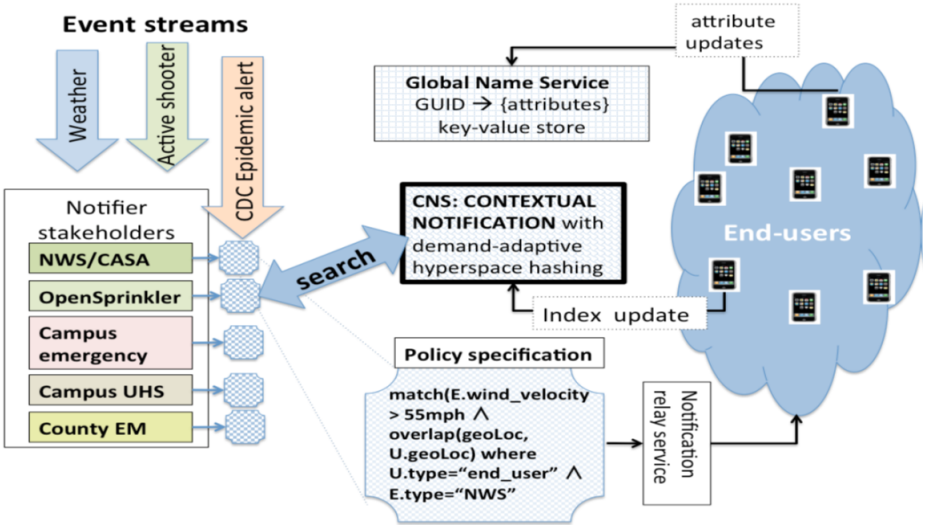


Figure 2.8: Example of application for CNS (Source: CNS article)

In order to guarantee a decent performance for searches and updates, it is desir-

able that CNS partitions its key-value datastore across several machines with some special care. The CNS authors propose a heuristic for this purpose. It partitions the value-space into $\sqrt{n}$ mutually exclusive and exhaustive (MEE) partitions, called regions, where n is the number of machines. In each iteration of this heuristic (for a total of $\sqrt{n}-1$ iterations), it picks a preexisting region, then it picks a point on one of the attribute axes and divides the region at that point creating two new regions. It then calculates the Jain's Fairness Index (JFI) value[6][6] for this partitioning configuration. The function that returns this value is actually an objective function based on JFI that takes into account update and search operations seen so far in order to create regions that are contacted fairly. It repeats the process so that in the end of each iteration it stays with the partitioning configuration that gives the best JFI value. Each region is then assigned to a unique subset of $\sqrt{n}$ machines (which means it is replicated $\sqrt{n}$ times). Thus, an update may contact up to $2\sqrt{n}$ machines - the $\sqrt{n}$ machines associated to the previous region and the $\sqrt{n}$ machines associated to the new region; a search may contact up to $\sqrt{n}$ machines, as a subset of $\sqrt{n}$ machines already covers the entire value-space.

The authors of CNS prove in their article that there is no subspace partitioning or replication scheme - even with complete knowledge of the workload - for which it is possible to achieve better than $\Omega(\sqrt{n})$ scalability, unless there is either only updates or only searches in the workload. They also prove that it is possible to achieve $\sqrt{n}$ scalability if the number of regions is $\sqrt{n}$ and each of these regions is assigned to a unique subset of $\sqrt{n}$ machines, and updates and searches are uniformly distributed across regions. This is the reason for so many $\sqrt{n}$ appearing here.

The problem with this heuristic is that in order to know the configuration that gives the best JFI value, it must check all possible partitioning configurations. This process is highly compute-intensive. Also, if the number of regions, $\sqrt{n}$, is not a power of 2, the value-space is not equally divided; some regions are bigger than others - a region may cover the double of the area the other cover. It means that some regions may be more contacted than others.

Next we give an example of how CNS does the partitioning of its key-value datastore.

Let $A = \{A1, A2, A3\}$ be the set of attributes.

Let $N = \{N1, N2, \cdots, N16\}$ be the set of machines.

Thus, $n = 16$ and the number of MEE regions, $\sqrt{n}$, is 4.

Let $R = \{[(A1, [0, 1]), (A2, [0, 1]), (A3, [0, 1])]\}$ be the set of regions.

In the first iteration, we pick the hyperplane $A1 = 0.35$ to divide the region $[(A1, [0, 1]), (A2, [0, 1]), (A3, [0, 1])]$ into two new regions. Thus, $R = \{[(A1, [0, 0.35]), (A2, [0, 1]), (A3, [0, 1])], [(A1, [0.35, 1]), (A2, [0, 1]), (A3, [0, 1])]\}$.

[6]The function that returns the JFI value is shown in the section 4.1.1.

In the second iteration, we pick the hyperplane $A2 = 0.35$ to divide the region $[(A1, [0.35, 1]), (A2, [0, 1]), (A3, [0, 1])]$ into two new regions. Thus, $R = \{[(A1, [0, 0.35]), (A2, [0, 1]), (A3, [0, 1])], [(A1, [0.35, 1]), (A2, [0, 0.35]), (A3, [0, 1])], [(A1, [0.35, 1]), (A2, [0.35, 1]), (A3, [0, 1])]\}$.

In the third and final iteration, we pick the hyperplane $A3 = 0.5$ to divide the region $[(A1, [0.35, 1]), (A2, [0.35, 1]), (A3, [0, 1])]$ into two new regions. Thus, $R = \{[(A1, [0, 0.35]), (A2, [0, 1]), (A3, [0, 1])], [(A1, [0.35, 1]), (A2, [0, 0.35]), (A3, [0, 1])], [(A1, [0.35, 1]), (A2, [0.35, 1]), (A3, [0, 0.5])], [(A1, [0.35, 1]), (A2, [0.35, 1]), (A3, [0.5, 1])]\}$.

Figure 2.9 illustrates this example.

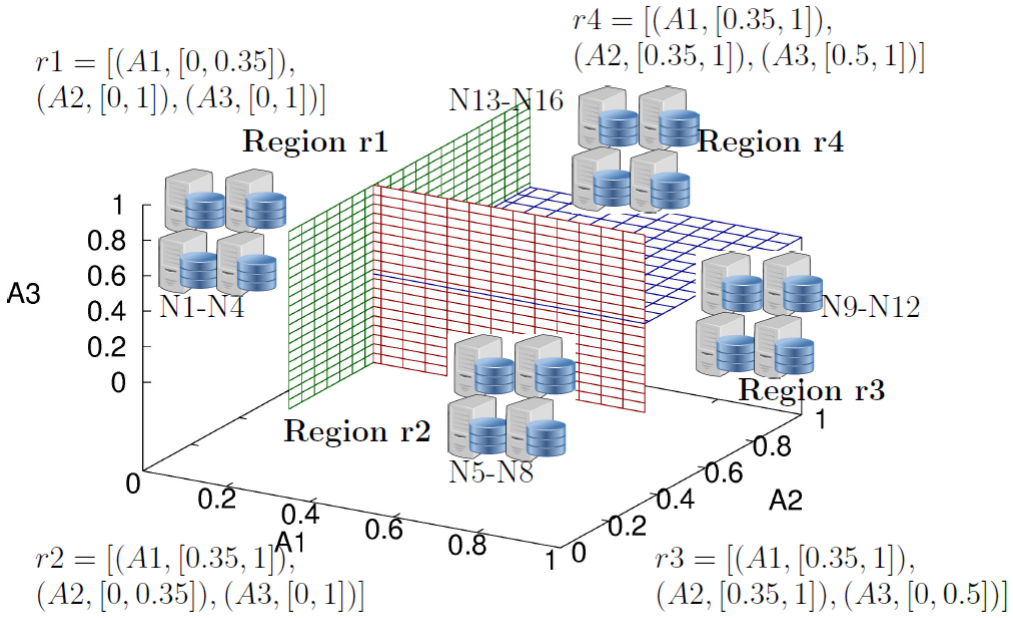


Figure 2.9: Example of partitioning using CNS (Source: CNS article)

Next, we give another example of how the partitioning can generate unbalanced regions. For this purpose, consider a scenario in which $n = 9$. Thus, the final number of regions is 3, which is **not a power of 2**. Initially, we have $R = \{[(A1, [0, 1]), (A2, [0, 1]), (A3, [0, 1])]\}$. Let us pick the hyperplane $A1 = 0.5$ to divide this initial region into two new regions. Now, $R = \{[(A1, [0, 0.5]), (A2, [0, 1]), (A3, [0, 1])], [(A1, [0.5, 1]), (A2, [0, 1]), (A3, [0, 1])]\}$. Finally, we pick the hyperplane $A2 = 0.5$ to divide the region $[(A1, [0.5, 1]), (A2, [0, 1]), (A3, [0, 1])]$ into two new regions. In the end we have the following regions: $R = \{[(A1, [0, 0.5]), (A2, [0, 1]), (A3, [0, 1])], [(A1, [0.5, 1]), (A2, [0, 0.5]), (A3, [0, 1])], [(A1, [0.5, 1]), (A2, [0.5, 1]), (A3, [0, 1])]\}$. One can note that the first region (the green one) has the double of the size of each one of the other two regions (see Figure 2.10). And, there is no other way to partition the value-space into 3 regions,

by splitting one region into two new ones at every iteration, that would not create this unbalance. This is because the way the partitioning is done, it is required that the number of regions must be always a power of 2.

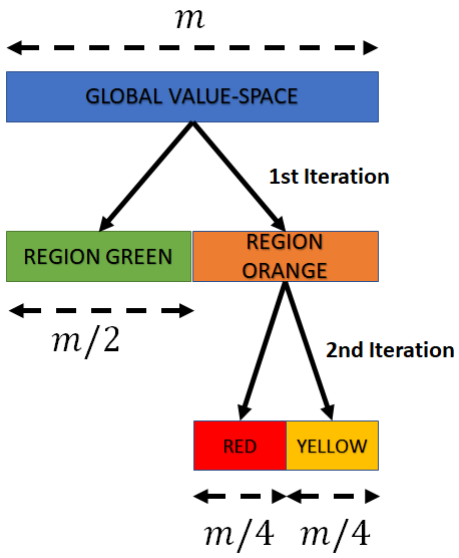


Figure 2.10: Dividing the value space into $\sqrt{n}$ regions when $n = 9$.

There is another major problem with this heuristic. Even taking into account the operations' distributions to create regions, it does not consider the case that these distributions might change. In fact, CNS does generate regions that are contacted fairly (assuming $\sqrt{n}$ is a power of 2) until the operations' distributions change, when the configuration of partitioning it created is not valid anymore. We believe that the solution to this problem is to support repartitioning, which consists of performing a new partitioning from time to time based on new observed operations. The problem is that CNS partitioning process is highly compute-intensive as we stated earlier, which makes repartitioning impracticable.

2.2 Quantiles over a data stream

Quantiles are points that divide an ordered dataset into subsets of (nearly) equal sizes. The ϕ-quantile, with $0 < \phi \leq 1$, of a dataset of size n is the value whose rank is $\phi \cdot n$. Median and quartiles are examples of quantiles. The median is the 0.5-quantile that divides a dataset into 2 halves. The quartiles are the 0.25-quantile, the 0.5-quantile and the 0.75-quantile that divide a dataset into 4 quarters.

As data streams can theoretically be infinite and a computer's memory is always finite, one cannot afford to know every element from a data stream to sort them and then compute the quantile points. It is desirable that an algorithm to find quantiles over data streams is able to operate by doing a single pass over the data. In addition,

some data has to be discarded to save memory and, for this, a margin of error ϵ is allowed. Therefore, the ϵ-approximate ϕ-quantile, with $0 < \phi \leq 1$ and $0 \leq \epsilon \leq 1$, of a dataset of size n is any value whose rank is inside $[n \cdot (\phi - \epsilon), n \cdot (\phi + \epsilon)]$.

