Universidade do Minho
Escola de Engenharia

Sergio Miguel da Silva Carvalho

MonetDB in the Cloud
Data Distribution and Replication for a Cloud Environment

August 2010

Universidade do Minho
Escola de Engenharia

Sergio Miguel da Silva Carvalho

MonetDB in the Cloud
Data Distribution and Replication for a Cloud Environment

Master's Thesis
Master in Informatics Engineering

Supervisors:
Prof. Dr. Rui Carlos Oliveira
Departamento de Informatica, Universidade do Minho
Prof. Dr. Martin Kersten
Centrum Wiskunde & Informatica, Amsterdam
Dr. Fabian Groffen
Centrum Wiskunde & Informatica, Amsterdam

Agosto de 2010

É autorizada a reprodução parcial desta dissertação, apenas para efeitos de investigação, mediante declaração escrita do interessado, que a tal se compromete.

Universidade do Minho, ___ / ___ / ___

Assinatura: ____

Acknowledgements

This project would not have been possible without the people that surrounded me during the whole journey, to whom I would like to express my most sincere gratitudes.

To Dr. Rui Oliveira and Dr. Martin Kersten for giving me the opportunity to do my master thesis in such a wonderful city as Amsterdam and in such a distinguished place as CWI.

Special thanks to Dr. Fabian Groffen for guiding me through the whole process, for the patience discussing all topics and for being tireless reviewing this document.

To all people I met during my stay in CWI, in particular the INS-1 group, for making me feel one of them.

Ao meu pai, Fernando Carvalho, à minha mãe Adelaide Tomé e à minha irmã, Inês Carvalho, por todo o apoio, ajuda e compreensão que sempre demonstraram.

A toda a minha família, em especial aos meus primos, aos meus velhos amigos da Póvoa de Varzim e aos novos que fiz em Braga pelos bons momentos partilhados.

To all the friends I made in Amsterdam, for the companionship and fun moments that made this year so great.

Resumo

MonetDB na Nuvem:

Distribuição e Replicação de Dados para um Sistema de Nuvem

O conceito de computação em nuvem surgiu recentemente como uma forma conveniente e flexível de disponibilizar recursos computacionais. Na nuvem pode-se facilmente adquirir e libertar recursos, característica que poderá trazer muitos benefícios mas também introduz novos desafios. Para as bases de dados distribuídas o maior desafio será tentar tirar partido desta natureza dinâmica e adaptável da nuvem. Várias problemáticas podem ser tidas em conta relativamente a este tópico, no entanto neste trabalho iremo-nos focar principalmente na distribuição e redistribuição de dados, pretendendo eficiência aquando a adição ou remoção de máquinas virtuais.

As nuvens são suportada por sistemas de grandes dimensões, com compromissos e responsabilidades para com os clientes. Aqui tudo é pensado de modo a fornecer um serviço resiliente, onde se tenta prevenir eventuais falhas para evitar ao máximo consequentes interrupções do serviço. Contudo, mesmo a nuvem não é infalível e as máquinas virtuais aqui alojadas estão igualmente sujeitas a falhas. Caso isto aconteça estas máquinas serão certamente recuperadas e reinseridas na nuvem, no entanto não é garantido que o estado dos seus dados seja preservado. Deste modo, para além de distribuição será também abordada replicação, de forma a garantir maior segurança e disponibilidade dos dados e se necessário possibilitar um restauro do sistema.

Abstract

MonetDB in the Cloud:

Data Distribution and Replication for a Cloud Environment

Cloud computing has recently emerged as a convenient and flexible way to provide computational resources. In a cloud environment resources can easily be acquired and let go, which may bring many benefits but also introduces some new challenges. Here the main challenge for distributed databases is to be able to take advantage of the dynamic and easily adapting nature of the cloud. Many issues can be addressed in this topic, however this work mainly focuses on data distribution and redistribution, aiming at efficiency when adding or removing virtual machines.

Clouds are supported by systems of big dimensions, with commitments and responsibilities to their customers. Every service provider try to prevent failures at all costs to avoid consequent outages of the service. However, even the cloud is not bullet proof and virtual machines allocated in this environment may fail as well. While machines certainly will return into the cloud, their data state is not guaranteed to be preserved over failure. Therefore, this work considers data replication on top of data distribution, in order to provide improved data reliability and availability and allow, if necessary, a system restore.

Contents

1 Introduction **1**
1.1 Use Cases 2
1.1.1 SkyServer 2
1.1.2 Telecommunications Company 3
1.1.3 Sports Statistics Website 3
1.2 Project Objectives 4
1.3 Document Structure 4

2 Background **5**
2.1 Cloud Computing 5
2.1.1 The Different Types of Clouds 7
2.1.2 The Virtualization Role 8
2.2 Virtualization 8
2.2.1 The Different Types of Virtualization 9
2.2.2 Virtualization Products 11
2.3 Distributed Databases 13
2.3.1 Distributed Databases Management Systems 13
2.3.2 Map/Reduce 15
2.3.3 Data Distribution 15
2.4 MonetDB 16
2.4.1 Distributed Computation in MonetDB 16

3 Related Work **19**
3.1 Efficiently Extensible Mapping for Balanced Data Distribution 19
3.2 Linear Hashing 20
3.3 Consistent Hashing 21
3.4 Replication Under Scalable Hashing 21
3.5 Controlled Replication Under Scalable Hashing 22

3.6 Distributed Database Management Systems . . . 22

4 Data Distribution 25

4.1 Introduction . . . 25
4.2 Simple Algorithms for Data Distribution . . . 26
4.2.1 Division . . . 27
4.2.2 Hashing . . . 29
4.2.3 Range . . . 31
4.3 Revised Algorithms for Data Distribution . . . 32
4.3.1 Linear Hashing . . . 33
4.3.2 Linear Division . . . 35
4.3.3 Consistent Hashing . . . 37
4.4 General Overview . . . 39

5 Prototype 43

5.1 Implementation . . . 43
5.1.1 Data Set . . . 43
5.1.2 Technological Choices . . . 44
5.1.3 Algorithms . . . 44
5.2 Evaluation . . . 46
5.2.1 Loading the Data . . . 46
5.2.2 Adding a bucket . . . 48
5.2.3 Removing a bucket . . . 50

6 Data Replication 53

6.1 Introduction . . . 53
6.2 Two buckets: original and replica . . . 54
6.2.1 Adding a machine . . . 55
6.2.2 Removing a machine . . . 55
6.2.3 Considerations . . . 56
6.3 Three buckets: original, replica and replica fragment . . . 56
6.3.1 Adding a machine . . . 57
6.3.2 Removing a machine . . . 58
6.3.3 Fault Tolerance . . . 59
6.4 Completely Distributed . . . 60
6.4.1 Adding a machine . . . 60
6.4.2 Removing a machine . . . 61

6.4.3 Fault Tolerance 62
6.5 Original data and replicas in different machines 63

7 Conclusion **65**
7.1 Results and Overview 65
7.2 Future work 66

A Gentoo Configuration **69**

B Data Schema **71**

Bibliography **77**

List of Figures

2.1 Some of the cloud characteristics 6
2.2 x86 Privilege Rings 9

4.1 Loading data using the division algorithm 27
4.2 Adding a bucket to the system and redistributing the data with the division algorithm 28
4.3 Removing a bucket from the system and redistributing the data with the division algorithm 28
4.4 Load a set of data using the hash algorithm 29
4.5 Adding a bucket to the system and redistributing the data with the hash algorithm 30
4.6 Removing a bucket from the system and redistributing the data with the hash algorithm 30
4.7 Load a set of data using the range algorithm 31
4.8 Adding a bucket to the system and redistributing the data with the range algorithm 32
4.9 Adding a bucket to the system and redistributing the data with the linear hashing algorithm 33
4.10 Removing a bucket from the system and redistributing the data with the linear hashing algorithm 35
4.11 Adding a bucket to the system and redistributing the data with the linear division algorithm 36
4.12 Removing a bucket from the system and redistributing the data with the linear division algorithm 37
4.13 Loading data using the consistent hash algorithm 38
4.14 Adding a bucket to the system and redistributing the data with the consistent hashing algorithm 39

5.1 Load operation 47
5.2 Time elapsed adding a bucket 49
5.3 Records moved adding a bucket 49
5.4 Time elapsed removing a bucket 51
5.5 Records moved removing a bucket 51

6.1 Adding a machine using the n+1 approach for the replicas 55
6.2 Adding a machine with replication using the three buckets per machine approach 57
6.3 Removing a machine with replication using three buckets per machine in a setting with an odd number of machines 58
6.4 Adding a machine with replication using a completely distributed approach 61
6.5 Removing a machine with replication using a completely distributed approach 62

List of Tables

2.1 Virtualization Characteristics 11
2.2 Virtualization Products 12
2.3 Virtualization Products Characteristics 13

3.1 Comparison of efficiency 20

4.1 Data distribution algorithms comparison 40

6.1 Replication comparison in data distribution algorithm 63

Chapter 1

Introduction

Databases are used everywhere, providing data storage for a wide variety of businesses. They are used in supermarkets, hospitals, universities and many other places holding very different kinds of data. Over time more and more data is stored which makes the tasks of databases more and more time consuming. To improve the speed when accessing the data, databases are taking the direction of distribution.

Distributed computing has increasingly become a more important discipline, being often taken as the way forward when trying to achieve greater computational power, improved availability and better reliability. Therefore many Database Management Systems (DBMS) are approaching this paradigm as a mean to share workload and obtain better performance. Having distributed databases can be more difficult to implement and maintain but it enables faster answers to queries, taking advantage of more computer resources at the same time.

The variety of distributed systems that can be addressed is vast. It can go from just a couple of connected machines, to a cluster, a grid and more recently a cloud. Cloud computing shows up as a very trendy distributed system of our days. The popularity of this concept has resulted in multiple implementations with different characteristics. The idea of a cloud stands out mainly for its ability to provide a dynamic service that flexibly adapts to clients needs, making it easy to allocate and deallocate resources at anytime. With this in mind, an approach to a cloud system has to take into account this possible volatility of the system, being prepared to efficiently adapt to new configurations.

In this work we aim to deploy a distributed database in the cloud, considering more particularly the issue of data distribution.

1.1 Use Cases

Distributed computing should be applied wisely. A frequent problem is that having too many machines may deteriorate performance, but the optimal number of machines is in the most cases not known in advance or it evolves during time. A system able to grow and shrink efficiently would bring much benefits to this paradigm. Cloud computing is the environment we are considering. It makes possible to purchase computational resources in a flexible way, which can also be very advantageous in many scenarios involving distributed databases.

In this thesis we approach distributed databases in the cloud, taking particular interest in distribution and redistribution of the data throughout the various virtual machines. We believe that faster query processing can be achieved if data is more distributed, as by using more machines the search scope is reduced. However we are not going deeper in how the data is going to be queried as we see it as another big issue in distributed databases.

The types of databases we are addressing in this project are static databases. Here no updates or transactions are being performed during query time, which eliminates the need to concern about data consistency issues. This kind of databases are more common than one might imagine and they represent a very important issue as there are many cases where the amounts of data are significantly big.

1.1.1 SkyServer

The Skyserver [19, 8] project is a challenging real-life demanding application, that works over a scientific data warehouse. Skyserver provides public access to the data collected by the Sloan Digital Sky Survey (SDSS), an astronomic survey that ambitiously aims to create a digital map of a large part of the universe.

A project of this dimension leads inevitably to large amounts of data that need efficient data management and high performance for data mining applications, in order to extract qualitatively rich knowledge. Every night the SkyServer telescope produces about 200GB. However, this data is not released daily, instead the available data is only updated once every year. In the meanwhile scientists and enthusiasts want to be able to query this huge data set in the fastest possible way. This stands as a motivation to develop new algorithms and techniques addressing data management challenges in large scale scientific databases. Distribution is one of those challenges, as it can enable faster query processing.

Imagine we are providing a service that allows the users to efficiently query the

SkyServer data. It would be natural that this service would be more overloaded in the days following the release of a new data set than in the rest of the year. During this period, if our system was deployed in a cloud, we could easily grow to a bigger system and prevent performance degradation.