2.2.1 The Greenwald-Khanna (GK) Algorithm

At each unit of time an observation represented by its value v is seen, such that at a given instant of time n, n observations were seen. The GK algorithm [7, 8] maintains a data structure $Q(n)$ – called summary – that contains an ordered sequence of tuples that correspond to a subset of the n observations seen thus far. The summary also maintains for each observation v, the lower and upper bounds – $rmin(v)$ and $rmax(v)$ respectively – on the rank of v, which correspond to the minimum and the maximum possible rank of v among the observations seen so far. Formally, the summary is an ordered collection of tuples $Q(n) = \{t_0, t_1, \cdots, t_{s-1}\}$, where each tuple $t_i = (v_i, g_i, \Delta_i)$ contains 3 values: i) v_i that is a value which represents one of the n observations seen thus far; ii) g_i that is a value equals to $rmin(v_i) - rmin(v_{i-1})$; and iii) Δ_i that is a value equals to $rmax(v_i) - rmin(v_i)$. The values v_0 and v_{s-1} always correspond, respectively, to the lowest and highest values seen so far. Some observations can be taken from the definitions above:

1. $rmin(v_i) = g_i + rmin(v_{i-1}) = \sum_{j \leq i} g_j$
2. $rmax(v_i) = rmin(v_i) + \Delta_i = \sum_{j \leq i} g_j + \Delta_i$
3. $g_i + \Delta_i - 1 = rmax(v_i) - rmin(v_{i-1}) - 1$ is an upper bound on the total number of observations that may have fallen between v_{i-1} and v_i
4. $\sum_i g_i = n$

Proposition: Given a summary $Q(n)$, a ϕ-quantile can always be identified to within an error of $max_i(g_i + \Delta_i)/2$.

Corollary: If at any time n, the summary $Q(n)$ satisfies the property that $max_i(g_i + \Delta_i) \leq 2\epsilon n$, then we can answer any ϕ-quantile query to within an ϵn precision.

The capacity of a tuple t_i at time n, denoted by $cap(t_i, n)$, is defined as $\lfloor 2\epsilon n \rfloor - \Delta_i$ and it increases over time (as n increases). It is important to state that Δ_i of a tuple t_i never changes, while g_i may increase over time because of merge operations. It is said that a tuple is full at time n if $g_i + \Delta_i = \lfloor 2\epsilon n \rfloor$, i.e., if $g_i = cap(t_i, n)$. Therefore, the capacity of a tuple t_i is the maximum number of observations that can be counted by g_i before t_i becomes full.

Lower capacity tuples will eventually be merged into larger capacity ones (or into tuples of similar capacity) in order to minimize the size of $Q(n)$. The choice

for preserving higher capacity tuples is that they correspond to values whose ranks are known with higher precision (recall that their Δ's are smaller, which means the difference $rmax - rmin$ is also smaller). Tuples with similar capacity, i.e., with approximately equal values of $log_2(capacity)$, are grouped into the same geometric classes referred to as bands. Thus, at time n, a tuple t_i is in a band α if $cap(t_i, n) \approx 2^\alpha$. As the capacities increase over time, the band of a tuple also increases over time. The band of a tuple t_i at time n is denoted by $band(t_i, n)$. If at some time m, we have $band(t_i, m) \leq band(t_j, m)$ then for all times $n > m$, we have $band(t_i, n) \leq band(t_j, n)$ (since $\Delta_i \geq \Delta_j$ for all n). Thus, if tuples are ever in the same band, they never appear in different bands as n increases. The $band(t_i, n)$ is α if

$p - 2^\alpha - (p \ mod \ 2^\alpha) < \Delta_i \leq p - 2^{\alpha-1} - (p \ mod \ 2^{\alpha-1})$, with $p = \lfloor 2\epsilon n \rfloor$

(or equivalently, $2^{\alpha-1} + (p \ mod \ 2^{\alpha-1}) \leq cap(t_i, n) < 2^\alpha + (p \ mod \ 2^\alpha)$)

Especial cases: If $\Delta_i = p$, then $band(t_i, n) = 0$ (the lowest band); else if $\Delta_i = 0$, then $band(t_i, n) = \lceil log_2(p) \rceil$ (the highest band).

An additional data structure – an ordered tree structure – is used in order to assist the merge operation among bands. Given a summary $Q(n) = \{t_0, t_1, \cdots, t_{s-1}\}$, a quantile tree $T(n)$, at time n, associated with $Q(n)$, contains a node V_i for each t_i, and a special root node R. The parent of a node V_i is the node V_j such that j is the least index greater than i with $band(t_j, n) > band(t_i, n)$. If no such index exists, then R is set to be the parent. The children of each node are ordered as they appear in $Q(n)$. All children (and all descendants) of a given node n correspond to tuples that have capacities smaller (or equivalently, have Δ values larger) than that of tuple t_i. The idea is that firstly child nodes are merged into their parent nodes and only then merge sibling nodes.

Operations supported by GK:

Quantile(ϕ): Computes an ϵ-approximate ϕ-quantile from the summary $Q(n)$ after n observations. It looks for v_i such that $rmin(v_i) \geq r - \epsilon n$ and $rmax(v_i) \leq r + \epsilon n$, where r is the rank associated with the ϕ-quantile and is equal to $\lfloor \phi \cdot (n-1) \rfloor$. Any v_i that satisfies the $rmin$ and $rmax$ conditions is eligible to be returned as an ϵ-approximate ϕ-quantile.

Insert(v): Inserts a tuple t containing v into the summary. Firstly, it sets $g = 1$, as t is a new tuple. Secondly, it sets Δ following some conditions: if v is the new minimum or the maximum observation seen (or equivalently, if t is the new t_0 or t_{s-1}), then $\Delta = 0$. Otherwise, $\Delta = \lfloor 2\epsilon n \rfloor - 1$. Once set, Δ will never change. Finally, it looks for the smallest i, such that $v_{i-1} \leq v < v_i$ and inserts t between t_{i-1} and t_i.

Compress(): Compresses the summary by merging adjacent tuples based on their bands. Figure 2.11 shows how this operation works.

```
COMPRESS()
  for i from s − 2 to 0 do
    if ((band(t_i, n) ≤ band(t_{i+1}, n)) &&
        (g*_i + g_{i+1} + Δ_{i+1} < 2εn)) then
      g_{i+1} = g_{i+1} + g*_i;
      Remove t_i and all its descendants;
    end if
  end for
end COMPRESS
```

Figure 2.11: The GK's compress operation (Source: GK article)

At every instant of time n, GK firstly checks whether it is time to compress the summary or not. It will run *Compress*() whenever $n \equiv 0 \ mod \ (1/2\epsilon)$. Then, GK runs $Insert(v)$ to insert into the summary a new tuple associated to the observation v seen at time n. Figure 2.12 show the pseudocode for GK algorithm.

```
Initial State
  Q_GK ← ∅; s = 0; n = 0.
Algorithm
To add the (n + 1)st observation, v, to summary
Q_GK(n):

  if (n ≡ 0 mod 1/(2ε)) then
    COMPRESS();
  end if
  INSERT(v);
  n = n + 1;
```

Figure 2.12: The GK algorithm (Source: GK article)

2.2.2 Sliding Window Greenwald-Khanna (GK-window) Algorithm

GK-window [8] summarizes the most recent W observations seen. For this purpose, it implements a fixed sliding window of size W. While the window does not cover W elements, it increases its size whenever a new observation is seen. After the first W arrivals, the window will always cover precisely W observations. From this point on, for each newly arrived observation, the oldest observation in the window

is deleted. The window is divided into blocks of $\epsilon W/2$ consecutive observations. Each block is summarized using the GK algorithm with an $\epsilon/2$ precision. Thus, the window will contain $2/\epsilon$ summaries (assuming the window already covers W observations). The only block summarizing data is the most recent one and is called under construction. The blocks that already contains $\epsilon W/2$ observations are called complete and are no longer modified. When a single observation in a block exits the window (i.e., when the oldest observation from the W most recent ones is discarded), this block is said to be expired. Whenever a query for a ϕ-quantile is done, the algorithm combines all the summaries from the $2/\epsilon$ non-expired blocks - the summaries from the complete blocks and from the under construction block are combined together but the expired block's summary is ignored - into a single $\epsilon/2$-precision quantile summary. Then, it computes an $\epsilon/2$-approximate ϕ-quantile from that summary using the GK algorithm.

The combine operation: $Combine(Q', Q'')$
Let $Q' = \{x_1, x_2, \cdots, x_a\}$ and $Q'' = \{y_1, y_2, \cdots, y_b\}$ be two quantile summaries. This operation [8, 9] produces a new quantile summary $Q = \{z_1, z_2, \cdots, z_{a+b}\}$ by performing the union of Q' and Q'', and setting the rank of each element as follows. Let x_r and y_r be, respectively, the smallest elements in Q' and Q'' that are not already in Q. At each iteration i, we pick the smallest element between x_r and y_r to be z_i. For now, assume that z_i corresponds to some element x_r in Q'. Let y_s be the largest element in Q'' that is not larger than x_r (y_s is undefined if no such element exists), and let y_t be the smallest element in Q'' that is not smaller than x_r (y_t is undefined if no such element exists). Then

$$rmin_Q(z_i) = \begin{cases} rmin_{Q'}(x_r) & \text{if } y_s \text{ undefined} \\ rmin_{Q'}(x_r) + rmin_{Q''}(y_s) & \text{otherwise} \end{cases}$$

$$rmax_Q(z_i) = \begin{cases} rmax_{Q'}(x_r) + rmax_{Q''}(y_s) & \text{if } y_t \text{ undefined} \\ rmax_{Q'}(x_r) + rmax_{Q''}(y_t) - 1 & \text{otherwise} \end{cases}$$

If at some iteration i, z_i corresponds to some element y_r in Q'', the process of setting the rank of z_i follows the same rules, except that there would be x_s and x_t instead of y_s and y_t.

2.2.3 A Test Case for the GK Algorithm

Lets consider the following input sequence: $\{12, 10, 11, 10, 1, 10, 11, 9\}$. We are going to show how the GK algorithm performs when observes this sequence of values. We are going work with an error $\epsilon = 0.25$. Thus, we are going to perform a new

compression every $1/2\epsilon = 2$ new observations.

Let N be the number of observations seen so far, and S our summary. Then, $N = 0$ and $S = \{\}$ as no observation was seen so far.

I) Iteration 1: Observation 12 arrives

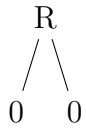
a) Before inserting the new value we check whether we need to perform a compression. In our case we do not need compression, as there is no observation in our summary.

b) Now we insert 12 into our summary S.
$g(12) = 1$ as we are inserting a new tuple.
$\Delta(12) = 0$ as 12 is the new minimum / maximum observation seen so far.
This is our summary so far: $S = \{(12, 1, 0)\}$.
Now that we have seen one new observation $N = N+1 = 1$. Then, we are able to calculate $p = \lfloor 2\epsilon N \rfloor = \lfloor 2 \cdot 0.25 \cdot 1 \rfloor = \lfloor 0.5 \rfloor = 0$. We calculate p in order to know the band of each tuple. This information is required when performing compressions. Recall that, by definition, if a tuple has $\Delta = p$ then it is in band 0. Also, if a tuple has $\Delta = 0$ then it is in band $log_2(\lceil p \rceil)$. In order to know the band of a tuple when its $\Delta \neq 0$ or $\Delta \neq p$, we use the following formula: $p - 2^{\alpha} - (p \ mod \ 2^{\alpha}) < \Delta \leq p - 2^{\alpha-1} - (p \ mod \ 2^{\alpha-1})$.
Now we know that band 0 contains the tuples with the following *deltas*: band 0: Deltas = $\{0\}$. We can also draw the tree structure T mentioned in the section 2.2.1:

II) Iteration 2: Observation 10 arrives

a) No compression to be performed as $N \mod 1/2\epsilon = 1 \mod 1/2\epsilon \neq 0$

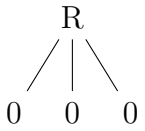
b) Insert observation 10 into summary S
$\Delta(10) = 0$ as 10 is the new minimum observation seen.
$S = \{(10, 1, 0), (12, 1, 0)\}$
$N = N + 1 = 2 \rightarrow 2\epsilon N = 1 \rightarrow p = \lfloor 2\epsilon N \rfloor = 1$
band 0: Deltas = $\{0, 1\}$
T:



III) Iteration 3: Observation 11 arrives

a) $N \mod 1/2\epsilon = 0$: must perform compression
Now we must test whether we can delete one of the tuples: $(10, 1, 0)$ $(12, 1, 0)$.
$band(10) \leq band(12)$? True, because $0 \leq 0$.
$g(10) + g(12) + \Delta(12) < 2\epsilon N$? False, because $1 + 1 + 0 < 1$.
Since the result was false to at least one of the requirements for compression, we cannot delete $(10, 1, 0)$.

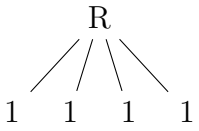
b) Insert observation 11 into summary S
$\Delta(11) = \lfloor 2\epsilon N \rfloor - 1 = \lfloor 1 \rfloor - 1 = 0$
$S = \{(10, 1, 0), (11, 1, 0), (12, 1, 0)\}$
$N = N + 1 = 3 \rightarrow 2\epsilon N = 1.5 \rightarrow p = \lfloor 2\epsilon N \rfloor = 1$
band 0: Deltas $= \{0, 1\}$
T:



IV) Iteration 4: Observation 10 arrives

a) $N \mod 1/2\epsilon \neq 0$: no compression to be performed

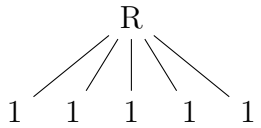
b) Insert observation 10 into summary S
$\Delta(10) = \lfloor 2\epsilon N \rfloor - 1 = \lfloor 1.5 \rfloor - 1 = 0$
$S = \{(10, 1, 0), (10, 1, 0), (11, 1, 0), (12, 1, 0)\}$
$N = N + 1 = 4 \rightarrow 2\epsilon N = 2 \rightarrow p = \lfloor 2\epsilon N \rfloor = 2$
band 0: Deltas $= \{2\}$
band 1: Deltas $= \{0, 1\}$
T:



V) Iteration 5: Observation 1 arrives

a) $N \mod 1/2\epsilon = 0$: must perform compression
Testing: $(11, 1, 0)$ $(12, 1, 0)$
band: $1 <= 1 \rightarrow True$
$1 + 1 + 0 < 2 \rightarrow False$
Cannot delete $(11, 1, 0)$
Testing: $(10, 1, 0)$ $(11, 1, 0)$
band: $1 <= 1 \rightarrow True$
$1 + 1 + 0 < 2 \rightarrow False$
Cannot delete $(10, 1, 0)$
Testing: $(10, 1, 0)$ $(10, 1, 0)$ band: $1 <= 1 \rightarrow True$
$1 + 1 + 0 < 2 \rightarrow False$
Cannot delete $(10, 1, 0)$

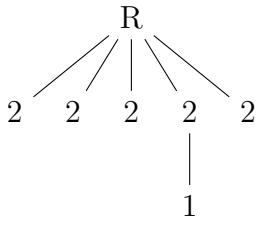
b) Insert observation 1 into summary S
$\Delta(1) = 0$ as 1 is the new minimum observation seen.
$S = \{(1, 1, 0), (10, 1, 0), (10, 1, 0), (11, 1, 0), (12, 1, 0)\}$
$N = N + 1 = 5 \rightarrow 2\epsilon N = 2.5 \rightarrow p = \lfloor 2\epsilon N \rfloor = 2$
band 0: Deltas $= \{2\}$
band 1: Deltas $= \{0, 1\}$
T:



VI) Iteration 6: Observation 10 arrives

a) $N \mod 1/2\epsilon \neq 0$: No compression to be performed

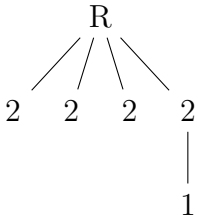
b) Insert observation 10 into summary S:
$\Delta(10) = \lfloor 2\epsilon N \rfloor - 1 = \lfloor 2.5 \rfloor - 1 = 1$
$S = \{(1, 1, 0), (10, 1, 0), (10, 1, 0), (10, 1, 1), (11, 1, 0), (12, 1, 0)\}$
$N = N + 1 = 6 \rightarrow 2\epsilon N = 3 \rightarrow p = \lfloor 2\epsilon N \rfloor = 3$
band 0: Deltas $= \{3\}$
band 1: Deltas $= \{1, 2\}$
band 2: Deltas $= \{0\}$
T:



VII) Iteration 7: Observation 11 arrives

a) $N \mod 1/2\epsilon = 0$: must perform compression
Testing: $(11, 1, 0)$ $(12, 1, 0)$
Band: $2 <= 2 \rightarrow True$
$2 + 1 + 0 < 3 \rightarrow False$
Cannot delete $(11, 1, 0)$
Testing: $(10, 1, 1)$ $(11, 1, 0)$
Band: $1 <= 2 \rightarrow True$
$1 + 1 + 0 < 3 \rightarrow True$
Can delete $(10, 1, 1)$: Merge $(10, 1, 1)$ with its parent $(11, 1, 0)$.
$S = \{(1, 1, 0), (10, 1, 0), (10, 1, 0), (11, 2, 0), (12, 1, 0)\}$
Testing: $(10, 1, 0)$ $(11, 2, 0)$
Band: $2 <= 2 \rightarrow True$
$1 + 2 + 0 < 3 \rightarrow False$
Cannot delete $(10, 1, 0)$
Testing: $(10, 1, 0)$ $(10, 1, 0)$
Band: $2 <= 2 \rightarrow True$
$1 + 1 + 0 < 3 \rightarrow True$
Can delete $(10, 1, 0)$: Merge $(10, 1, 0)$ with its sibling $(10, 1, 0)$
$S = \{(1, 1, 0), (10, 2, 0), (11, 2, 0), (12, 1, 0)\}$
Testing: $(1, 1, 0)$ $(10, 2, 0)$
Band: $2 <= 2 \rightarrow True$
$1 + 2 + 0 < 3 \rightarrow False$
Cannot delete $(1, 1, 0)$

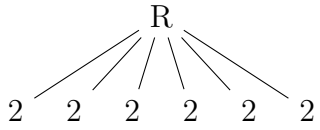
b) Insert observation 11 into summary S:
$\Delta(11) = \lfloor 2\epsilon N \rfloor - 1 = \lfloor 3 \rfloor - 1 = 2$
$S = \{(1, 1, 0), (10, 2, 0), (11, 2, 0), (11, 1, 2), (12, 1, 0)\}$
$N = N + 1 = 7 \rightarrow 2\epsilon N = 3.5 \rightarrow p = \lfloor 2\epsilon N \rfloor = 3$
band 0: Deltas $= \{3\}$
band 1: Deltas $= \{1, 2\}$
band 2: Deltas $= \{0\}$
T:



VIII) Iteration 8: Observation 9 arrives

a) $N \mod 1/2\epsilon \neq 0$: No compression to be performed

b) Insert observation 9 into summary S:
$\Delta(9) = \lfloor 2\epsilon N \rfloor - 1 = \lfloor 3.5 \rfloor - 1 = 2$
$S = \{(1,1,0),(9,1,2),(10,2,0),(11,2,0),(11,1,2),(12,1,0)\}$
$N = N + 1 = 8 \rightarrow 2\epsilon N = 4 \rightarrow p = \lfloor 2\epsilon N \rfloor = 4$
band 0: Deltas $= \{4\}$
band 1: Deltas $= \{3\}$
band 2: Deltas $= \{0,1,2\}$
T:



Now lets take a look at our summary so far:

$$S = \{(1,1,0),(9,1,2),(10,2,0),(11,2,0),(11,1,2),(12,1,0)\}$$

We will try to find a 0.5-quantile. We first calculate the rank $r = \lfloor 0.5 \cdot 8 \rfloor = 4$. Then we calculate r_{max} and r_{min} for each value $v \in S$. If $r_{max}(v) \leq r + \lfloor \epsilon N \rfloor = 6$ ($\epsilon N = 0.25 \cdot 8 = 2$) and $r_{min}(v) \geq r - \lfloor \epsilon N \rfloor = 2$, then v is a 0.25-approximate 0.5-quantile.
$r_{min}(1) = 1$
$r_{max}(1) = r_{min}(1) + \Delta(1) = 1 + 0 = 1$

$r_{min}(9) = r_{min}(1) + g(9) = 1 + 1 = 2 \rightarrow 2 \geq 2$
$r_{max}(9) = r_{min}(9) + \Delta(9) = 2 + 2 = 4 \rightarrow 4 \leq 6$

$r_{min}(10) = r_{min}(9) + g(10) = 2 + 2 = 4 \rightarrow 4 \geq 2$
$r_{max}(10) = r_{min}(10) + \Delta(10) = 4 + 0 = 4 \rightarrow 4 \leq 6$

$r_{min}(11) = r_{min}(10) + g(11) = 4 + 2 = 6 \rightarrow 6 \geq 2$
$r_{max}(11) = r_{min}(11) + \Delta(11) = 6 + 0 = 6 \rightarrow 6 \leq 6$

$r_{min}(11) = r_{min}(11) + g(11) = 6 + 1 = 7$
$r_{max}(11) = r_{min}(11) + \Delta(11) = 7 + 2 = 9$

$r_{min}(12) = r_{min}(11) + g(12) = 7 + 1 = 8$
$r_{max}(12) = r_{min}(12) + \Delta(12) = 8 + 0 = 8$

As we can see each one of the values 9, 10 and 11 is a 0.25-approximate 0.5-quantile.

Chapter 3

A Demand-Aware Heuristic for Value-Space Partitioning and Repartitioning

3.1 Model

Let A denote a set of attributes of size m, and let $\{a_i\}_{1 \leq i \leq m}$ denote the elements of A. Let $R(a_i)$ denote the range set that corresponds to the values a_i can assume. We define a value-space - represented as an m-dimensional vector of attribute-range pairs $[(a_1, [l_1, h_1]), \cdots, (a_m, [l_m, h_m])]$, where $a_i \in A$ and $[l_i, h_i] \subseteq R(a_i)$ - as the set of all vectors $[v_1, \cdots, v_m]$ such that $v_i \in [l_i, h_i]$, with $1 \leq i \leq m$. Let $H(A)$ denote the global value-space represented as $[(a_1, R(a_1)), \cdots, (a_m, R(a_m))]$. Let D denote a key-value datastore (i.e., a collection of key-value records) represented as $\{\langle g_1, [(a_1, v_1), \cdots, (a_m, v_m)]\rangle, \cdots\}$, where g_1 is a globally unique identifier (GUID) that is the key of the record $\langle g_1, [(a_1, v_1), \cdots, (a_m, v_m)]\rangle$, and $[(a_1, v_1), \cdots, (a_m, v_m)]$ is an m-dimensional vector of attribute-value pairs (with $a_i \in A$ and $v_i \in R(a_i)$, $1 \leq i \leq m$) that is the value of the record $\langle g_1, [(a_1, v_1), \cdots, (a_m, v_m)]\rangle$. We use the notation $g_i.a_i$ to represent the value v_i of attribute a_i of GUID g_i.