1.1.2 Telecommunications Company

Another scenario involving big amounts of data can be found in telecommunications companies where lots of data is generated each day, regarding all the information related to clients operations (calls, video calls, text messages, multimedia messages). However there is not much interest in examining this data every day to see what has changed.

For example, typically one wants to know how many calls were made by each client in the end of each month, in order to calculate every phone bill. If using a distributed database working over a cloud, more computational resources can easily be purchased in the beginning of the month, in order for all the bills to be processed in proper time. After this task has been executed, all the extra machines that were requested can be removed again, as there is no need to support superfluous expenses.

1.1.3 Sports Statistics Website

Here we are thinking on a website that provides football statistics which is very likely to be more overloaded before an important football match. Let's take as example a classic match between Real Madrid and Barcelona for the Spanish Championship. Millions of people around the world are watching the match and following all the related news before and after it.

Journalists consult the statistics trying to find interesting historical facts to make a publication (like who won more times the classic). Many football fans consult the statistics to be aware of how both teams have been performing over the season. Bettors look at the statistics in order to try to make the best prognostic.

This kind of queries are going to overload the system probably only few hours before the match starts. If the system is in the cloud, we can easily request more virtual machines for this period of time and distribute the whole system through the added computational resources. This would allow to cope with all the requests without penalizing much the performance. This situation may apply in various scenarios where a service suffers specific workload picks caused by an event that can be seen in advance.

1.2 Project Objectives

In this work we focus on some aspects that have to be considered when dealing with a distributed database in the cloud. We focus on the distributed database design, primarily in what concerns the data distribution and data replication. Since the cloud is known for its inherent elasticity there is the need to infer about what should happen to data when virtual machines are turned off or new ones are added to the system. In these scenarios, we want data to be efficiently distributed over all the available machines and in the most balanced way possible. In this thesis we address and compare some algorithms to distribute data in order to understand what is the best approach and what commitments it implies. The main goals of this work are:

- Study the efficiency of each different approach to data distribution;
- Implement and evaluate different techniques in order compare and understand them better;
- Introduce replication to the studied algorithms and infer on the implications.

1.3 Document Structure

In Chapter 2 we will give some insight on the background required to understand the context that surounds the subject being approached. We will briefly talk about some important topics as Cloud Computing, Virtualization and Distributed Database. Then in Chapter 3 we will discuss data distribution looking into some related work that has been done in the field.

In Chapter 4 we will approach some algorithms to distribute data, giving some insight on their behavior and clarifying what happens to the data in case a machine is added or removed from the system. In Chapter 5 we will give some details on the implementation of the presented algorithms in a prototype written in Java. This program will then be evaluated in order to allow a better comparison between the efficiency of the different techniques.

Later, in Chapter 6, we bring a discussion on different ways to introduce replication in a system of such peculiar characteristics, where the distribution of data have to be ready to adapt to changes in the system.

Finally in Chapter 7 the paper will be concluded with a look on what was done and achieve. In this chapter we will also suggest some possible future work.

Chapter 2

Background

This chapter intends to give some insight on important concepts that represent the basis of our work. Here we will give an overview on the characteristics of Cloud Computing. We will see what has been done in virtualization of machines and what is the relation with the cloud. Then we will present the MonetDB project and the paths that are being taken into distributed systems. Finally we will present distributed databases and some topics around it.

2.1 Cloud Computing

Cloud computing brings up a convenient new way to purchase resources, being already playing an important role in the technological world of our days. However, this is not a completely consensual concept, embracing sometimes many different characteristics depending on who is giving the definition. From the literature there are some interesting statements that can help an approach to the subject. In [2] cloud computing is seen as what refers "to both the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those services". On the other hand in [20] a cloud is described as "an elastic execution environment of resources involving multiple stakeholders and providing a metered service at multiple granularities for a specified level of quality of service".

Moreover, taking a look into published studies on the field[2, 20, 15] some other characteristics can be pointed out as we show in Figure 2.1. An important feauture that immediately comes to mind is elasticity. In Cloud Computing the illusion is given that resources are infinite and available on demand, being the client always able to subscribe and unsubscribe computational power, according to its needs. This resource elasticity is evidently also allied with economical elasticity, meaning that

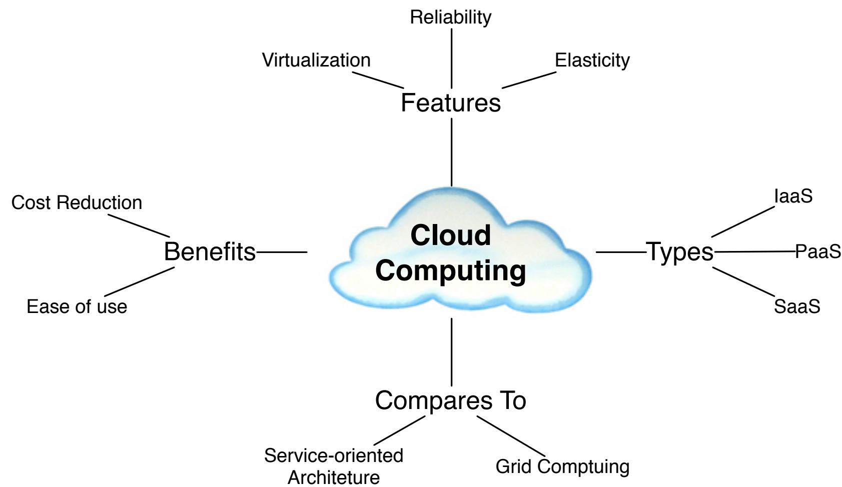

Figure 2.1: Some of the cloud characteristics

the client only pays for what he is using, allowing to comfortably adjust the budget to what suits him better.

Another relevant property is the no need for an up-front commitment, which can greatly enable start-up companies, making it easy to power up a new project with less concerns on how things are going to evolve. If the project is successful and grows, one can quickly increase the resources used, if it does not succeed the investment was not as big as if we had bought all the required hardware and software. It can also be an added value for already established companies which have to deal with punctual workloads or need to work faster to complete a job before a deadline (e.g. the price of purchasing 1 server for 1000 hours is the same as purchasing 1000 servers for 1 hour).

When talking about cloud computing, grid computing often comes to mind as a previous concept that shares basic similarities like the service oriented approach and the same target users. However cloud computing brings up two important differences comparing to what was being done in the grid[15]. While in the grid a user request could consume large portions of the available resources, in the cloud a request is often limited to a small fraction of the total pool, with the goal to support a large number of users. The other main difference concerns the way resources are put together and administrated. The grid is managed by different administrative domains while in the cloud there is usually a single administrative authority.

We have seen that cloud computing carries many advantages as flexibility and utility, however in a world where more and more the companies asset is in the data, many may be skeptic on going into this kind of environment. A reason frequently

pointed out is the fact that when working in a cloud, one does not know where the information is kept and who can have access to it. Also the quality of the Service Level Agreements (SLA), which have been improving in the last years, may still not be enough to some more demanding users. This aspects can lead several companies to opt for a private physical infrastructure, where they can be totally in control.

2.1.1 The Different Types of Clouds

Moreover there can be distinguished different types of cloud computation[2, 20]. Depending on the abstraction level provided to the user, we can define three types of clouds:

- **Infrastructure as a Service (IaaS):** lowest abstraction level where the resources provided are measured as physical hardware. The user can purchase for instances the desired processing power and available memory size.
- **Platform as a Service (PaaS):** here we can deploy applications built with a language supported by the cloud as long as it respects the given API. The price to pay may be defined by the number of hours of computation used, storage size or data transfers.
- **Software as a Service (SaaS):** the highest level of abstraction provides the user implementations of well defined business functions. Here the user is not so much in touch with the explicit features of a cloud but more with the application that is working on it.

Nowadays there are many companies providing cloud computing services in all the different levels of abstraction mentioned above. One of the biggest is Amazon, with the Elastic Compute Cloud (EC2), an IaaS, which offers low level instances, available with different OS, being the user able to choose the one that suites him better. Concerning resources, here there are three main kinds of instances: Standard, High-Memory and High-CPU. Each of those provides a different configuration, useful when planing on running a Memory or CPU intensive application. The underlying virtualization platform used in EC2 is Xen.

In the infrastructure level, other two well known cloud providers are GoGrid and Rackspace. In the open source world, there is also being developed an application that makes it possible to create a cloud infrastructure over computer clusters. Its

name is Eucalyptus and it allows full management of virtual instances, implementing the Amazon EC2 interface and for now supporting KVM and Xen virtual images.

Going to an intermediate abstraction level we have Azure from Microsoft, a cloud service that supports applications which use the .NET framework and are compiled for the Common Language Runtime (CLR). There are currently two available SDK's for the Azure Services Platform: Java SDK for .NET Services and the Ruby SDK for .NET Services.

In the highest end of the abstraction spectrum there are the application domain-specific platforms. Here we find for instance the Google Docs services. They provide a specific application that works over a cloud environment.

2.1.2 The Virtualization Role

Virtualization is intrinsically related with Cloud Computing, as it brings very useful features to a service of such peculiar characteristics. With virtualization comes flexible management of virtual machines over the hardware resources, making it possible for the system to quickly scale. It can also be seen as a powerful tool when dealing with the debug of a large scale distributed-systems, by providing division and isolation.

On the other side virtualization introduces overhead which is reflected in degradation of the performance. In some cases some isolation issues can also be noticed, where different virtual machines running on the same resources may interfere with each other. We have a tradeoff between convenience and performance, where commodity is clearly winning as work is continuously being done in order to improve virtualized systems.

2.2 Virtualization

Virtualization is a technique to provide abstraction over computer resources, allowing the coexistence of more than one operative system (OS) over the same set of hardware. This way, one can easily increase and decrease the number of running virtual machines in a cheap and convenient way. Contrarily to what some might think, this is not a new concept, having been brought up by IBM more than 30 years ago. However, the performance penalties attached, made it only now a viable solution, due to new implementation techniques and faster underlying hardware.

Although virtualization may somewhat penalize performance, it is the easiest

way to manage computational resources. The appliance of an abstraction level allows different tenants to use the same set of hardware for different intents. This is possible with the use of virtual machines that can be created, destroyed and even relocated to adapt to load and performance requirements. This flexibility shows up to be precious in an environment like Cloud Computing, where there are multiple users, working simultaneously and with varying amount of resources.

There is many virtualization software available, enabled by many different techniques and implementations. When talking about virtualization for cloud computing, some important characteristics to consider are the possibility to support different operating systems, the isolation between virtual machines, the overhead introduced by the virtualization and scalability[3].

We are looking into virtualization for the x86 architecture as it represents a considerable share of the existing hardware, having big part of virtualization studies and implementations been focused on it. In this architecture four levels of privileges are defined, commonly represented as Ring 0,1,2 and 3 as we illustrate in Figure 2.2. 0 represents the most privileged level, nearer the hardware, and 3 the less privileged one. This way, the operative system runs its instructions on Ring 0, as it needs direct access to memory and hardware, while applications execute on Ring 3[16, 22].

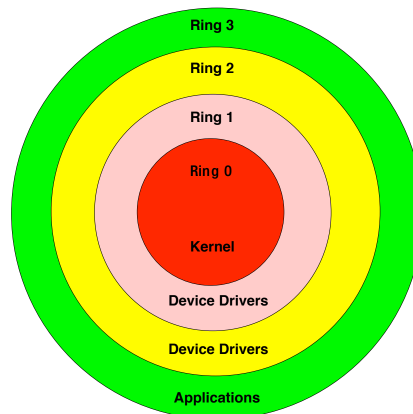

Figure 2.2: x86 Privilege Rings

2.2.1 The Different Types of Virtualization

Concerning virtualization we can distinguish three main different techniques: full virtualization, paravirtualization and operating system-level virtualization. The first two types of virtualization rely on a more or less complex layer which is responsible to intermediate the virtualized OS and the hardware resources. This layer is of-

ten called the Hypervisor or Virtual Machine Manager (VMM). The privilege rings where the hypervisor and the guest OS run vary depending on the kind of virtualization and the software in question. The operating system-level virtualization does not rely on a hypervisor, being the kernel responsible for isolating multiple user-space instances in the same machine. On Table 2.1 we can find a brief comparison between this three main types of virtualization.