Our model supports two operations over D. i) A search that is represented as a k-dimensional vector of attribute-range pairs $[(a_{n_1}, I_{n_1}), \cdots, (a_{n_k}, I_{n_k})]$ with $1 \leq n_1 \leq \cdots \leq n_k \leq m$, where $a_{n_i} \in A$ and $I_{n_i} = [low_{n_i}, high_{n_i}] \subseteq R(a_{n_i})$, $1 \leq i \leq k$. This operation returns every GUID g such that $(low_{n_1} \leq g.a_{n_1} \leq high_{n_1}) \wedge \cdots \wedge (low_{n_k} \leq g.a_{n_k} \leq high_{n_k})$. We use the notation $s.I_{n_i}$ to represent the range I_{n_i} of search s. ii) An update that is represented as a 2-tuple consisting of a GUID and a k-dimensional vector of attribute-value pairs $\langle g, [(a_{n_1}, v_{n_1}), \cdots, (a_{n_k}, v_{n_k})]\rangle$ with $1 \leq n_1 \leq \cdots \leq n_k \leq m$, where $a_{n_i} \in A$ and $v_{n_i} \in R(a_{n_i})$, $1 \leq i \leq k$. This operation assigns $g.a_{n_i} \leftarrow v_{n_i}$ for every $1 \leq i \leq k$.

We define a region as a value-space that is represented as an m-dimensional vector of attribute-range pairs $[(a_1, I_1), \cdots, (a_m, I_m)]$, where $a_i \in A$ and $I_i = [l_i, h_i) \subseteq R(a_i)$, with $1 \leq i \leq m$. We use the notation $r.I_i$ to represent the range I_i of region r. A region is a partition of the global value-space $H(A)$. Therefore, the intersection of any two regions is empty and the generalized union of regions is $H(A)$. Next we give an example of regions:

- $H(A) : [(a_1, [0, 1)), (a_2, [0, 1)), (a_3, [0, 1))]$
- Regions: $R = \{r_1, r_2, r_3, r_4\}$, where

  $r_1 = [(a_1, [0, 0.35)), (a_2, [0, 1)), (a_3, [0, 1))]$

  $r_2 = [(a_1, [0.35, 1)), (a_2, [0, 0.35)), (a_3, [0, 1))]$

  $r_3 = [(a_1, [0.35, 1)), (a_2, [0.35, 1)), (a_3, [0, 0.5))]$

  $r_4 = [(a_1, [0.35, 1)), (a_2, [0.35, 1)), (a_3, [0.5, 1))]$

The attributes (or equivalently, the elements of A) can be seen as the axes of the global value-space. The Figures 3.1 and 3.2 show respectively the global value space before and after being partitioned into the four regions from the previous example.

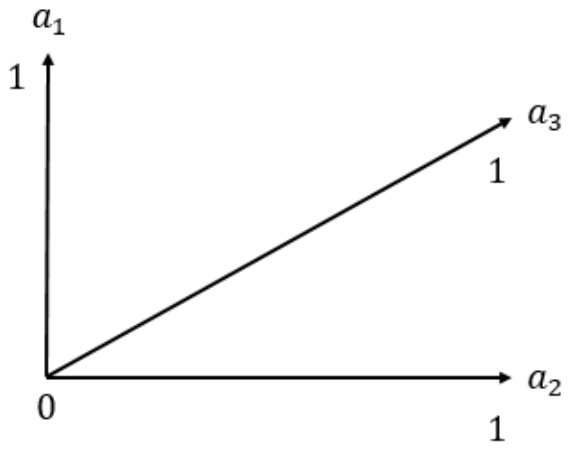


Figure 3.1: Global value-space before being partitioned: there is only one region that covers the entire value space

3.2 Touches

In this section we define what we call touches. We first define update touches, and then we define search touches. In the end we illustrate the idea with a comprehensive example.

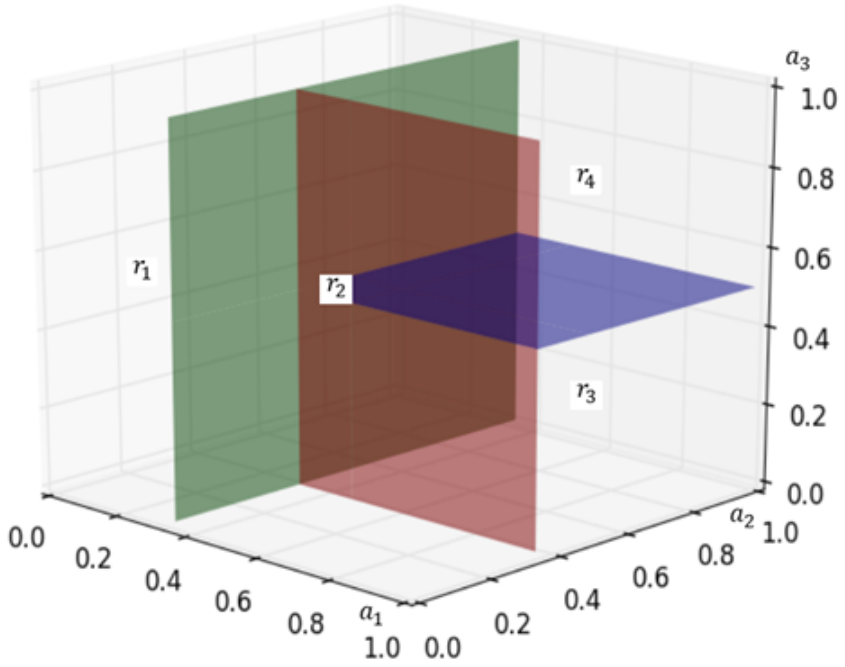


Figure 3.2: Global value-space after being partitioned into 4 regions: r_1, r_2, r_3 and r_4

3.2.1 Update touches

Let r denote a region (i.e., an m-dimensional value space).
Case 1)
Let g denote a GUID that has not yet been assigned to any record.
Let u denote an update operation that sets the attributes $\{a_{n_1}, \cdots, a_{n_k}\}$ of g, with $1 \leq n_1 \leq \cdots \leq n_k \leq m$, such that every $g.a_{n_i} \in r.I_{n_i}$, with $1 \leq i \leq k$.
In such case, we say that u touches r.
Case 2)
Let g denote a GUID such that for every $1 \leq i \leq m$, $g.a_i \in r.I_i$.
Let u denote an update operation that modifies the attributes $\{a_{n_1}, \cdots, a_{n_k}\}$ of g, with $1 \leq n_1 \leq \cdots \leq n_k \leq m$.
Case 2.1)
Suppose u modifies the attributes of g such that it is still true that for every $1 \leq i \leq m$, $g.a_i \in r.I_i$.
In such case, we say that u touches r.
Case 2.2)
Suppose u modifies the attributes of g such that there exists $1 \leq i \leq m$, $g.a_i \notin r.I_i$.
Then there must exist a region r' such that for every $1 \leq i \leq m$, $g.a_i \in r'.I_i$.
In such case, we say that u touches r and r'.

3.2.2 Search touches

Let R denote the set of all regions.
Let s denote a search operation.
Let $R' \subseteq R$ denote a subset of regions such that for all $r \in R'$ it is true that $s.I_{n_i} \cap r.I_{n_i} \neq \emptyset$, for every $1 \leq i \leq k$.
Then we define a set G_r of GUIDs, for every $r \in R'$, such that for all $g \in G_r$ it is true that $g.a_{n_i} \in s.I_{n_i} \cap r.I_{n_i}$, for every $1 \leq i \leq k$. (Note that G_r can be an empty set)
For every $r \in R'$, we say that the cardinality of G_r is the number of touches on r because of s.

3.2.3 Example

Assume the global value-space is divided into these 3 regions:

- $r_1 = [(a_1, [0, 0.33)), (a_2, [0, 1]), (a_3, [0, 1])]$
- $r_2 = [(a_1, [0.33, 0.66)), (a_2, [0, 1]), (a_3, [0, 1])]$
- $r_3 = [(a_1, [0.66, 1)), (a_2, [0, 1]), (a_3, [0, 1])]$

We will organize this example into epochs. At each epoch, one operation will be issued.

Epoch 1: Assume that an *update* operation is issued:

$$\langle GUID_1, [(a_1, 0.78), (a_2, 0.36), (a_3, 0.91)] \rangle$$

This *update* touches the region r_3 (because the value for each attribute is inside the respective range of this region) and, thus, the $GUID_1$ is placed in this region.

Thus,

	r_1	r_2	r_3
GUIDs	{}	{}	{1}
# touches	0	0	1

Epoch 2: An *update* operation is issued:

$$\langle GUID_2, [(a_1, 0.15), (a_2, 0.43), (a_3, 0.02)] \rangle$$

This *update* touches the region r_1 and, thus, the $GUID_2$ is placed in this region.

Thus,

	r_1	r_2	r_3
GUIDs	$\{2\}$	$\{\}$	$\{1\}$
# touches	1	0	1

Epoch 3: An *update* operation is issued:

$$\langle GUID_3, [(a_1, 0.49), (a_2, 0.22), (a_3, 0.1)] \rangle$$

This *update* touches the region r_2 and, thus, the $GUID_3$ is placed in this region. Thus,

	r_1	r_2	r_3
GUIDs	$\{2\}$	$\{3\}$	$\{1\}$
# touches	1	1	1

Epoch 4: An *update* operation is issued:

$$\langle GUID_4, [(a_1, 0.24), (a_2, 0.9), (a_3, 0.37)] \rangle$$

This *update* touches the region r_1 and, thus, the $GUID_4$ is placed in this region. Thus,

	r_1	r_2	r_3
GUIDs	$\{2, 4\}$	$\{3\}$	$\{1\}$
# touches	2	1	1

Epoch 5: An *update* operation is issued:

$$\langle GUID_5, [(a_1, 0.75), (a_2, 0.53), (a_3, 0.93)] \rangle$$

This *update* touches the region r_3 and, thus, the $GUID_5$ is placed in this region. Thus,

	r_1	r_2	r_3
GUIDs	$\{2, 4\}$	$\{3\}$	$\{1, 5\}$
# touches	2	1	2

Epoch 6: An *update* operation is issued:

$$\langle GUID_6, [(a_1, 0.42), (a_2, 0.12), (a_3, 0.33)] \rangle$$

This *update* touches the region r_2 and, thus, the $GUID_6$ is placed in this region. Thus,

	r_1	r_2	r_3
GUIDs	$\{2, 4\}$	$\{3, 6\}$	$\{1, 5\}$
# touches	2	2	2

Epoch 7: An *update* operation is issued:

$$\langle GUID_7, [(a_1, 0.13), (a_2, 0.39), (a_3, 0.07)]\rangle$$

This *update* touches the region r_1 and, thus, the $GUID_7$ is placed in this region. Thus,

	r_1	r_2	r_3
GUIDs	$\{2, 4, 7\}$	$\{3, 6\}$	$\{1, 5\}$
# touches	3	2	2

Epoch 8: An *update* operation is issued:

$$\langle GUID_8, [(a_1, 0.96), (a_2, 0.18), (a_3, 0.65)]\rangle$$

This *update* touches the region r_3 and, thus, the $GUID_8$ is placed in this region. Thus,

	r_1	r_2	r_3
GUIDs	$\{2, 4, 7\}$	$\{3, 6\}$	$\{1, 5, 8\}$
# touches	3	2	3

Epoch 9: An *update* operation is issued:

$$\langle GUID_2, [(a_1, 0.85), (a_2, 0.62), (a_3, 0.96)]\rangle$$

This *update* touches the regions r_1 and r_3. It touches r_1 because it is where $GUID_2$ was previously in (it needs to touch this region in order to update this GUID). It also touches r_3 because its where $GUID_2$ is moving to.

Thus,

	r_1	r_2	r_3
GUIDs	$\{4, 7\}$	$\{3, 6\}$	$\{1, 2, 5, 8\}$
# touches	4	2	4

Epoch 10: An *update* operation is issued:

$$\langle GUID_6, [(a_1, 0.34), (a_2, 0.55), (a_3, 0.28)]\rangle$$

This *update* touches only the region r_2, as $GUID_6$ stays in this region after being updated.

Thus,

	r_1	r_2	r_3
GUIDs	$\{4, 7\}$	$\{3, 6\}$	$\{1, 2, 5, 8\}$
# touches	4	3	4

Epoch 11: An *update* operation is issued:

$$\langle GUID_1, [(a_1, 0.18), (a_2, 0.51), (a_3, 0.17)] \rangle$$

This *update* touches the regions r_1 and r_3. Also, $GUID_1$ goes to r_1.
Thus,

	r_1	r_2	r_3
GUIDs	$\{1, 4, 7\}$	$\{3, 6\}$	$\{2, 5, 8\}$
# touches	5	3	5

Epoch 12: An *update* operation is issued:

$$\langle GUID_3, [(a_1, 0.65), (a_2, 0.66), (a_3, 0.92)] \rangle$$

This *update* touches only the region r_2, and $GUID_3$ does not move.
Thus,

	r_1	r_2	r_3
GUIDs	$\{1, 4, 7\}$	$\{3, 6\}$	$\{2, 5, 8\}$
# touches	5	4	5

Epoch 13: An *update* operation is issued:

$$\langle GUID_9, [(a_1, 0.55), (a_2, 0.41), (a_3, 0.94)] \rangle$$

This *update* touches the region r_2 and, thus, the $GUID_9$ is placed in this region.
Thus,

	r_1	r_2	r_3
GUIDs	$\{1, 4, 7\}$	$\{3, 6, 9\}$	$\{2, 5, 8\}$
# touches	5	5	5

Epoch 14: An *update* operation is issued:

$$\langle GUID_{10}, [(a_1, 0.41), (a_2, 0.61), (a_3, 0.31)] \rangle$$

This *update* touches the region r_2 and, thus, the $GUID_{10}$ is placed in this region.
Thus,

	r_1	r_2	r_3
GUIDs	$\{1, 4, 7\}$	$\{3, 6, 9, 10\}$	$\{2, 5, 8\}$
# touches	5	6	5

Next we show the GUIDs after the *update* operations:

	a_1	a_2	a_3
$GUID_1$	0.18	0.51	0.17
$GUID_2$	0.85	0.62	0.96
$GUID_3$	0.65	0.66	0.92
$GUID_4$	0.24	0.9	0.37
$GUID_5$	0.75	0.53	0.93
$GUID_6$	0.34	0.55	0.28
$GUID_7$	0.13	0.39	0.07
$GUID_8$	0.96	0.18	0.65
$GUID_9$	0.55	0.41	0.94
$GUID_{10}$	0.41	0.61	0.31

Epoch 15: A *search* operation is issued:

$$[(a_1, [0.14, 0.42]), (a_2, [0.5, 1]), (a_3, [0, 0.4])]$$

This *search* touches the regions r_1 and r_2 (because the ranges of these regions overlap the range of this search for all attributes), and the GUIDs 1, 4, 6 and 10 (because these GUIDs satisfy the range of this search). The search touches each region for each GUID that meets the search criteria (r_1 and r_2 are touched twice each).

Thus,

	r_1	r_2	r_3
GUIDs	$\{1, 4, 7\}$	$\{3, 6, 9, 10\}$	$\{2, 5, 8\}$
# touches	7	8	5

Epoch 16: A *search* operation is issued:

$$[(a_1, [0.55, 0.9]), (a_2, [0.4, 0.7]), (a_3, [0.9, 1])]$$

This *search* touches the regions r_2 and r_3, and the GUIDs 2, 3, 5 and 9. Region r_2 and r_3 are touched twice each.

Thus,

	r_1	r_2	r_3
GUIDs	$\{1, 4, 7\}$	$\{3, 6, 9, 10\}$	$\{2, 5, 8\}$
# touches	7	10	7

Epoch 17: A *search* operation is issued:

$$[(a_1, [0.3, 0.7]), (a_2, [0.41, 0.66]), (a_3, [0.28, 0.94])]$$

This *search* touches only region r_2, but touches all its GUIDs (3, 6, 9 and 10). Region r_2 is touched 4 times.

Thus,

	r_1	r_2	r_3
GUIDs	$\{1, 4, 7\}$	$\{3, 6, 9, 10\}$	$\{2, 5, 8\}$
# touches	7	14	7

Region r_2 was touched 14 times, while region r_1 and r_3 were touched only 7 times each. As one can see in this example, it is clear the system's unbalance. The aim of our heuristic is to re-split the regions in such a way it reaches a better balance for future operations.

A new configuration for these regions could be:

- $r_1 = [(a_1, [0, 0.35)), (a_2, [0, 1]), (a_3, [0, 1])]$
- $r_2 = [(a_1, [0.35, 0.64)), (a_2, [0, 1]), (a_3, [0, 1])]$
- $r_3 = [(a_1, [0.64, 1)), (a_2, [0, 1]), (a_3, [0, 1])]$

3.3 Our heuristic

Let n denote the size of the global set of machines. We partition the global value-space $H(A)$ into $\sqrt{n}$ mutually exclusive and exhaustive (MEE) regions. Each region is assigned to $\sqrt{n}$ machines and, thus, it is replicated $\sqrt{n}$ times[7]. Each update touches at most $2\sqrt{n}$ machines ($\sqrt{n}$ machines assigned to the previous region and $\sqrt{n}$ machines assigned to the current region). A search never touches more than $\sqrt{n}$ machines, as a subset of $\sqrt{n}$ machines already covers the entire value space. The Figure 3.3 shows the global value space partitioned into three regions (partitions P1, P2, and P3), with each region assigned to a different subset of three machines.

We partition the global value-space $H(A)$ regarding a single attribute axis. We look for $\sqrt{n} - 1$ quantiles (points) on that attribute axis; then we divide $H(A)$ at those points, creating $\sqrt{n}$ partitions (regions). In order to find the quantiles, we compute the touches generated by each operation; then we utilize a variation of the Greenwald-Khanna algorithm with a sliding window to find the quantiles, using the

[7]We will use $\sqrt{n}$ regions and replicate each region $\sqrt{n}$ times because the authors of CNS prove in their article that there is no way of reaching linear scalability and the closest one can achieve is sublinear scalability of $\sqrt{n}$.

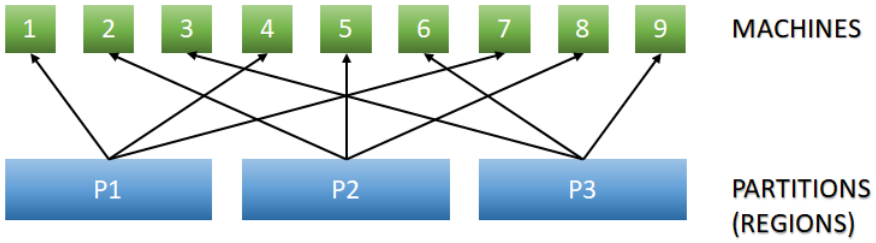


Figure 3.3: Global value space partitioned into 3 regions (partitions P1, P2, and P3) and assigned to a set of 9 machines

touches as input. Next we give an example on how our heuristic handles operations as input.

Say we have just seen three update operations:

- $\{"GUID" : "\text{android-450e0415e1fb32b1}", "Latitude" : 55.859129\}$
- $\{"GUID" : "\text{android-1dbf1ed7cd014c85}", "Latitude" : 45.233800\}$
- $\{"GUID" : "\text{android-e2ae1c3e2914c9ce}", "Latitude" : 37.428524\}$

The input to the Greenwald-Khanna algorithm would be 55.859129, 45.233800 and 37.428524. We essentially convert a stream of operations into a stream of touches (i.e. touches at those points or equivalently at the regions associated with those points). Then we partition the global value-space so that touches are distributed among regions in a balanced way. This way we also ensure that touches are fairly distributed among machines.

3.4 The Algorithm

At every instant of time an operation is issued. Each operation is then converted into an observation so that it can be an input to the GK algorithm. This is what the Algorithms 1 and 2 do.

The Algorithm 1 receives an operation as its input. It then calls the Algorithm 2 in order to create an array of observations from that operation. These observations are essentially the points that were touched by the operation. Each observation is then inserted into the GK-Window algorithm so that it can create its summary, which will help us later on finding the quantiles points that are used to partition the value-space into regions.

The Algorithm 2 also receives an operation as its input. It firstly creates an empty array of observations. Then, it checks whether the operation is an update or a search. If its an update, it checks the GUID's previous value associated to a certain axis (recall that we are going to partition the value-space regarding a single

Algorithm 1 insertGK(Operation op)

1: *observations* ← **convertTouchesIntoGKObservations**(op)
2: **for all** *obs* ∈ *observations* **do**
3: **greenwaldKhannaWindow**(*obs*) ▷ Inserts the observation into the GK-Window algorithm, which then updates its list of blocks (summaries)
4: **end for**

attribute), and the new value (also recall that each update has a GUID associated with it and that it updates the values of its GUID). If this GUID is not new, it means that it has a previous value for that attribute axis. It also means that this update touches the point represented by this value. Thus, we add this value to the observations array. Because updating the values of a GUID might mean that this GUID is moving from a point to another, we update the number of GUIDs at the former point by decreasing it by one (we maintain a map or dictionary to keep this kind of information). Also, if the new value (for that axis) is actually a new value, then we add this value to the array of observations. We now have to update the number of GUIDs at the point represented by this new value - we do it by increasing it by one. On the other hand, if the operation is a search, we iterate over the points with GUIDs located at it. For each of these points, we check whether it is in the search's range. If it is inside the range, the number of GUIDs at this point is the number of touches at it. This is because the search must return all the GUIDs that meet its criteria. Thus, we add this point to the array of observations the amount of times it was touched.

By this time, GK-Window algorithm has already created a list of blocks or summaries that enables us to compute the quantile points at any moment, should we need to partition or repartition the value-space.

The Algorithm 3 is invoked to do the partitioning or repartitioning. It firstly creates an empty array of regions. Then, it calls the Algorithm 4 in order to have a list of quantiles. Knowing the quantiles, it creates each one of the regions by using the quantiles to define the region's range. Each newly created region is then added to the array of regions. Finally, it returns the array of regions.

The Algorithm 4 is invoked to find the quantile points used to partition or repartition the value-space. It firstly creates a map (or equivalently, a dictionary) of quantiles where the key is the *phi*, and the value is the quantile itself. For each value of *phi*, which is $i/number\_of\_regions$ with $1 \leq i \leq number\_of\_regions - 1$, it invokes the GK-Window algorithm to calculate the quantile associated to that *phi*. In order to calculate the quantile, GK-Window combines its block summaries into a single summary. Then, it uses the original GK algorithm to find the quantile from this combined summary. After calculating the quantile, our algorithm adds it to the map of quantiles. In the end, it returns this map of quantiles.