Full Virtualization[10, 6, 16, 22, 14] provides an abstraction over the resources, allowing to run the guest OS without any modification, being able to virtualize any proprietary OS. Here the hypervisor tend to be more complex as it has to provide for each Virtual Machine all the services of the physical system. However, this is the easiest way to move and migrate a guest OS as the same instance can run virtualized or on native hardware. VMware virtualization products, VirtualBox and Parallels are some examples of software implementing full virtualization.

On the other hand the *Paravirtualization*[6, 3, 16, 22, 14] technique implies that the guest OS is aware of its virtualization, so it can cooperate with the hypervisor in a faster and more efficient way. In this kind of virtualization the devices are not emulated. They are accessed by lightweight virtual drivers that offer a better performance, however this demands for the guest OS kernel to be modified in order to work with the new system calls for the new services. Xen is a good example where paravirtualization is available.

The last kind of virtualization is the *Operating system-level virtualization*[23], where no hypervisor is used. Instead the kernel of the operating system is modified in order to be able to isolate multiple user-space instances within one host machine. As some performance benefits can be achieved from not having the hypervisor overhead, many drawbacks come along with this kind of virtualization. All the instances share the same kernel, so in case it fails, everything crashes. Furthermore this is not a very flexible virtualization technique as every instance has to run the same kernel. These limitations take out some important virtualization characteristics like flexibility and independence between different instances.

Recently, a new method for implementing full virtualization was also introduced, relying on specific virtualization systems calls provided by the hardware. It is named *Hardware-Assisted Virtualization*, and supported by the most recent x86 and x86-64 processors from Intel (Intel VT) and AMD (AMD-V). This is expected to be the fastest full virtualization implementation and can be achieved with a relatively simple hypervisor as the main services are provided by the operative system that is hosting the virtualization. One example of this virtualization technique is Kernel-

based Virtual Machine (KVM).

The last approach being made on OS virtualization is *Hybrid Virtualization*[16, 14], which means putting together hardware-assisted full virtualization and paravirtualization in order to take the best of each. Hardware-assisted virtualization has a reduced virtualization overhead, since it uses hardware instructions, but when it comes to deal with I/O or memory intensive applications the emulated virtual devices are slower than the paravirtualized ones. What is proposed is to combine some specific paravirtualized drivers, but still use full virtualization for the processor. The results obtained in [16, 14] are quite encouraging, leading to think this could be the way to go further.

	Full Virtualization	**Paravirtualization**	**OS level Virt.**
Multiple OS	Yes	Yes	No
Kernel Modified	No	Yes	Yes
Hypervisor	Yes	Yes	No
Independent crashes	Yes	Yes	No

Table 2.1: Virtualization Characteristics

2.2.2 Virtualization Products

Having been presented the different virtualization techniques available, there is the need to inspect which software can achieve better results taking into account some important parameters like different operating system support, performance overhead and scalability. In Table 2.2 we can see which products address mostly each kind of virtualization.

First of all looking into OS support, the full virtualization technique is able to run every OS as there is no need to modify the guest. This way VMware, Virtual Box, Parallels and KVM can theoretically support all x86 OS. In turn Xen, can only provide Paravirtualization to OS that are modified to do so, which makes it impossible to cope with proprietary OS. To overcome this issue, Xen also offers a full virtualization mechanism based on hardware assistance which allows running Windows XP or Windows Server instances. Operating system-level virtualization is not considered further in our study as it can only virtualize all the instances with the same kernel, not providing flexibility nor independence, characteristics indispensable in a Cloud environment.

Full Virtualization	**Paravirtualization**	**OS level Virtualization**
VirtualBox	Xen	LinuxVServer
VMware Server	VMware ESX	OpenVZ
VMware ESX		WPARs
KVM		Solaris Zones
Xen		

Table 2.2: Virtualization Products

Each software has it own way to approach virtualization, using different techniques and providing different features. There are many comparison studies documented concerning mostly open source technology like Xen and KVM. Probably due to license constraints, we found it harder to discover tests involving products from the VMware family, specially in what concerns the VMware ESX hypervisor.

Searching for comparisons on virtualization we understand that Xen had been the one more widely explored, maybe for its open source character or for being more mature. Meanwhile we also have VMware Server that became a free application in 2006, VMware ESX a proprietary product suited for datacenters and KVM which is part of the linux kernel, only been released in 2007.

Looking into [23], we can see a comparison for High Performance Computing including Xen and Vmware Server. Here are performed tests addressing network utilization, SMP performance, filesystem performance and MPI scalability. The outcome shows that Xen overcomes VMware Server in all the tested points, except in filesystem performance where it was not possible to test VMware server due to some disk incompatibilities.

Another performance study[21], this time for the VMware ESX hypervisor and the Xen hypervisor, dates from 2007, being the only comparison study available which includes the VMware ESX hypervisor. Besides, this paper was written by VMware, the constructor of one of the tested applications, and it used as guest OS the Microsoft Windows Server 2003. Because this is one proprietary OS, Xen will have to be performing full virtualization using the hardware-assisted mechanism, which do not achieve as good results as Xen's paravirtualization, especially when it comes to network throughput[16, 21]. Even though the benchmark outcomes were not so dissimilar with the expected exception of the network throughput test, where Xen was significantly worse than VMware ESX.

Finally in [5], we can see a comparison test between Xen and KVM, where

isolation, performance and scalability are addressed. Concerning to the overall performance, Xen showed up to be slightly faster than KVM in CPU-intensive testing and in a kernel compilation, but KVM performed better in writing and reading. The performance degradation in the virtualized systems comparing to the native one was not so significant except in the kernel compilation where virtualization was slower more than 50%. In what touches to isolation Xen behaved well, showing good isolation for the memory, fork, CPU and disk stress tests. It only pointed out isolation problems on the network sender and receiver tests. KVM did better, showing good isolation properties for all the stress tests. Finally scalability was also tested and Xen proved to be a more mature software scaling linearly as the number of guests increased. KVM did not scale so well, presenting some guest crashes, not being able to maintain the performance as the number of guests increased.

In table 2.3 we can see a summary of the most significant aspects:

Software	Performance	Scalability	License
VMware Server	bad	bad	free but not open source
VMware ESX	good	good	proprietary software
KVM	fairly good	some problems	open source
Xen	good	good	open source

Table 2.3: Virtualization Products Characteristics

All operative systems can be virtualized, if not with paravirtualization, with full virtualization. To take the most advantage of the resources provided by the virtual machines we tried to make a personalized instance. We can minimize wastes by creating and tuning a virtual machine that uses little resources on the operative system and puts all its power in the principal task we want to perform. This comes at the cost of investments on how to keep the operating system minimal and yet operating sufficiently for the application in use. An attempt to reduce an operative system is shown in Appendix A with a Gentoo distribution.

2.3 Distributed Databases

2.3.1 Distributed Databases Management Systems

As defined in [13] a distributed database (DDB) is a collection of multiple logically interrelated databases distributed over a computer network. The software that man-

ages the DDB is named the distributed database management system (DDBMS) and it provides an access mechanism that makes this distribution transparent to users.

This transparency is taken as one of the main goals in a DDMBS and it can be achieved at many levels: network, replication, fragmentation. Network transparency consists on providing an abstraction over the network so that the user is not aware of the location and quantity of machines that are connected. Replication transparency is the ability to manage replicas of the data automatically in various places, without interfering with the normal functioning of the system. Finally fragmentation transparency is about being able to automatically fragment and distribute the data over the various constituents of the system.

Usually we expect that a DDBMS can provide improved reliability, availability and performance when compared to a monolithic system. An example of more reliability and availability can be seen in a scenario where if one component fails, it does not necessarily mean that the whole system will fail. If there was a data replication mechanism implemented, everything may still work just fine by relying on the parts of the system that are still in good health. Also distribution allows the exploitation of parallelism, which can lead to better performance when doing operations over the data. Furthermore, because this are systems compound by many machines, they are easier to expand than a regular DBMS, all we need to do is add new nodes and redistribute the data.

Implementation Issues

Having talked on what a DDBMS intends to achieve, it is time to present some of the main issues concerning its implementation: DDB design, query processing, concurrency control and reliability. Each of those topics can be approached in different ways with different outcomes to the system performance. This work is mostly focused on the first issue, where we study how to fragment and distribute data.

Data distribution is usually desirable because it enables the placement of data in close proximity to its place of use, thus potentially reducing transmission cost. However because we are looking into deploying it in a cloud environment this issue will not be taken into account as in a cloud there is no information about the localization of each machine. Thus partitioning the data will mostly be used to allow the exploitation of parallelism and for reducing the size of relations that are involved in user queries.

2.3.2 Map/Reduce

Nowadays there are ways to query data that do not require to keep track of each record's location. We are talking about an approach that inquire all the nodes in the system and comes up with a single answer. One technique able to provide this kind of feature is Map/Reduce [7], a very simple algorithm revived by Google that allows to distribute the workload throughout all the system.

The Map/Reduce algorithm is compound by two fundamental steps the Map and the Reduce. The Map consists on having a head machine that receives the requested operation and distributes it through all the available machines in the system - the workers. Each worker will calculate its own answer using its data and if necessary by interchanging of intermediate results with the other workers. The second and final step, the Reduce, consists on the head node getting all the results from each worker and calculating the final answer.

In this query technique there is no need to know where each part of the data is located, as every participant in the system is queried. This allows to look at data distribution on distributed databases from a whole different perspective.

2.3.3 Data Distribution

One fundamental step into database distribution concerns the way data is spread among the various nodes, characteristic that will greatly influence the whole behavior and performance of the system. Here some issues pop up: in which way and how much should data be fragmented, how should it be allocated and how much information should be kept about its location. Having this in mind and also thinking about integration with cloud computing, some algorithms may be addressed in order not only to allow the distribution of data but also to grant adaptability facing the introduction of new machines and the removal of existing ones.

When fragmenting a database the chosen unit of partitioning can go from whole relations to tuples or attributes. However considering an entire relation as the atom may not be the best option, as relations grow in different ways making it hard to balance the data. Furthermore it will limit exploitation of the parallelism gained with the distribution. Thus the partitioning should either be done in a vertical way by taking the attributes as the distribution unit, horizontally taking the records as the distribution unit or in a hybrid way where we have both vertical and horizontal distribution.

2.4 MonetDB

MonetDB is an open source column-oriented database management system[1] able to achieve high performances dealing with complex queries on big amounts of data. There are some important characteristics that distinguish this DBMS from other existent software. Two important features worth to be referred are the fact that it uses a column-wise storage organization and implements full materialization of intermediate results. The core of MonetDB is present in mserver which is responsible for all the mechanics behind the database operations. There is also a crucial program called Merovingian, which integrates MonetDB enabling it to perform the two main following tasks: handle connections from clients and redirect those connections to the appropriate mserver process; start an mserver process, which may be very useful in case of a crash.

Resuming into a practical scenario, we can have several different machines, each running a Merovingian instance which is responsible for 0 or more mservers. A client can make a request to any database in any machine in the network that Merovingian will take care of guaranteeing that the database in question is reached. Also if one of the mservers crashes, Merovingian is able to relaunch it. Going deeper into Merovingian, we can point out some more interesting details. Merovingian uses UDP connections through the default port 50000, to make the communication between different instances, periodically broadcasting messages in order to know who is constituting the network at each time. When a request is made for a database that is in another machine, Merovingian can proceed in two ways: as a proxy intermediating the communication between the client and the other Merovingian, useful in networks where the client does not have direct access to every machine; or it can redirect the client to the correct machine. Merovingian also supports a very handy feature that enables the tagging of databases. This way we can tag different databases in different machines with the same name, and then use the command *find* when wanting to retrieve them back.

2.4.1 Distributed Computation in MonetDB

As it was mentioned before there are some projects being developed on CWI that concern MonetDB and distributed systems. For the moment three different approaches are being taken:

- **Octopus:** Trying to achieve even greater performance there was the need

for MonetDB to take advantage of parallelism. Thus, work has been done to develop a new module called Octopus which intends to implement distributed query processing, based on some of Merovingian's capabilities. Here there is a defined head node responsible for holding all the data that is going to be processed. It communicates with the client receiving the requests and also with the other nodes delegating work. The workers are called tentacles and they start by being inquired about their state and the cost of processing a specific query in a determined set of data. The cost may be measured in different ways, according to what is intended to achieve: perform the least computation; transfer the least data; do it in the shortest time. After the head of the Octopus have chosen the nodes where the query is going to be executed, the data is shipped to each node respectively. However there is a recycler in each node, in charge of keeping some of the data, after a job has been performed. By doing this, significant improvements in the query execution can be obtained, as there is the possibility to reuse data and results that already are in the nodes, avoiding the work intensive task of transferring and processing it again.

- **Cyclotron:** Basically consists on having the database distributed by many computers but stored in their main memory. These sets of data that constitute the database are passed around from machine to machine through the network. Whenever a query is fired in a machine, it just has to wait until all the necessary data passes by. This concept is supported by the idea that networks are getting faster and faster and that it may be quicker to get data from the network than from a local hard drive.

- **Map/Reduce:** This project is an implementation of Google's Map/Reduce algorithm, which consists on spreading the computation through all the machines in the system and after everything has been processed gather all the different results and make one final answer. Here the data is already spread among the various machines and the only thing to do is fire the queries to all of them and in the end build a final result with all the outputs. An example of this is a query that asks for a maximum value of an attribute, which implies sending the same query to all the nodes. After each machine has done its processing, it sends back a result with the value of its maximum. Then, the head machine of the process, gather all the answers achieved and gives the user the highest result.

Chapter 3

Related Work

To take a distributed database into the cloud, data must adapt to system changes. Efficient data reorganization is an important topic concerning any distributed system, but it becomes even more important when the environment to address is the cloud. In this chapter we present and discuss some previous work that has been done on data distribution algorithms.

3.1 Efficiently Extensible Mapping for Balanced Data Distribution

In their paper [4], Choy, *et al.* discuss three different ways of addressing buckets (abstract representation that holds records) to bins (disks or servers): using a Round Robin assignment; keeping the administration of the mapping buckets-to-bins; and a new method introduced by this paper called Interval-Round-Robin. They are considering buckets continuously identified and their study is focusing on how the system is able to support database growth, i.e., to accommodate the addition of new bins.

In the Round Robin assignment we are always able to know the position of each bucket, as the buckets identifiers are continuos. The number of the bin can be calculated by just diving the bucket id by the total number of bins. With this technique every time a bin is added to the system all the buckets have to be reshuffled. This requires a lot of records to be moved upon a reorganization.

The mapping solution consists on maintaining a mapping of every bucket to a bin. This is better in terms of data reorganization, as if the system changes we are able to adapt only by moving the minimum data possible. However this implies

keeping more storage data (metadata).

Finally the new algorithm introduced in this paper is the Interval-Round-Robin. What they try to achieve with it, is an intermediate algorithm that does not need to move so much data as the first one and does not need to maintain as much metadata as the second one. This technique works by keeping track of the changes that occur in the system. There are blocks of buckets representing each time the system has changed. It also minimizes the amount of data to be moved in a reorganization.

A comparison between the algorithm's efficiency is represented in Table 3.1.

	Mapping Complexity	Mapping Storage	Records Relocation
Round-Robin	1	1	Large
Complete Mapping	1	R	Minimum
Interval-Round-Robin	O(log m)	$O(m^2)$	Minimum

Table 3.1: Comparison of efficiency

Concerning the mapping complexity we can see that the Interval-Round-Robin algorithm is the most complex one. It presents a complexity of O(log m) where m is the number of changes that occurred in the system. This happens because each bucket is dependent on the history of the system.

All the algorithms need to maintain some metadata. The Round Robin, keeps the least, just maintaining the information on the total number of bins in the system. The Stored-Directory keeps a tag on every bucket in order to map it to the respective bin. The Interval-Round-Robin keeps track of all the changes that occurred in the system during time.

The Interval-Round-Robin is also very efficient, being able to relocate the buckets with minimum cost. The amount of data kept with this algorithm increases as more changes in the system are being made.

However this new algorithm comes with a big handicap, which is the fact that it can just cope with a system that only grows. This means it cannot be adapted to the cloud environment, as there the system is able not only to grow but also to shrink.

3.2 Linear Hashing

This is a hash-based algorithm able to overcome the main handicap of normal hashing, adapting when the address space of the hash function changes. Linear

Hashing performs better when changes occur, only needing to recalculate a specific set of keys instead of everything. It was introduced in 1980 in the article [11] as an innovative way to address data, where the address space can dynamically grow and shrink.

More recently in [12], a generalization of this algorithm for distributed systems is presented, which is called LH*. It introduces an efficient way to implement Linear Hashing in a client server model.

The drawback of this algorithm when distributing data is the fact that data is not balanced at all times, not allowing for an optimal use of the disk space. It also does not support weighting, so all the buckets have to be treated equally.

3.3 Consistent Hashing

Consistent Hashing [9] introduces a hash based distributed algorithm that can easily adapt to changes in the address space. Contrary to what happens in LH* [12], this algorithm does not work linearly and buckets may be added or removed in an arbitrary order. Furthermore, it is also able to perform without the need for every client to have a consistent view of the system, which allows to reduce the number of messages going through the network.

A study on the implementation of this algorithm in a distributed cache over a network is presented in [9], where sometimes there is the need to scale in order to overcome higher demands. Hash functions are here constructed concerning three consistency properties: smoothness, spread and load. The smoothness property ensures that if a bucket is added or removed, only the necessary records to keep the system balanced are moved. The second property, the spread, says that over all the clients views the total number of caches to where an object can be assigned is small. Finally in the load property we have that over all the clients views the number of distinct objects assigned to a particular cache is also small.

3.4 Replication Under Scalable Hashing

This [17] is a family of decentralized algorithms allowing to map objects to a scalable collection of servers or disks. Changes in the system are handled very well and the data is reorganized just by moving the minimum objects possible.

These algorithms provide a very important feature concerning distributed systems, data replication. Here they provide an adjustable technique where different

degrees of replication may be addressed, with the guarantee that replicas of the same object are not placed in the same machine. The RUSH algorithm is also able to provide weighting, which means that if we have machines with different resources, different weights can be assigned to each one of them.

One of the main characteristics of this family of algorithms is the ability to efficiently reorganize the data. When a system change occurs, they provide optimal or near optimal solutions that minimizes the number of objects to be moved in order to keep the data balanced. The algorithms were thought to handle the addition and removal of a group of buckets instead of individual modifications. This way removing a bucket is not very efficient comparing to the optimal solution. However if removing a sub-cluster (group of buckets) the performance is near optimal.

The problem with this algorithm family is that it is not able to calculate the inverse. This means that is hard to answer the question which objects are stored on a given disk. For doing this there is the need to iterate through all the object identifiers, figure out where each key is stored and then return only the objects that are kept in the questioned disk.

3.5 Controlled Replication Under Scalable Hashing

CRUSH [24] is closely related to the algorithm family RUSH, providing the same essential characteristics like decentralization, weighting of devices, well balanced distribution of data and optimal or near optimal reorganization.

The biggest novelty introduced by this algorithm is mainly the improved control over replicas placement, with the possibility of creating different failure domains. This feature allows a better control over the replicas locations which improves the reliability of the system. For example it is possible that three replicas of the same object are defined to be placed in three different cabinets so they do not share the same electrical circuit.

3.6 Distributed Database Management Systems

Data partitioning is implemented in almost all major DDBMS products like Oracle, DB2, SQL Server, MySQL and Postgres. By reducing the size of tables, this feature intends to provide improved performance and simplify data management.

The types of partitioning usually available in these systems are:

- **Range:** defining various ranges to which data is assigned;
- **Hash:** allows to separate data based on a computed hash key that is defined on one or more table columns, with the end goal being an equal distribution of values among partitions;
- **Key:** performs an even distribution of data through a system-generated hash key;
- **List:** the data is partitioned by explicitly listing which key values appear in each partition.

Data partitioning is usually done considering the expected workload, in order to achieve better performance in query processing. However being the distribution based on ranges and hash functions, it may not provide a very efficient solution when the system changes and data needs to be redistributed. This issue can be quite important if the composition of the system happens to be changing very often, like in the cloud where users are able to flexibly scale up or down depending on the current demand.

Many database vendors also offer a cloud-oriented flavor of their products. We have the examples of SQL Server for Amazon EC2, MySQL Enterprise for Amazon EC2 and Microsoft Azure SQL. They all claim to be able to scale databases up and down based on the business needs. However, being them all proprietary products, no more details are given. There is no further information on how this process is done or how efficient is it.

One last product with much potential is Greenplum, a database software able to scale to large distributed systems. It relies on a shared-nothing architecture, trying to take the most advantage of parallelism. This software distinguishes itself from others for the fact that it also implements a MapReduce technique.

Chapter 4

Data Distribution

One fundamental step into database distribution concerns the way data is spread among the various nodes, a characteristic that will greatly influence the whole behavior and performance of the system. Here some questions arise: how and how much should data be fragmented, how should it be allocated and how much information should be kept about its location. Having this in mind and also thinking about cloud computing, some algorithms are going to be addressed in order not only to allow the distribution of data but also to grant adaptability facing systems modifications.

4.1 Introduction

Data distribution is one of the most important issues to consider in a distributed database. Even more if we are going to deploy it in a distributed environment such as cloud computing, where the data partitioning has to adapt to system changes. In this work we aim to take the most advantage of parallelism, balance the data as equal as possible among the system and minimize the amount of metadata that need to be maintained.

A database is composed by relations, and a relation is composed by attributes and records. Taking this into account, what would it be a reasonable unit of partitioning? A whole relation as partition unit is not a good option for many obvious reasons. First of all, very often a database only holds few relations, which makes it hard to distribute them throughout many nodes. Then only having a relation per node does not allow to exploit much parallelism. Moreover the various relations of a database can have very different sizes which does not allow a proper data balancing.

We need a more fine-grained approach to fragmentation. Looking inside rela-

tions we find attributes and records as possible partition units. Here we have three ways of partitioning the data: vertically (attributes), horizontally (records) and in a hybrid way (both). Because MonetDB is a column-oriented DBMS, choosing to distribute the data vertically would seem to be the obvious option. However, relations are usually composed by few columns and many records, which would eventually require the introduction of hybrid partitioning in order to get the data balanced. Partitioning the data horizontally is the better option where data balancing can easily be achieved. Because of this we are going to consider the last approach and partition the data horizontally, which involves distributing complete records over the various nodes.

The distribution of the records can be approached in two different ways: just dividing the whole data over all the machines and not keeping any knowledge on the position of each record; or using a distributed data structure so the place where each record is, can be known via some administration.

Not keeping information on the data location might be easier and faster to do, but problems come when we want to query the data. If we do not know the position of each record it is not possible to define a query plan and send it to the correct nodes, like it is traditionally done in Distributed Database Management Systems. However, nowadays there are other efficient techniques to query data that do not rely on any information about its location. An example of such technique is Map/Reduce, that was recently made popular by Google. It works by just querying every node in the system in order to achieve the final answer. This way, we are able to consider both alternatives to perform the data distribution.

In the rest of this chapter we call the places where data is stored, buckets. This abstraction may allow us to take better advantage of the whole system resources, as one machine is able to hold multiple buckets. This means that if a machine has more resources than others, one can just put more buckets on it and make a better use of the total resources.

We now start by first presenting the typical most simple techniques to distribute data. Next, we will explore some more flexible algorithms that are better capable to adapt to the cloud concept.

4.2 Simple Algorithms for Data Distribution

When allocating data in a distributed fashion there are three algorithms that immediately come to mind. The easiest way is just dividing equally the records over

the available machines. Another possibility is using a hash function, that requires a bit more administration, mapping the key of each record to a machine. The last alternative is to define key ranges and assign them to a machine. In the next section we discuss these techniques in detail.