Algorithm 2 convertTouchesIntoGKObservations(Operation op)

```
1: observations ← [ ]
2: if op is UPDATE then
3:     guid_old_value ← op.getGUIDPreviousValue(axis)          ▷ Gets the GUID's previous value associated to axis
4:     guid_new_value ← op.getGUIDNewValue(axis)               ▷ Gets the GUID's new value associated to axis
5:     if guid_old_value ≠ −1 then                             ▷ If it is not a new guid
6:         observations.add(guid_old_value)                    ▷ Update touches the point guid_old_value
7:         previous_count ← guidsPerPoint.get(guid_old_value)  ▷ Checks the number of guids at that point
8:         new_count ← previous_count - 1                      ▷ Decreases that number
9:         guidsPerPoint.put(guid_old_value, new_count)        ▷ Updates the number of guids at that point
10:    end if
11:    if guid_old_value ≠ guid_new_value then                 ▷ If the update modifies this attribute value
12:        observations.add(guid_new_value)                    ▷ then it touches the point guid_new_value
13:    end if
14:    previous_count ← guidsPerPoint.get(guid_new_value)      ▷ Checks the number of guids at that point
15:    guidsPerPoint.put(guid_old_value, previous_count+1)     ▷ Updates the number of guids at that point
16: else                                                       ▷ If op is SEARCH
17:    for all point in guidsPerPoint with guids_count > 0 do
18:        if point ≥ op.search_low_range AND point ≤ op.search_high_range then
           ▷ If this point is inside the search's range
19:            touchesAtThisPoint ← guidsPerPoint(point)       ▷ then the number of guids at this point is the number of touches at this point
20:            i ← 0
21:            while i < touchesAtThisPoint do
22:                observations.add(point)                     ▷ The point is observed the amount of times it was touched
23:                i++
24:            end while
25:        end if
26:    end for
27: end if
```

Algorithm 3 partitionGK()

```
1: regions ← [ ]
2: r ← 1
3: low ← 0
4: high ← 1
5: quantiles ← findQuantiles()
6: for all quantile in quantiles do
7:     new_region ← createRegion("Region #"+r)          ▷ Creates new region
8:     new_region.setRange(axis, low, quantile)   ▷ Sets its range to [low, quantile) for axis
9:     regions.add(new_region)                    ▷ Adds this new region to regions
10:    r++
11:    low ← quantile
12: end for
13: new_region ← createRegion("Region #"+r)
14: new_region.setRange(axis, low, high)
15: regions.add(new_region)
16: return regions
```

Algorithm 4 findQuantiles

```
1: quantiles_map ← { }
2: number_of_regions ← √number_of_machines
3: i ← 1
4: while i ≤ number_of_regions − 1 do
5:     phi ← i/number_of_regions
6:     quantile ← GKWindowQuantile(phi)      ▷ GK-Window algorithm finds quantile given phi
7:     quantiles_map.put(phi, quantile)
8:     i++
9: end while
10: return quantiles_map
```

Chapter 4

Experiments and Results

4.1 Experiments

For the experiments we implemented our own simulator in JAVA. We simulated 64 machines with 24-attribute GUIDs distributed across them. We generated 2^{13} GUIDs and 2^{18} operations for each experiment, for a total of 120 experiments. Let RHO be the fraction of search queries among all operations. We performed 30 experiments for each one of the following values for RHO: 0, 0.25, 0.5 and 0.75.

Let A_1, A_2, $\cdots$, A_{24} be random variables that denote the values that each attribute of an update operation can assume. Let $A_{1,low}$, $A_{1,high}$, $A_{2,low}$, $A_{2,high}$, $\cdots$, $A_{24,low}$ and $A_{24,high}$ be random variables that denote the values for *low* and *high* range that each attribute of a search operation can assume.

We divided each experiment into 4 epochs and for each epoch we randomly pick a distribution (among uniform, normal and exponential distributions) for A_1, A_2, $\cdots$, A_{24}, $A_{1,low}$, $A_{1,high}$, $A_{2,low}$, $A_{2,high}$, $\cdots$, $A_{24,low}$ and $A_{24,high}$. We also set the distributions so that $A_{1,high}$ follows same distribution as $A_{1,low}$, $A_{2,high}$ follows same distribution as $A_{2,low}$, and so on. That is, the random variables for the range of a single attribute follow the same distribution. The parameters for the random distributions are also randomly set. Having set the distributions, we generate the operations.

We generate an update operation by associating a GUID to it and generating a value for each attribute from the respective random variable. In order to generate a search query, we firstly randomly set the number of attributes it will define. Once we know the number of attributes this search will define, we randomly pick attributes from the set of attributes and generate a value for each attribute range from the respective random variable. Picking random attributes for a search means that not every search query will define the attribute our heuristic is using to partition the value-space. In fact, many searches will not define this attribute. It is important

for us because it enables us to validate that our heuristic works even when there are operations that do not define the attribute it picked to perform the partitioning.

We simulated 6 different heuristics for space partitioning: CNS (a heuristic proposed by the authors of CNS), Quantiles (a version of our heuristic that finds the quantiles without using the GK algorithm), Quantiles+GK (our heuristic that uses a sliding-window version of GK algorithm), Replicate-At-All, Query-All and HyperDex.

We implemented a different version of our heuristic which calculates the real quantiles, without making use of the GK algorithm. The reason for that is we want to compare these versions of our heuristic and be able to see whether the GK algorithm makes the heuristic lose performance as this algorithm accepts a margin of error to create its summary. One may ask why not always use the version that calculates real quantiles as it does not work with a margin of error. We use summaries for a reason. We do not want to have to go over the whole stream of data, especially if it is endless, every time we need to calculate a quantile.

The Replicate-At-All heuristic replicates the entire global value space for all machines so that each machines has the same content. The Query-All heuristic uses a consistent hashing algorithm to calculate which machine is associated to each GUID. The HyperDex heuristic is a well known solution for the problem of subspace partitioning. It divides the global value space into multiple lower-dimensional subspaces. Each of these subspaces uses a subset of attributes as their axes. It then divides each subspace into non-overlapping regions and assigns a machine to each region. Whenever there is a search, it picks the subspace with the subset of attributes that best match its attributes in order to contact the fewest machines. We implemented our own HyperDex heuristic: it divides the 24-attribute hyperspace into 8 3-attribute hyperspaces with 8 regions each. Each region is then assigned to 1 machine. Our heuristics and CNS divide the global value-space in the same way (into $\sqrt{n}$ disjoint regions, each one assigned to $\sqrt{n}$ machines, where n = total number of machines), but we use quantiles to create regions and CNS uses a weighted JFI function (see section 4.1.1).

In each experiment, initially all heuristics generate their own region configuration for the global value-space. As operations arrive, the demand aware heuristics (Quantiles, Quantiles+GK and CNS that take into account the operations' distributions to create regions) periodically perform repartitionings in order to reach a better balance of touches on regions. It is particularly useful as in each epoch the distributions change.

4.1.1 Metrics

JFI based on touches

This metric captures the notion of how well balanced the regions are in terms of touches. It returns a value between 0 and 1. The closer to 1, the better the performance of the heuristic we are evaluating. The closer to 0, the worse the performance of this heuristic. We use a weighted function based on the original *Jain's Fairness Index* (JFI)[6] to calculate this metric.

$$wJFI = \rho \cdot JFI_{searches} + (1 - \rho) \cdot JFI_{updates}, \text{ where:}$$

$$JFI_{updates} = \frac{(\sum_{r=1}^{n} u_r)^2}{n \cdot \sum_{r=1}^{n} u_r^2}, \text{ with } u_r = \text{update touches on region } r \text{ and } n = \text{number of regions.}$$

$$JFI_{searches} = \frac{(\sum_{r=1}^{n} s_r)^2}{n \cdot \sum_{r=1}^{n} s_r^2}, \text{ with } s_r = \text{search touches on region } r \text{ and } n = \text{number of regions.}$$

$$RHO = \rho = \text{fraction of searches.}$$

CNS works with a variation of this weighted function, but instead of working with touches of update and search operations per region, it uses the total number of update and search operations per region.

JFI based on the number of GUIDs per region

This metric captures the notion of how well balanced the regions are in terms of number of GUIDs per region. The JFI value is calculated based on the following function:

$$JFI_{\#GUIDs} = \frac{(\sum_{r=1}^{n} g_r)^2}{n \cdot \sum_{r=1}^{n} g_r^2}, \text{ with } g_r = \text{number of guids on region } r \text{ and } n = \text{number of regions.}$$

Number of Messages per Machine

This metric captures the notion of load (i.e. number of touches) on each machine represented by the number of messages sent to each machine. For CNS and our heuristic, an update imposes the sending of 1 message to at least $\sqrt{n}$ machines and at most $2\sqrt{n}$ machines (in case this update moves a GUID from one region to another), where n is the total number of machines. A search imposes the sending of 1 message to any machine associated to each touched region. For HyperDex, an update touches up to 2 machines per existing subspace, which means it imposes the sending of 1 message to at least 1 machine per existing subspace and at most

2 machines (per existing subspace). This is because it touches up to 2 regions per subspace (depending on whether it moves a GUID from one region to another). A search touches only one subspace and will impose the sending of 1 message to 1 machine associated to each touched region. For Query-All, an update touches only one machine, which means it imposes the sending of 1 message to 1 machine. A search touches all the machines and thus it imposes the sending of 1 message to each existing machine. For Replicate-At-All, an update touches all the machines and thus it imposes the sending of 1 message to each existing machine. A search touches only one machine, which means it imposes the sending of 1 message to 1 machine. We also consider repartitioning messages. These messages are sent to machines in order to move its content (GUIDs) to other machines because of a repartitioning. For simplicity's sake, we consider that a group of contents leaving or coming to a machine imposes one message to that machine.

At every 30 experiments for a value of RHO we generated 3 graphs: *JFI (based on touches per region) per Repartitioning*[8], *JFI (based on GUIDs per region) per Repartitioning*, and *No. of Messages per Machine*. Each point in these graphs corresponds to the mean of the values for that same point in each of the 30 experiments.

In the end of the 120 experiments, we generated 3 graphs: *JFI (based on touches per region) per RHO*, *JFI (based on GUIDs per region) per RHO*, and *No. of Messages per RHO*. Each point (RHO) in these graphs corresponds to the mean of the values in the previous graphs associated to this RHO.

4.2 Results

The graphs 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7 and 4.8 show the JFI over time as more operations arrive to the system and repartitionings are performed. Only the demand aware heuristics (heuristics I, II and III; or equivalently *CNS*, *Quantiles*, *Quantiles+GK*) perform repartitioning, as they are the only ones that take into account the operations' distributions to perform the partitioning. We also decided to perform repartitioning with CNS heuristic (heuristic I), even though it takes too long to perform each one of the repartitionings, in order to show that performing repartitioning improves the balance of regions. The valleys in these graphs are because of the changes in the distributions that occur in each epoch (regions that were created based on the previous distribution of touches are not touched fairly anymore

[8]Each experiment is divided in 4 epochs. In each of these epochs, operations are generated following a different distribution. Repartitioning is performed periodically with time. Right before a repartitioning, we calculate the JFI value in order to check how fair regions are touched. The x-axis here is repartitionings exactly because we are calculating the metric right before each repartitioning.