4.2.1 Division

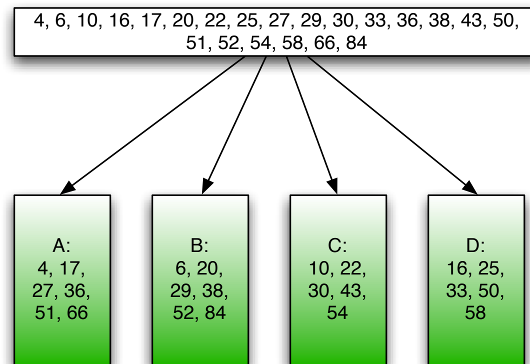

Figure 4.1: Loading data using the division algorithm

We call the first algorithm *Division* because it just consists of dividing equally the records over all the buckets present in the system. This is done by only considering the total number of tuples and it does not take into account any key or content they might be holding. Here we do not keep any metadata that identifies the location of each tuple.

In Figure 4.1 we can see how a set of data is distributed among four different buckets. The data was placed considering a round-robin technique, and we can see it is perfectly balanced.

When adding a new bucket to the system, we need to redistribute the data so new node has the same amount as the others, keeping the system balanced. Figure 4.2 illustrates what has to be done. We first divide the total number of records (22) by the total number of buckets (5). The result is 4 records, which represents the number of records that the new bucket will have to hold. As we had previously 4 buckets and 22 records, we will have to move one record from each to the new one.

When a bucket is removed the opposite occurs. As we can see in Figure 4.3, here all the data has to be shipped from the bucket to be removed into all the others. We want to keep the data as much balanced as possible, but if the division of the number of records by the numbers of buckets produces rest then the first buckets

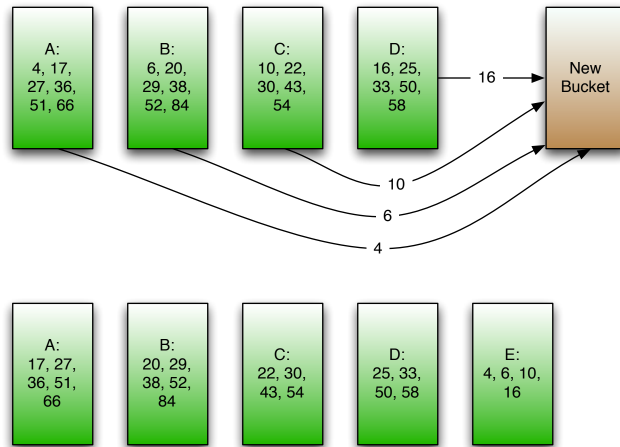

Figure 4.2: Adding a bucket to the system and redistributing the data with the division algorithm

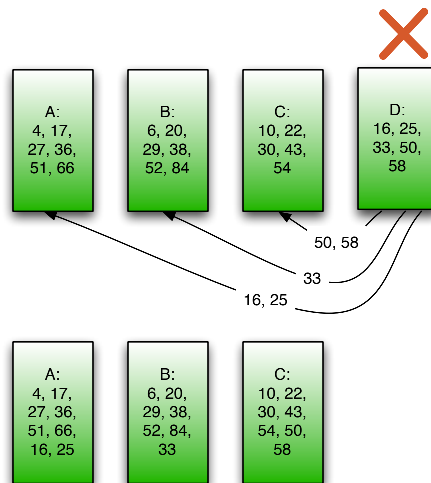

Figure 4.3: Removing a bucket from the system and redistributing the data with the division algorithm

(from the left to the right) will hold one more record than the others. This way the number of records to be moved for each bucket may vary, but at most one record, which in larger (more realistic) cases is negligible.

For both adding and removing a bucket we always only have to move the least records possible and the system is keept perfectly balanced.

4.2.2 Hashing

While the Division algorithm does not care about the content of the data, the Hashing algorithm works by using the defined key in each row. The hash function takes as input the identifier of each row and calculates a value in a specific range. In our case the output of the hash function maps the number of buckets we are addressing so the value calculated corresponds to the bucket where the record is going to be stored.

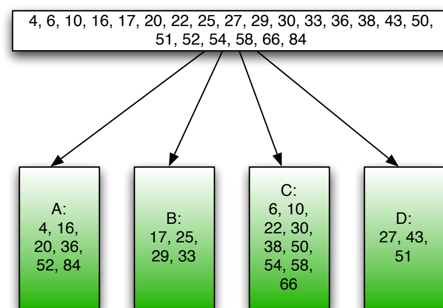

Figure 4.4: Load a set of data using the hash algorithm

Depending on the primary keys and the hash function it may be possible to have a more or less balanced system. The fact that we are always able to tell the position of each record may be very helpful if the system is exposed to many point queries.

However when it comes to adapt to a new system set, this algorithm requires to move more data than the simple divider. If the address space changes, by adding or removing a bucket, every single key has to be recalculated by the new hash function.

In Figure 4.4 the distribution of a data set among 4 different buckets is shown. The data was placed using a hash function which calculated the respective bucket for each key. The hash function being used is: f(x) = x mod n; where x is the key and n the total number of buckets available. As we can see the data ends up not being much balanced as the keys are not a uniform distributed sequence.

If we want to add a bucket to the system, then the address space of the hash function changes, which leads to a whole reorganization of all the data. This way, the position of every key has to be recalculated and eventually many records have to be moved to the new correct locations. This forces, in the most cases, a big percentage of the data to be moved, causing lots of traffic between all the buckets. We can see in Figure 4.5 that after adding a new bucket all the data is moved between the

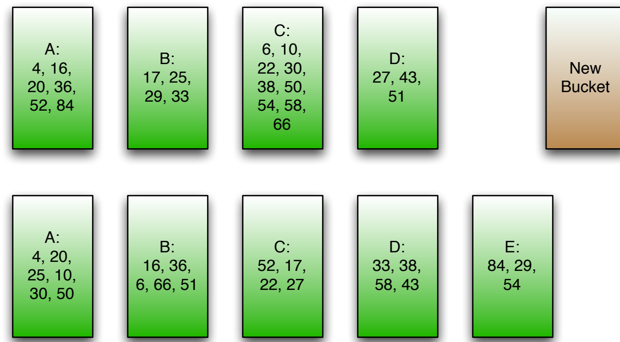

Figure 4.5: Adding a bucket to the system and redistributing the data with the hash algorithm

existing buckets and also to the new one. For readability the arrows with all the data exchanges are not shown.

When removing a bucket the same situation happens. The address space changes, so every key has to be recalculated. This also leads to interchange of records between all the buckets in order to keep the data in the correct places. Figure 4.6 shows the removal of one bucket, changing the system from having 4 buckets to 3. The data is still not well balanced, being bucket A holding 9 records, bucket B holding 8 and bucket C holding 5.

This algorithm requires a lot of data to be moved every time a change occurs in the system. However it allows to easily identify the place of each record at any time.

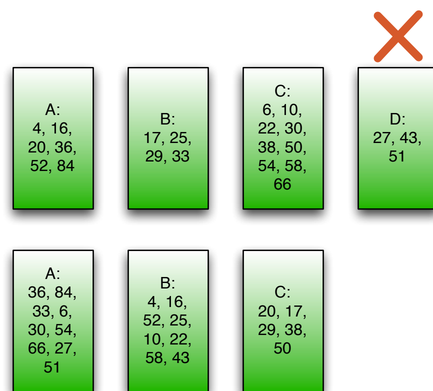

Figure 4.6: Removing a bucket from the system and redistributing the data with the hash algorithm

4.2.3 Range

The Range algorithm distributes data in consecutive ranges of primary key values. Here we have two possible approaches to achieve a balanced system, depending on the denseness of the keys. If we have a dense set of keys, then we can just mathematically divide the total range in equal parts for the existing machines. Here we now the ranges upfront so we just have to put each record in the respective place. On the other side, if the set of keys is sparse, we first have to sort the data and then make ranges based on the number of rows. Sorting the records and counting them is the only way to define ranges that balance the data.

This algorithm can be helpful when facing point and range queries, as we always know the exact position of each record and they are organized by ranges.

In Figure 4.7 we can see the distribution of a set of data using the Range algorithm. Because the set of keys is not dense we have to define the ranges in each bucket with different sizes in order to have a balanced the system. Bucket 1 is holding the keys from 4 to 20, bucket 2 from 22 to 33, the bucket 3 from 36 to 51 and bucket 4 from 52 to 84.

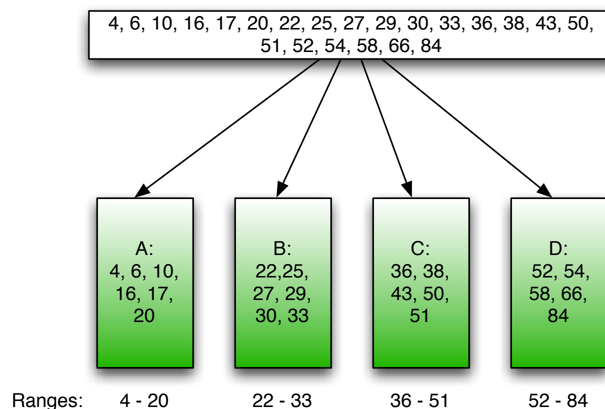

Figure 4.7: Load a set of data using the range algorithm

When adding a new bucket to the system all the ranges have to be redefined, which implies moving more records than the minimum optimal. We can see in Figure 4.8 a representation of this operation. Here we see that besides having to move the records to fill the new bucket, we have also to move records in between the old buckets.

The opposite procedure is done to remove a bucket. This also implies moving many records between buckets, in order to keep the system balanced. Besides having

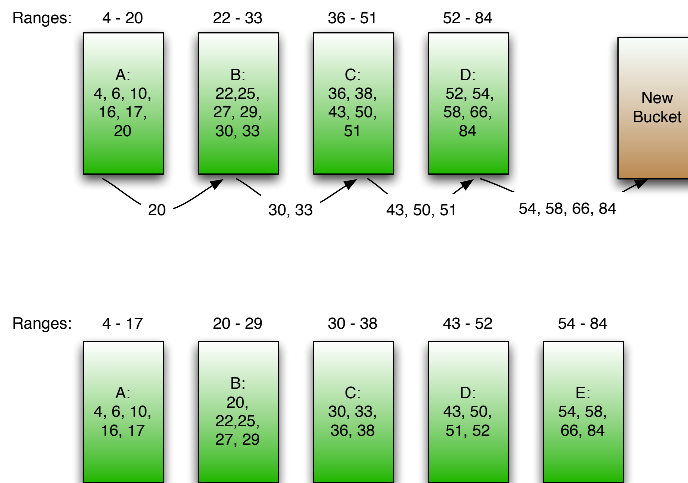

Figure 4.8: Adding a bucket to the system and redistributing the data with the range algorithm

to distribute the records from the bucket to be removed we also have to transfer data between the other buckets.

This algorithm could also have been approached in a different manner, minimizing the amount of data to move whenever a system change occurs. This would be done by allowing to have more than one range in each bucket bucket and keeping metadata that identifies which bucket have which range. Imagining that a new bucket is added we would just have to take a portion of the ranges in each existing bucket and transfer it to the new one, without having to move unnecessary data. However this approach can become complicated with all the adding and removing of buckets, getting harder to store and maintain the metadata.

4.3 Revised Algorithms for Data Distribution

We saw that some of the previous algorithms, as Hashing and Range, are not so efficient when data needs to be reorganized, requiring the movement of much unnecessary data. Because we want to address an elastic system like the cloud we need solutions that can reduce the movement of data as much as possible.

We are only going to proceed with algorithms based on division and hash. The study on algorithms based on range techniques are left out in our work, mostly due to matters of time. With the decision for Hashing and Division we still have the contrast between algorithms that distribute the data according to their key value

and algorithms that do not.

4.3.1 Linear Hashing

Linear Hashing comes mainly to overcome the inflexibility of normal Hashing when the address space of the hash function changes. This algorithm works in a more dynamic manner allowing the expansion of the address space without the need to recalculate all keys. We can address records to a specific number of buckets and if this number changes only the keys in one bucket need to be recalculated.

Like in the hash algorithm, here all the records are initially loaded using a hash function that relies on the primary key of each record and the total number buckets available. The Figures 4.9 and 4.10 illustrate the execution of the algorithm when a new bucket is added and removed respectively. Here we can see how the variables n and *bucket pointer* evolve and how the data is redistributed.