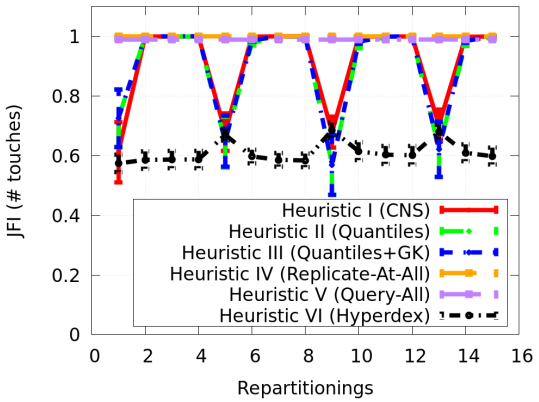


Figure 4.1: JFI (touches) X Repartitioning (RHO 0)

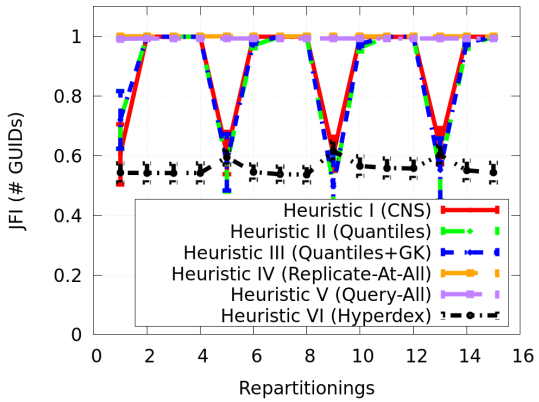


Figure 4.2: JFI (GUIDs) X Repartitioning (RHO 0)

because this distribution changed). As repartitionings are performed, the results are improved.

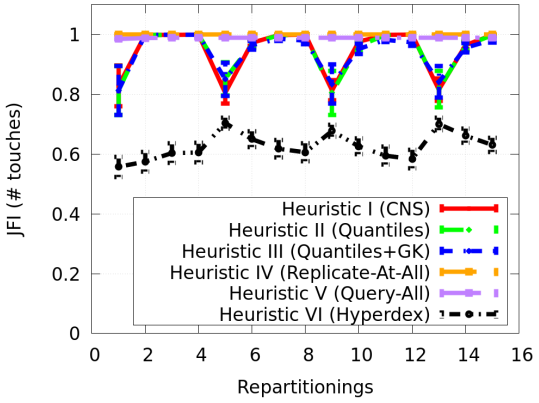


Figure 4.3: JFI (touches) X Repartitioning (RHO 0.25)

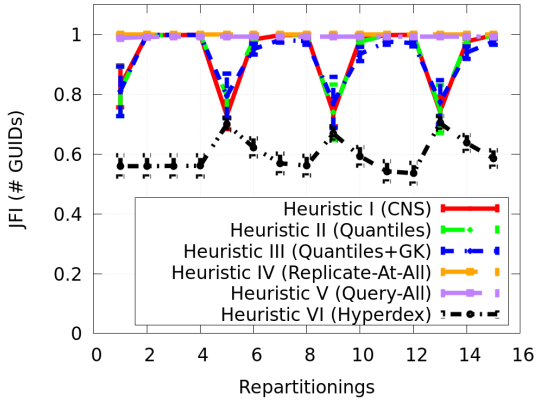


Figure 4.4: JFI (GUIDs) X Repartitioning (RHO 0.25)

The *CNS'* performance is worse when $RHO = 0.75$ (when there are more searches). This is due to the metric it adopts to perform the partitioning, which only takes into account the number of operations processed in each region. Thus, a search is only counted one time in a region regardless of the amount of GUIDs it contains. We did not use the metric of touches for this heuristic as it is not adopted by the authors of the article. Moreover, it would increase the complexity of the partitioning as it would now have to take into account the GUIDs (it would have to calculate a touches-based JFI that is more costly than the one for the number of operations in each region and would have to check which guids are in each one of the possible new regions at each iteration of this heuristic).

The *replicate-at-all* heuristic curve stays steady at 1 as update operations touch

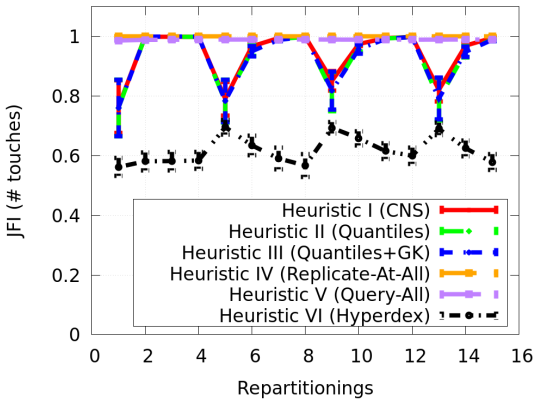


Figure 4.5: JFI (touches) X Repartitioning (RHO 0.5)

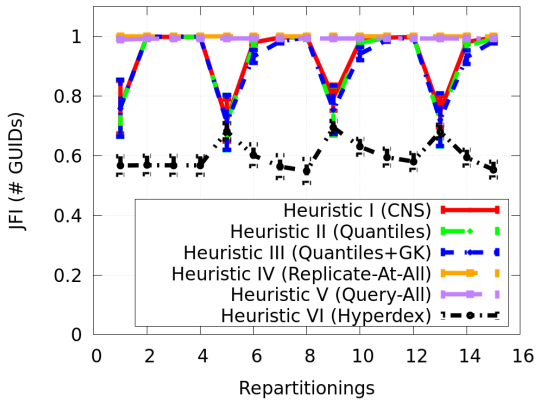


Figure 4.6: JFI (GUIDs) X Repartitioning (RHO 0.5)

all the machines and search operations are distributed uniformly among machines. The *query-all* heuristic curve also stays steady near 1 as update operations touches uniformly the machines (it implements consistent hashing) and search operations touch all the machines. In the end all the machines have roughly the same amount of touches for these two heuristics.

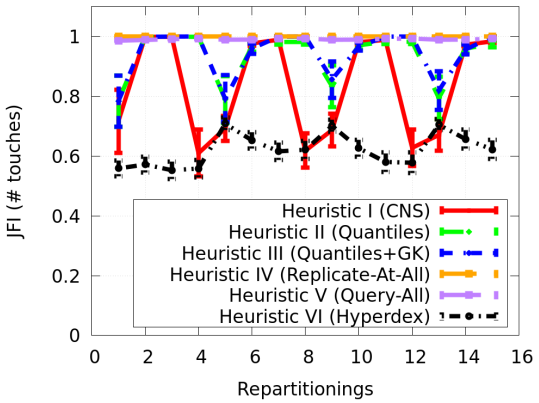


Figure 4.7: JFI (touches) X Repartitioning (RHO 0.75)

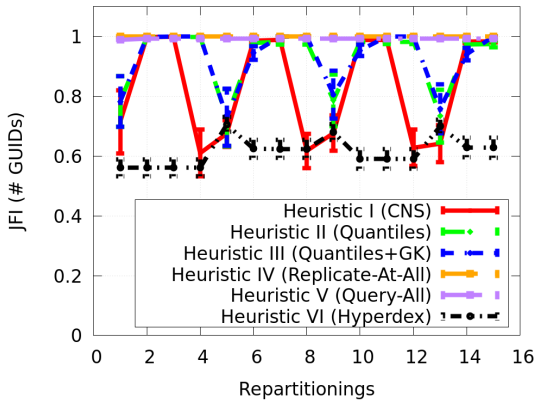


Figure 4.8: JFI (GUIDs) X Repartitioning (RHO 0.75)

Because the *HyperDex* heuristic is not a demand aware heuristic, it does not make sense to perform repartitioning based on the operations' distributions. Thus, it stays the whole experiment with its initial configuration of regions. It shows some improvement in its results for a time whenever the distributions change. This is because touches are usually concentrated in some regions. When there is a distribution change, touches are concentrated in other regions. Thus, it results in a temporary fairness in terms of touches per region. However, as more operations arrive, this unbalance reappears as touches keep concentrated in some regions because

this heuristic does not perform repartitioning.

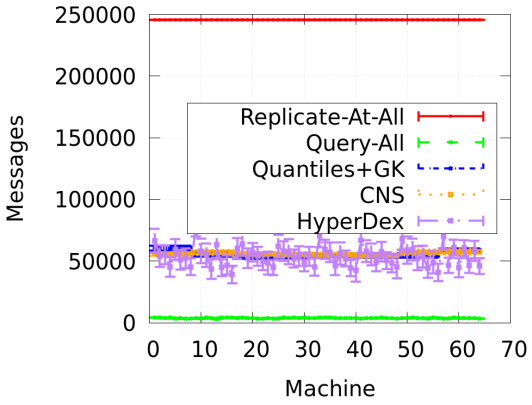


Figure 4.9: No. Messages X Machine (RHO 0)

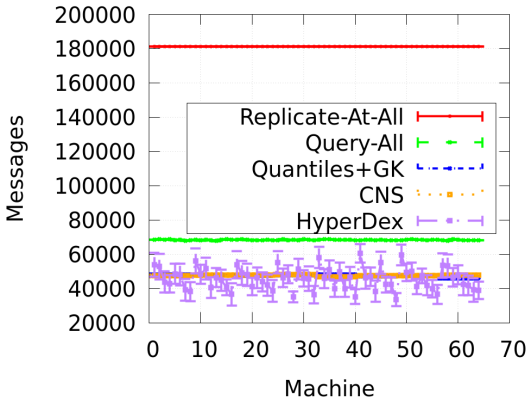


Figure 4.10: No. Messages X Machine (RHO 0.25)

We observed that the curves for GUIDs and for touches follow the same pattern for all heuristics. That is, the metrics *JFI of touches per region* and *JFI of GUIDs per region* both capture the notion of how well balanced the regions are. If we take for instance our heuristic, the idea of partitioning (and repartitioning) is to uniformize touches among regions, even though the operations' distributions are not uniform. In this sense, the tendency is that regions not only have a fair amount of touches, but also of GUIDs. Recall that GUIDs are distributed among regions by update operations and that the number of GUIDs in a region influence the number of touches in that region because of a search.

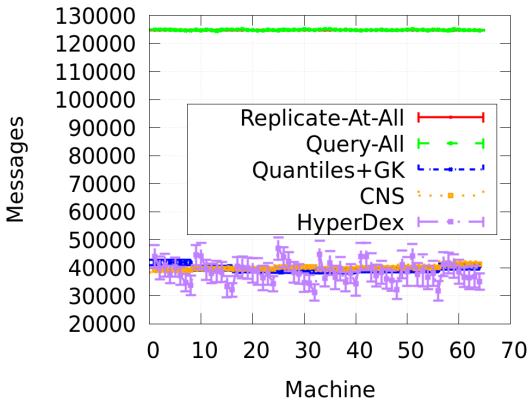


Figure 4.11: No. Messages X Machine (RHO 0.5)

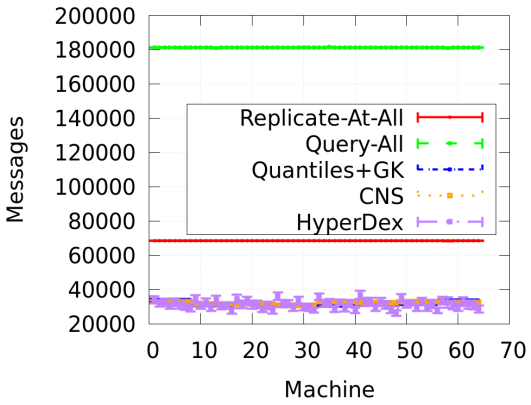


Figure 4.12: No. Messages X Machine (RHO 0.75)

The graphs 4.9, 4.10, 4.11 and 4.12 show the number of messages sent to each machine. The *replicate-at-all* heuristic imposes a high number of messages to be sent to its machines when $RHO = 0$ as every update operation must touch all the

machines. As *RHO* increases, this number of messages decreases as a search only needs to touch one of the machines. The *Query-all* heuristic will impose a higher number of messages to be sent to its machines as *RHO* increases. This is because a search must touch all the machines. *CNS* and our heuristic have almost a steady curve regarding the number of messages sent to its machines. This is because they partition in such a way regions are touched fairly (and thus, its machines as well). The *HyperDex* curve shows that some regions are much more touched than others. This is because it does not partition taking into account the operations' distributions in order to have a fair number of touches per region.