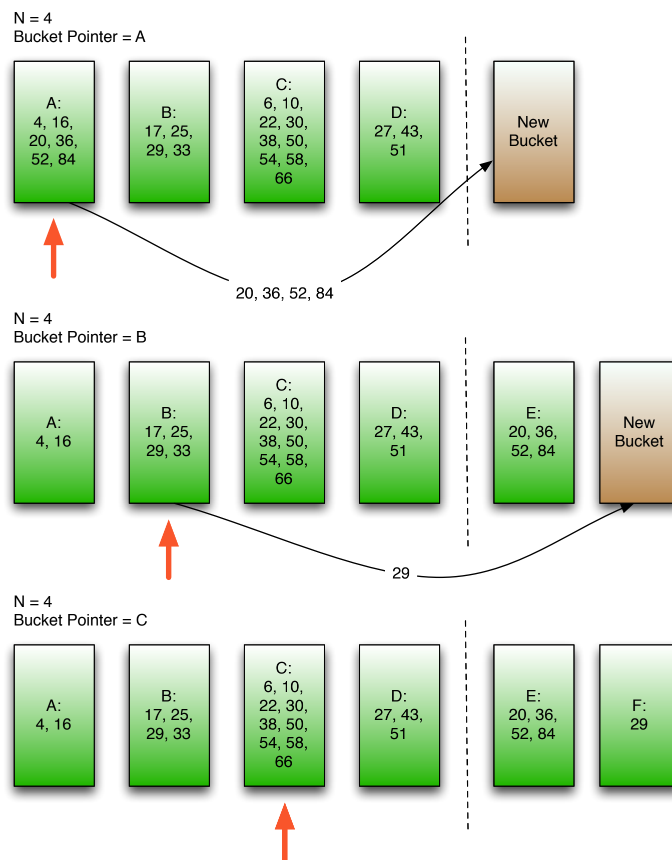

Figure 4.9: Adding a bucket to the system and redistributing the data with the linear hashing algorithm

The hash function used is the same as previously: f(x) = x mod n; where x is

the key and n the total number of buckets available. Looking to Figure 4.9, we find that adding a new bucket requires the recalculation of all the keys present only in the bucket being pointed (A). In normal Hashing if a bucket is added, the hash functions changes and all the keys need to be rehashed. In Linear Hashing this does not happen because the algorithm evolves with two different hash functions. All the buckets before the *bucket pointer* and after n have their records mapped according to a different function than the buckets that are in between. In the example the buckets before the *bucket pointer* and after n are mapped by f(x) = x mod 8, and the ones in between still use f(x) = x mod 4.

In the figure we can see that in the first iteration of adding a bucket, only the keys on bucket A were recalculated, and 4 of them were passed to the new bucket E. After this the bucket pointer is moved to B. If we add another bucket the same happens, being the keys in B recalculated and if necessary moved to the new bucket (F). In this case, only one key is moved. After this move, the bucket pointer points to C. This goes on for the addition of two more buckets, until the bucket pointer exceeds n. When this happens the algorithm is reseted (pointing again to A) and the variable n is doubled (becoming in this case 8).

To remove a bucket, the reverse process has to be performed. This algorithm works in a linear way, so when new bucket is added it has to be put in the end of the sequence and when removing a bucket it has also to be taken from the end. To better understand how this works consider the example in Figure 4.10. To remove bucket D first the bucket pointer has to be decremented. However, the pointer is already pointing to the first bucket. Hence, n needs to take half of its value, in this example 2, and the bucket pointer will point also to the bucket correspondent to n, in this case B. Then all the records in D are moved to B. Removing another bucket is similar. First the bucket pointer is decremented to A and then all the records are moved from bucket C to bucket A.

In Linear Hashing only the last bucket can be removed. However this might not be a problem if we are addressing a distributed system like cloud computing. The abstraction provided by the cloud allows us to purchase virtual machines without really knowing the physical structure that is behind. To the clients eyes the virtual machines available for purchasing may all have the same specifications and can be acquired as many as needed. They are physically maintained by the service provider, transparently to the client. This way, because we are able to play in a virtually homogenous system, using a linear algorithm just able to remove the last bucket is as good as using any other algorithm able to remove any bucket.

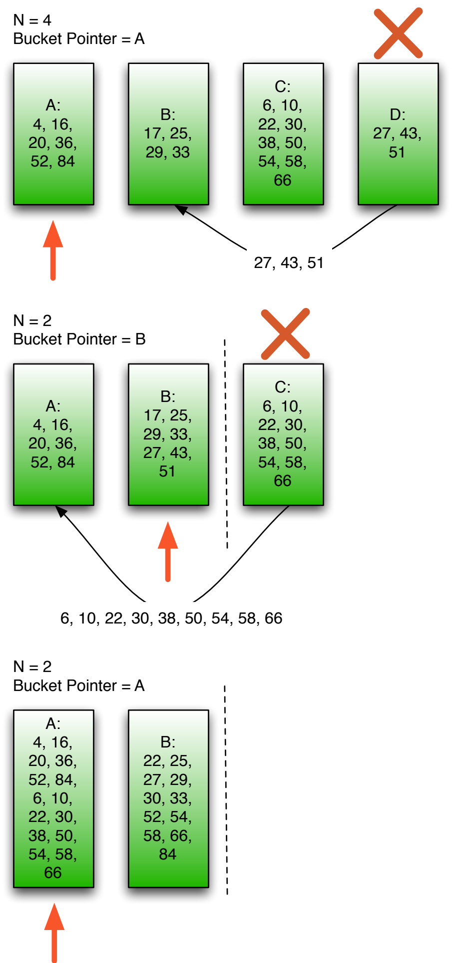

Figure 4.10: Removing a bucket from the system and redistributing the data with the linear hashing algorithm

4.3.2 Linear Division

The Linear Division algorithm is based on splitting in half and merging complete buckets. Initially all the records are loaded by dividing the data according to the total number of records and the number of buckets. Figure 4.11 and 4.12 show how this algorithm proceeds when a bucket is added and removed respectively. The system starts with the data loaded as with the Division method.

When adding a new bucket half of the keys in the bucket currently being pointed to, have to be moved to the new bucket. The bucket pointer is incremented by one afterwards. This repeats till the pointer reaches n where it comes back to A and n doubles its value. In Figure 4.11 this scenario is displayed. A new bucket is being added so the first half of the data in A is passed to the new bucket E. Then the bucket pointer is moved from A to B. Adding another bucket produces an identical

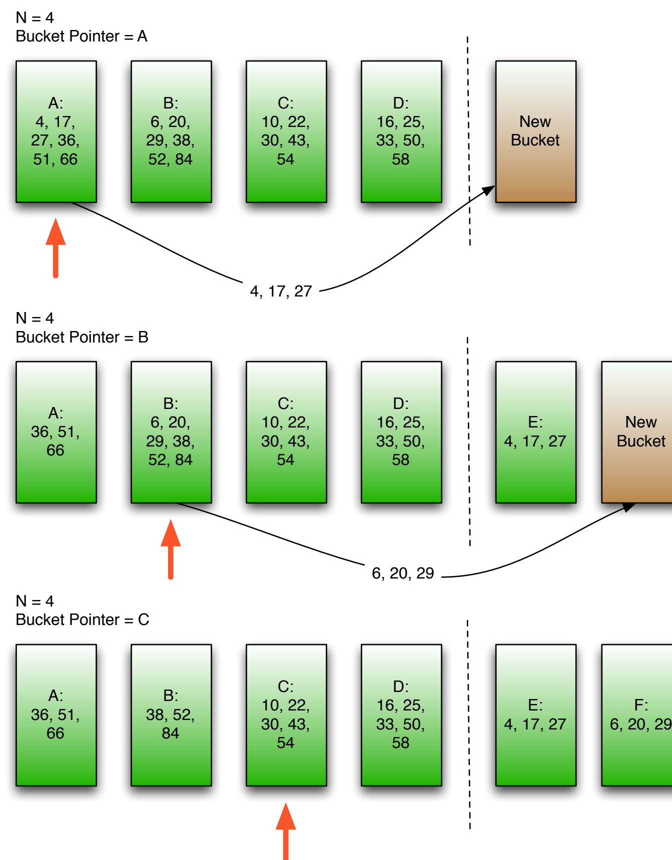

Figure 4.11: Adding a bucket to the system and redistributing the data with the linear division algorithm

effect. Half of the data is moved from B to the new bucket F and the bucket pointer goes to C. This continues until the bucket pointer reaches n (D is the fourth bucket in our case). Here n is doubled, becoming 8, and the bucket pointer returns to A.

To remove a bucket from the system we have to do the reverse process. The Figure 4.12 illustrates the removal of two buckets. For removing the bucket D we first have to decrement the bucket pointer. However in this example the pointer starts in the first bucket. This way n have to become half of its value, 2 in this case, and the bucket pointer now points to the 2nd bucket in the sequence, B in this case. Then all the records in D are moved to B and D is finally eliminated. To remove another bucket we just have to do the same thing. First decrement the bucket pointer, now to A, and then move all the records from bucket C to bucket A.

In this algorithm, like in Linear Hash, only the last bucket can be removed. However, as we saw in the previous section, considering the characteristics of the

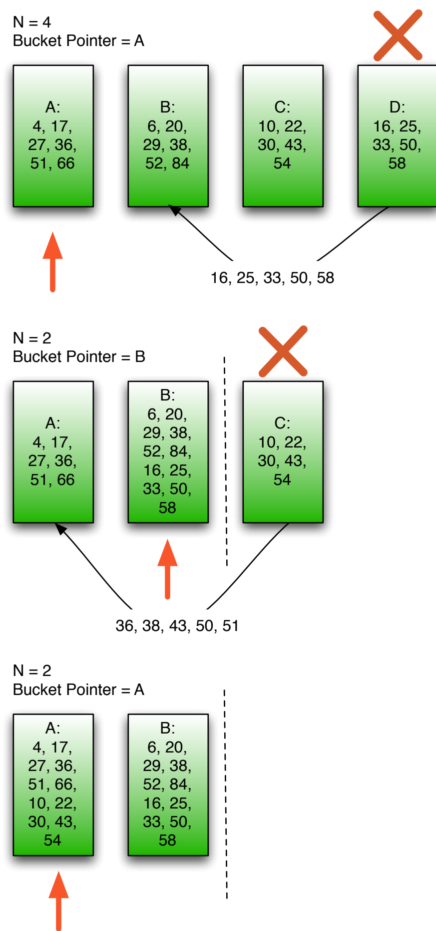

Figure 4.12: Removing a bucket from the system and redistributing the data with the linear division algorithm

cloud this restriction does not stand as a problem.

4.3.3 Consistent Hashing

Consistent Hashing is presented as a hash-based distributed algorithm able to efficiently cope with address space changes. The big advantage of addressing data with this algorithm is that whenever the system grows or shrinks we only need to move the optimal minimum number of records to keep the system balanced. Unlike Linear Hashing, here every bucket can be removed with the same cost and the system is kept balanced at all time.

To demonstrate how this algorithm works, let's imagine a ring representing the whole range of keys. This range has to be known in advance and cannot change during execution. To each bucket is assigned a defined number of bucket points that

are randomly distributed in the ring.

The records are stored in the bucket holding the bucket point correspondent to its key, if there is none the record is stored in the bucket holding the next bucket point in the ring. The only administration that needs to be kept is the position of the bucket points in the ring.

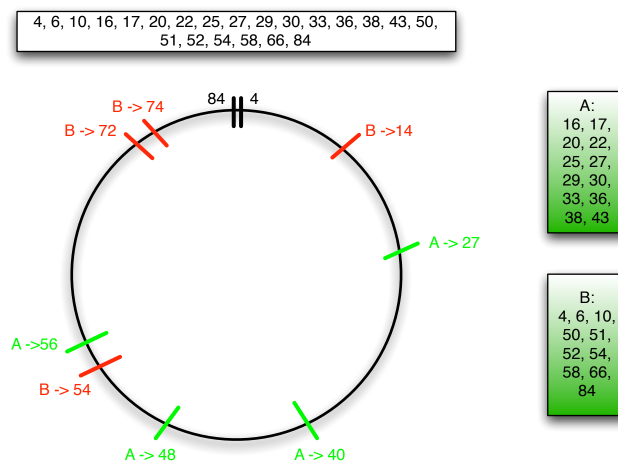

Figure 4.13: Loading data using the consistent hash algorithm

One factor that can influence the data balancing is the number of bucket points present in the algorithm. This should not be a complete mapping of all keys in the ring, in order not to store much metadata. However, intuition says that the less bucket points we have per bucket the more chances for data to be unbalanced. We want the number of bucket points to be large enough so each bucket is assigned to a reasonable number of different points in the ring. This should be considerably big comparing to the initial number of buckets, so even if the system grows, each bucket should still be assigned to a significant number of points. When choosing the total number of bucket points one should make an educated guess concerning the system we are addressing.

Figure 4.13 represents a system with two b buckets (A and B), each of those with four bucket points randomly distributed in the ring. We want to load the 22 keys ranging from 4 to 84 that are shown on the top. The final distribution of the keys is shown in the two buckets on the right side of the picture. The bucket points 27, 40 and 48 all belong to bucket A and are assigned consecutively in the ring. This way, it would be enough to only represent the point 48, as all the keys from 15 to 48 are going to bucket A. However we represented them all to make explicit that to each bucket, four bucket points were assigned.

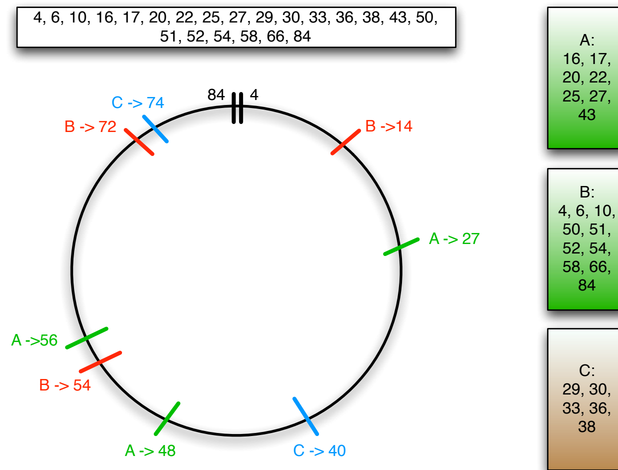

Figure 4.14: Adding a bucket to the system and redistributing the data with the consistent hashing algorithm

In Figure 4.14 we can see a possible resultant distribution from adding a bucket to the system of Figure 4.13. The algorithm randomly takes a bucket point from each bucket giving it to the new bucket C. Then all the data associated to those bucket points has to be moved to bucket E. In the end bucket A and B hold 3 bucket points and C only 2 because the total number of bucket points in the system is only 8. The amount of data moved was only the necessary to fill the new bucket.

To remove a bucket the algorithms does the opposite. It uniformly assigns all the bucket points from the bucket being removed to the remaining buckets and the necessary data is moved.

4.4 General Overview

In this section we discuss and compare some of the most important characteristics of the previously presented algorithms. In Table 6.1 we can see a brief comparison in terms of data moved when adjusting the system and the achieved data balancing.

All the algorithms here discussed can present a perfectly balanced system. However, this is sometimes dependent on some factors, as the keys being addressed, the hash function chosen or the execution state of the algorithm.

The Division and the Range algorithms are able to do it at all time, independent of those factors. The Division because it simply distributes the data evenly over all the machines. A Range algorithm, if we do not define equal ranges depending on the total range of keys, but variable ranges depending on the distribution of the

Algorithms	**Data Balancing**	**Records Relocation**
Division	Perfect	Minimum
Hashing	Variable	Large
Range	Perfect	Large
Linear Hashing	Variable	Variable
Linear Division	Variable	Small
Consistent Hashing	Variable	Variable

Table 4.1: Data distribution algorithms comparison

keys.

The hash-based algorithms, like Hashing and Linear Hashing, have their data balancing dependent on the hash function being used and on the keys being addressed. If the hash function is adequate to the keys, we may achieve good results, otherwise we might end up with a very unevenly loaded system.

For linear algorithms, like Linear Hashing or Linear Division, the system can be more or less balanced depending on the state of execution of the algorithm. This means that if we start with 4 machines and we have a system that is growing, it is only perfectly balanced again when it reaches the double of the initial number of nodes, 8 machines in this case.

In Consistent Hashing we consider the whole range of keys as a ring, where data balancing depends on the distribution of the bucket points and on the keys being addressed.

Another important characteristic differentiating the previous algorithms is the amounts of data that have to be moved when readjusting the system. The Division algorithm is the only one to readjust moving always just the minimum amount of data to keep the system balanced. On the other hand, techniques like Hashing and Ranges imply moving large amounts of data. In these algorithms, besides having to move data from/to a bucket that is being removed/added, we also have to relocate data between all the other buckets.

The Linear Hashing can move a variable number of records, it depends on how they were distributed by the hash function when loading. In turn Linear Division starts perfectly balanced so it moves a constant amount of data till the number of buckets reaches double or half. This can actually be useful if we are only adding and then removing some buckets without reaching the double. Here the system is not perfectly balanced but we still add distribution, which may also improve the queries

performance.

Consistent Hashing moves a variable amount of records when readjusting the system depending on how data was loaded initially and on which bucket points are having their bucket assignment changed. Linear Hashing and Consistent Hashing differ from simple Hashing in the fact that when the system setting changes they only need to move records to fill/empty one bucket. No extra data is moved between the other buckets to readjust the algorithm.

All the algorithms here presented are very light in terms of administration, there is just the need to keep the information on the total number of buckets, total number of records and sometimes a pointer or two. The only exception is the Consistent Hashing algorithm that requires to store a mapping between all the buckets and respective bucket points.

Concerning the ways distributed data can be queried we are aware of two possible approaches. One is the traditional implementation of a DDBMS, where information is stored on the exact location of each record, so the queries can be prepared to address the right parts of the system. On the other hand we also have algorithms like the revived Map/Reduce, which provide an easy way to distribute the workload through the whole system, querying data without needing to know where each specific record is. Only data distributed with Division and Linear Division cannot be queried using the first approach.

Chapter 5

Prototype

In this chapter we present a prototype application that implements the behavior of some described algorithms when facing systems changes. After implementing the algorithms we perform some evaluations in order to compare the efficiency of each when distributing and redistributing data.

5.1 Implementation

5.1.1 Data Set

To proceed with further research on data distribution we have chosen a data set to play with. The data represents a real world scenario, but at the same time has a relatively simple schema so we can focus more on the distribution issues. This is the "ontime" data set [18] that contains the information about all airplane flights in the USA from 1987 until 2010. In Appendix B we can find the SQL code that creates the schema, which is only composed by one relation that holds 94 attributes. The total size of the data rounds approximately 50GB. However it can also be downloaded in smaller chunks, which comes in very handy for the purposes of our project.

The two small drawbacks of this data are the inexistent primary key and the existence of some duplicate records. Therefore we defined ourselves a primary key constituted by three attributes: date, flight number and origin of the flight. We also removed the duplicate records from the data set.

5.1.2 Technological Choices

The prototype is written in Java and the chosen underlying database system is MonetDB. Each bucket is represented by a different database and operations involving data are performed through JDBC. The different databases are managed through Merovingian[?], a useful daemon that comes as part of MonetDB.

Merovingian enables the creation and maintenance of databases located in different places of the same network and can also be controlled from a Java program. For example when adding a new bucket the Java program communicates with Merovingian requesting the creation and initialization of a new database. The inverse happens when removing a bucket, the Java application asks *Merovingian* to stop and destroy the chosen database. This allows the entire experiment to be performed from the Java application.

5.1.3 Algorithms

The implemented algorithms are some of those discussed before in Chapter 4: Division, Linear Hashing, Linear Division and Consistent Hashing. We chose these ones as they seemed to be the ones that can give a better answer when having to redistribute the data due to a system change.

- **Division:** This algorithm loads data in a round robin fashion, providing a perfect balancing. For doing this we just keep a pointer that goes around in the sequence of buckets indicating to where should the next record go.

  To add a new bucket we first calculate how many records should each bucket have in the new setting, diving the total number of records by the total number of buckets. Then we take the necessary records from each bucket (the ones before the pointer have one more record than the others) and we move them to the new bucket.

  To remove a bucket we take all the records from that bucket and we redistribute, following a round robin algorithm.

- **Linear Hashing:** To perform this algorithm the data is first loaded according to a hash function. Here the place where each record has to be stored is the rest of the division of the record id by the number of buckets in the system. The algorithms evolves depending on a bucket pointer that is initialized at the first bucket in the sequence of buckets.

Adding a new bucket we have to have recalculate all keys of the records in the bucket being pointed. Their position is now the rest of the division of the record id by the double of the initial number of buckets. This only can have to outcomes, either the record stays in the same bucket or is moved to the new one just added. In the end the bucket pointer is incremented.

To remove a bucket the bucket pointer is decremented and all the records in the last bucket are moved to the bucket being pointed.

- **Linear Division:** In the Linear Division we load the records with a round robin algorithm like we did in the Division.

  To add a new bucket we proceed as in Linear Hashing, but this time we do not calculate the keys of the records that have to be moved, we just move half of the records in the bucket being pointed.

  To remove a bucket is the same as previously, the bucket pointer is decremented and all the records in the last bucket are moved to the bucket being pointed.

- **Consistent Hashing:** This algorithm is initialized by first identifying the range of ids of the whole set of records, in order to create a ring of keys. Then a defined number of bucket points is assigned to each bucket and spread randomly all over the ring. To load the data, we take each record's key and we look for its match in the ring. If there is none it goes to the bucket holding the next bucket point.

  The number of bucket points assigned in the beginning of the algorithm does not change during execution. When adding a new bucket we divide the total number of bucket points in the system by the number of buckets, to know how many bucket points each bucket should have. Then we randomly take the necessary bucket points from every bucket and we assign them to the new bucket. Finally, we go through all the records in each bucket, to see which records have to be moved to the new bucket in order to follow the ring distribution.

  To remove a bucket we do the inverse process. First we calculate how many bucket points each bucket has to have. Then we take the bucket points from the bucket being removed and we randomly assign them evenly to the other buckets. Finally we go through all the records in the bucket to delete and we move them to the new correct place according to the ring.

5.2 Evaluation

In order to evaluate the behavior of the different algorithms presented, three important scenarios where the data has to be distributed are tested: loading the data; adding a bucket; removing a bucket.

In the operations of adding and removing a bucket we have considered two different measures of comparison: the number of records moved and the time elapsed. All tests were performed only in one machine, where each bucket is represented by a different database. Here it is not possible to see the time that would be spent if different machines would have to communicate over a network. However we are also comparing the different times elapsed, as we interest in seeing how much the performance variates with this conditions.

Moreover we know that the cost of redistributing data is mostly related to the amount of data that have to be shipped between buckets. This way and because we are aiming for efficiency managing the distribution of data, the number of records moved in each case seems to be a good term of comparison.

5.2.1 Loading the Data

To load data, we consider three different algorithms: Division, Hashing and Consistent Hashing. In this section we present four graphics in Figure 5.1 where different numbers of buckets and different amounts of records are experimented. This allows to understand the effects in performance caused by the modification of this parameters, being the time elapsed to load the data measured in the y-axis.

Going from 4 to 32 buckets we see that with 1000 tuples there are no visible differences between the two algorithms, but as the amount of data grows they eventually diverge. Using 100000 tuples we see that the Hashing performs slower.

We can see that both Division and Consistent Hashing take less time than the Hashing algorithm in all tests. One might find it odd that being Consistent Hashing a hash-based algorithm, how come it is performing as well as Division. This result from the fact that what is slowing down the Hashing is not the calculation of each record's key, but the lack of data balancing that overloads some buckets.

The hash loading takes more time than the consistent hashing because it is uneven for our set of keys. In the consistent hash the data balancing is mostly defined by the disposition of the bucket points in the ring. This is done in between a specific range of keys by an uniform distribution.