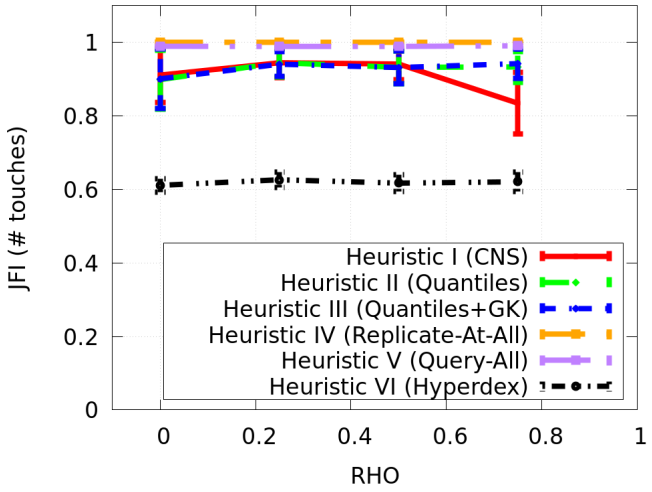


Figure 4.13: JFI (touches) X RHO

The graphs 4.13 and 4.14 show the average JFI of touches and GUIDs per RHO value. The demand aware heuristics (I, II and III) presented a good result with their curves at 0.9. The *query all* and *replicate all* heuristics presented the best results for these graphs (for the reason we explained earlier in this chapter) but as we will see in the next graph their results are not so good.

The graph 4.15 show the average number of messages sent to each machine per RHO value. The *replicate-at-all* heuristic initially presents a high average number of messages per machine but as RHO increases it rapidly decreases. We observe the opposite behavior with the *query-all* heuristic. As RHO increases, the average number of messages sent to each machine rapidly increases. It happens because when $RHO = 0$, i.e., when there are only update operations, messages are sent to all machines (for every update operation) for *replicate-at-all* heuristic and messages are sent uniformly to each machine (one message per update operation) for *query-all* heuristic. When *RHO* increases, the number of search operations also increases and the number of update decreases. Recall that a search operation imposes touches on only one machine for *replicate-at-all* heuristic and touches on every machine

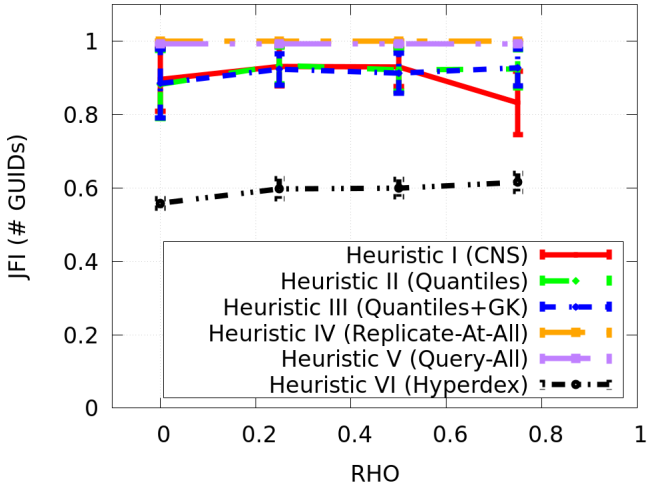


Figure 4.14: JFI (GUIDs) X RHO

for *query-all* heuristic. So, for *replicate-at-all* heuristic one message is sent to one machine and for *query-all* heuristic messages are sent to all machines.

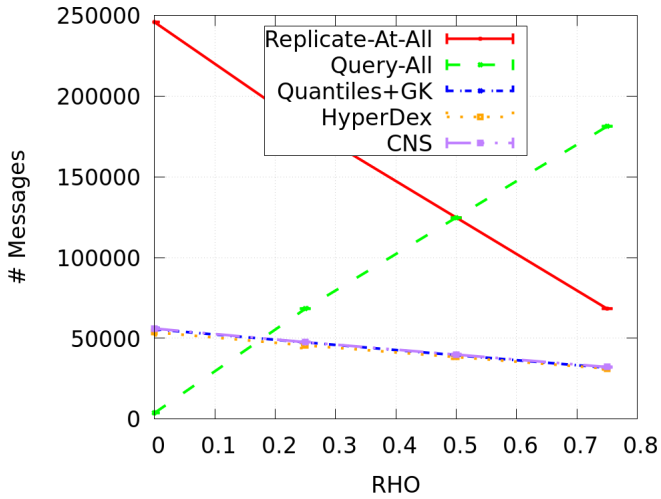


Figure 4.15: No. Messages X RHO

Our heuristic, *CNS* and *HyperDex* presented very close results. When $RHO = 0$ we only have updates, and update operations touch all the machines associated to each touched region. One update can touch up to 2 regions (the one where the content was stored and the new one where the content will now be stored). Which means each update operation can impose messages sent to all machines associated to two distinct regions. When RHO increases and we start dealing with searches (the number of updates are decreasing), the average number of messages has a slight fall as a message is sent to only one machine per region touched by that operation. For *HyperDex*, an update touches up to 2 regions per existing hyperspace. Thus, each update imposes the sending of 1 message to at most 2 machines per existing

hyperspace. A search touches up to all the regions of a single hyperspace. Thus, it imposes the sending of 1 message to at most one machine per touched region in a single hyperspace. In the case of this experiment, it means that an update imposes the sending of 1 message to at most 16 machines (same number for our heuristic, which sends 1 message to at most $2\sqrt{64}$ machines). A search imposes the sending of 1 message to at most 8 machines (same number for our heuristic, which sends 1 message to at most $\sqrt{64}$ machines). As we can see, this metric captures the amount of touches per machine.

Chapter 5

Conclusion

There are many solutions for achieving horizontal scalability in key-value data stores. We reviewed four solutions for this matter. *Replicate-at-all* replicates the entire key-value store at all machines. On the one hand, searches only need to touch one machine. On the other hand, updates need to be propagated to all machines. Moreover, as the key-value data store grows considerably in size, there is no other way but to apply vertical scaling. *Query-all* uses consistent hashing to uniquely and evenly map keys to machines. On the one hand, updates only need to touch one machine. On the other hand, searches need to touch all the machines. *HyperDex* uses subspace partitioning to create multiple lower-dimensional subspaces from a higher-dimensional space. A search never touches more than the amount of machines assigned to a subspace. An update touches up to two machines per existing subspace. However, this solution does not take into account the operations' distributions and, thus, some machines may be more touched than others. *CNS* use value-space partitioning to partition the value-space into regions. A search touches up to $\sqrt{\#machines}$ machines. An update touches up to $2 \cdot \sqrt{\#machines}$ machines. This solution takes into account the operations' distributions to create regions that are touched fairly. However, it does not consider the case that these distributions might change, which undermines its aim to create regions that are touched fairly.

In this sense, this work contributed in three ways. Firstly, we contributed by creating a demand-aware heuristic for value-space partitioning and repartitioning of a key-value store. It is a demand-aware heuristic because it partitions the value-space into mutually exclusive and exhaustive regions taking into account the operations' distributions so that regions are contacted fairly. Sencondly, we contributed by applying the Greenwald-Khanna algorithm (a data stream algorithm) in the partitioning process of our heuristic. It always keep an up-to-date summary for calculating the quantile points, which are the points at which we divide the value-space into regions. This approach enables a much less compute-intensive partitioning (as opposed to the CNS solution) and repartitioning. The fact that our heuristic is

designed to perform repartitioning (as opposed to other heuristics), enables us to create regions that are contacted fairly even though the operations' distributions change from time to time, while being much less compute intensive. Finally, our last contribution is the evaluation of our heuristic. We executed experiments varying the fraction of searches and updates, as well as their distributions, in order to evaluate the performance of our heuristic and compare it with other solutions like *replicate-at-all*, *query-all*, *HyperDex* and *CNS*.

The results show that our heuristic creates regions that are always contacted fairly, while not imposing a higher number of messages to be sent to the servers (because of update and search operations, and repartitioning messages) as the fraction of searches and updates varies, as well as their distributions. Our heuristic had superior performance than that of HyperDex in terms of fairness among machines, and similar performance than that of CNS without having to make use of its high compute-intensive way of partitioning. It also did not lose performance even having accepted a margin of error in the GK algorithm to create the summary. Moreover, the fact there are operations that do not define the attribute for which we partition the value-space, validates that our heuristic works well even in scenarios like this.

For a future work, one may want to extend this work by upgrading our heuristic to switch the partitioning of an attribute axis to another at any moment by creating, for instance, a GK summary in parallel that enables us to find quantile points on another attribute axis.

Bibliography

[1] ESCRIVA, R., WONG, B., SIRER, E. G. "HyperDex: a distributed, searchable key-value store". In: *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 25–36, Helsinki, Finland, August 13 - 17, 2012.

[2] VENKATARAMANI, A. "Scalable, Privacy-Preserving Contextual Communication", *Unpublished manuscript, College of Computer Science, University of Massachusetts Amherst.*

[3] KARGER, D., LEHMAN, E., LEIGHTON, T., et al. "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, El Paso, Texas, USA, May 04 - 06, 1997.

[4] SHARMA, A., TIE, X., UPPAL, H., et al. "A global name service for a highly mobile internetwork". In: *Proceedings of the 2014 ACM conference on SIGCOMM*, pp. 247–258, Chicago, Illinois, USA, August 17 - 22, 2014.

[5] VENKATARAMANI, A., KUROSE, J. F., RAYCHAUDHURI, D., et al. "MobilityFirst: a mobility-centric and trustworthy internet architecture", *ACM SIGCOMM Computer Communication Review*, v. 44, n. 3, pp. 74–80, July, 2014.

[6] JAIN, R., CHIU, D. M., WR, H. "A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems", *CoRR*, v. cs.NI/9809099, 01 1998.

[7] GREENWALD, M. B., KHANNA, S. "Space-efficient online computation of quantile summaries". In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pp. 58–66, Santa Barbara, California, USA, May 21-24, 2001.

[8] GREENWALD, M. B., KHANNA, S. "Quantiles and Equi-depth Histograms over Streams". In: *Garofalakis M., Gehrke J., Rastogi R. (eds) Data*

Stream Management, Data-Centric Systems and Applications, Springer, pp. 45–86, Heidelberg, Berlin, 2016.

[9] GREENWALD, M. B., KHANNA, S. "Power-conserving computation of order-statistics over sensor networks". In: *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 275–285, Paris, France, June 14 -16, 2004.

[10] ARASU, A., MANKU, G. S. "Approximate counts and quantiles over sliding windows". In: *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 286–296, Paris, France, June 14 - 16, 2004.

[11] VENKATARAMANI, A. "GigaPaxos: System Support for Group Scalability in Nano-Grained, Reconfigurable Geo-Replicated Systems", *Unpublished manuscript, College of Computer Science, University of Massachusetts Amherst*, 2016.

[12] VAN RENESSE, R., ALTINBUKEN, D. "Paxos Made Moderately Complex", *ACM Comput. Surv.*, v. 47, n. 3, pp. 42:1–42:36, fev. 2015. ISSN: 0360-0300. doi: 10.1145/2673577.

[13] LAMPORT, L. "Paxos Made Simple", *Sigact News - SIGACT*, v. 32, November, 2001.

[14] GAO, Z., VENKATARAMANI, A., KUROSE, J. F., et al. "Towards a quantitative comparison of location-independent network architectures". In: *Proceedings of the 2014 ACM conference on SIGCOMM*, pp. 259–270, Chicago, Illinois, USA, August 17 - 22, 2014.

[15] BENTLEY, J. L. "Multidimensional Binary Search Trees Used for Associative Searching", *Commun. ACM*, v. 18, n. 9, pp. 509–517, set. 1975. ISSN: 0001-0782. doi: 10.1145/361002.361007.

Appendix A

Algumas Demonstrações