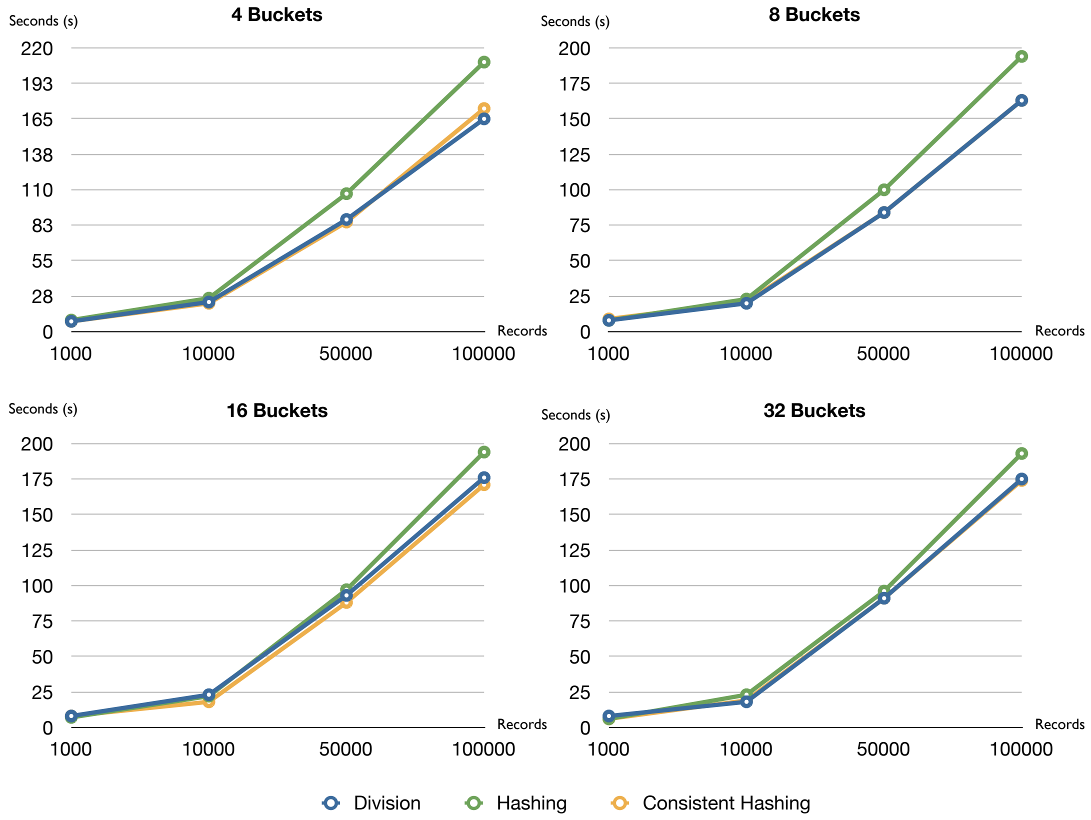

Figure 5.1: Load operation

5.2.2 Adding a bucket

In this section we compare the redistribution of the data made by the different algorithms when a new bucket is added. The comparison measures are time consumption and number of records moved, represented in the graphics of Figures 5.2 and 5.3 respectively.

The amount of data being used is 100000 records which is not so huge making the evaluation take ages but big enough to see the differences between the algorithms. We are doing the tests starting with 4 buckets and then successively adding a new one until we have 16 buckets.

Analyzing the graphics we see that the first one follows the second, i.e., the time elapsed adding a bucket is mostly influenced by the amount of data moved. As the number of buckets increase, the amounts of data being moved decrease, which makes it faster to readjust the system.

In the Linear Hashing algorithm we noticed that with the keys present in our data set, the system started up very unbalanced. The 1st bucket was holding 72515 records, the 2nd 6022, the 3rd 15460 and the 4th 6003. This way the execution of this algorithm was not very uniform, with places where much time was wasted moving lots of data (4,8,12). However, because of this, the rest of the execution was faster comparing to the others.

The execution of the Division algorithm should be taken as a reference, because it is the one always keeping the data balanced, only moving the least records possible and doing the least calculations. However we can see that Consistent Hash actually performs very similarly, with a good data balancing, where the amounts of records moved are decreasing almost at the same pace as in the Division. This happens because the data balancing in the Consistent Hashing algorithm is mostly defined by the random function that distributes the bucket points over the ring and not so much by the record's keys.

Finally, Linear Division performed as expected, moving constant data at constant time for the same iteration of the algorithm. This can be useful if we just want for instances two add more 2 buckets for little time and then go back to the initial 4 buckets setting. Here we would have moved less records, introduced more distribution and probably gained performance in query processing at the cost of having the system not perfectly balanced.

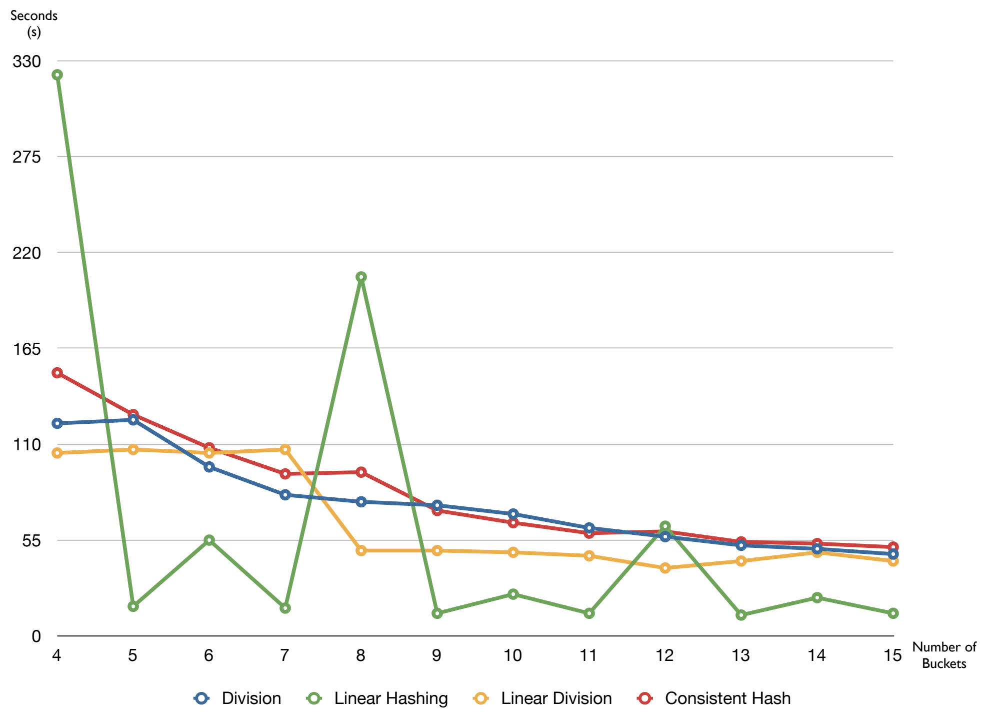

Figure 5.2: Time elapsed adding a bucket

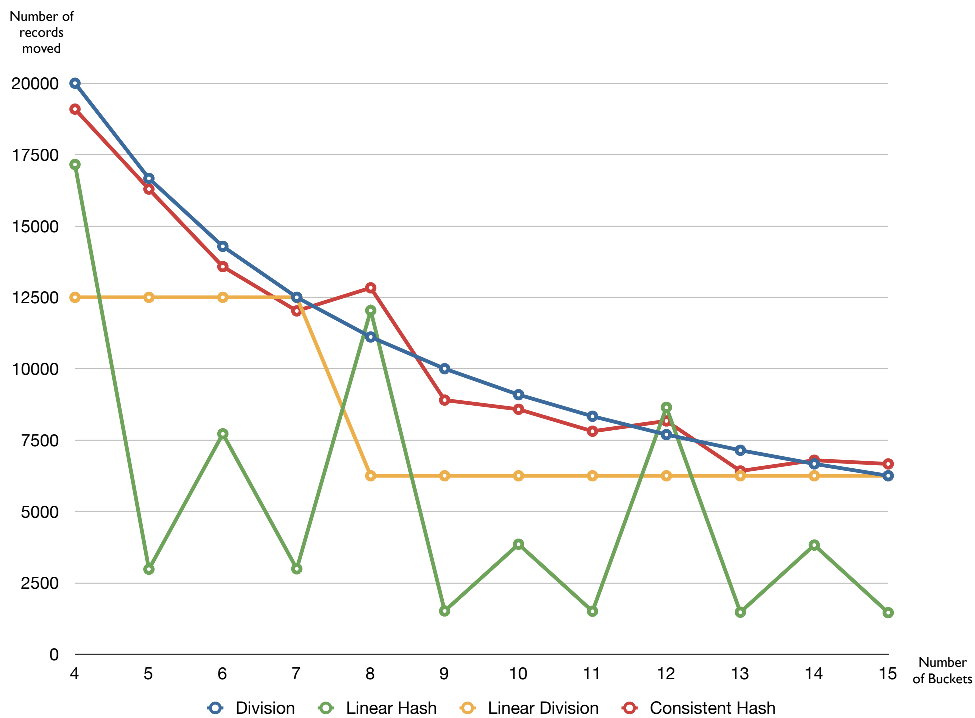

Figure 5.3: Records moved adding a bucket

5.2.3 Removing a bucket

To compare the different algorithms when removing a bucket we also used a data set of 100000 records. The graphics in Figures 5.4 and 5.5 show the obtained results, comparing time elapsed in the first and number of records moved in the second.

Removing a bucket proved to be a much faster operation than adding a bucket. This is mostly related to the fact that when removing a bucket we are reading the data just from one place but the writing, which the heaviest procedure, is done in several databases at the same time. On the other hand, to add a bucket, we are reading data from many places but just overloading one database with writing. Moreover MonetDB does not support concurrent writes, so the data has to be sent sequentially.

Again, as expected, every algorithm's performance follows the number of records being moved. The less buckets there are, the more data we need to move in order to readjust the system, therefore more time is used to perform each operation.

Linear Hashing and Linear Division performed as expected, with the big variations of elapsed time being justified in the variations of data balancing in the system.

Consistent Hashing proved again to be a good competitor for Division an almost linear decrease number of records being moved and elapsed time.

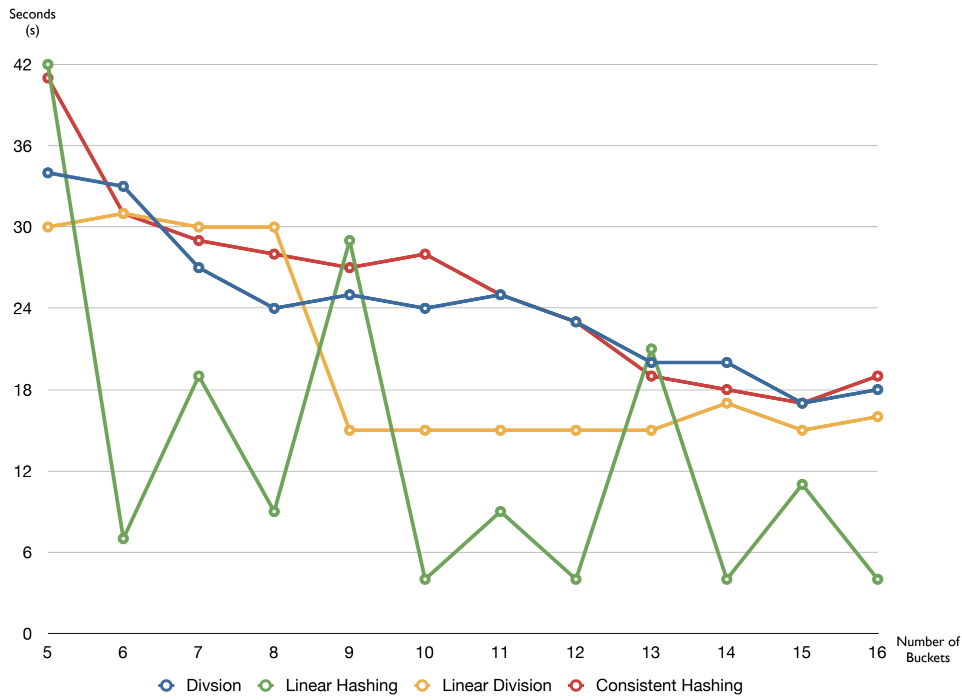

Figure 5.4: Time elapsed removing a bucket

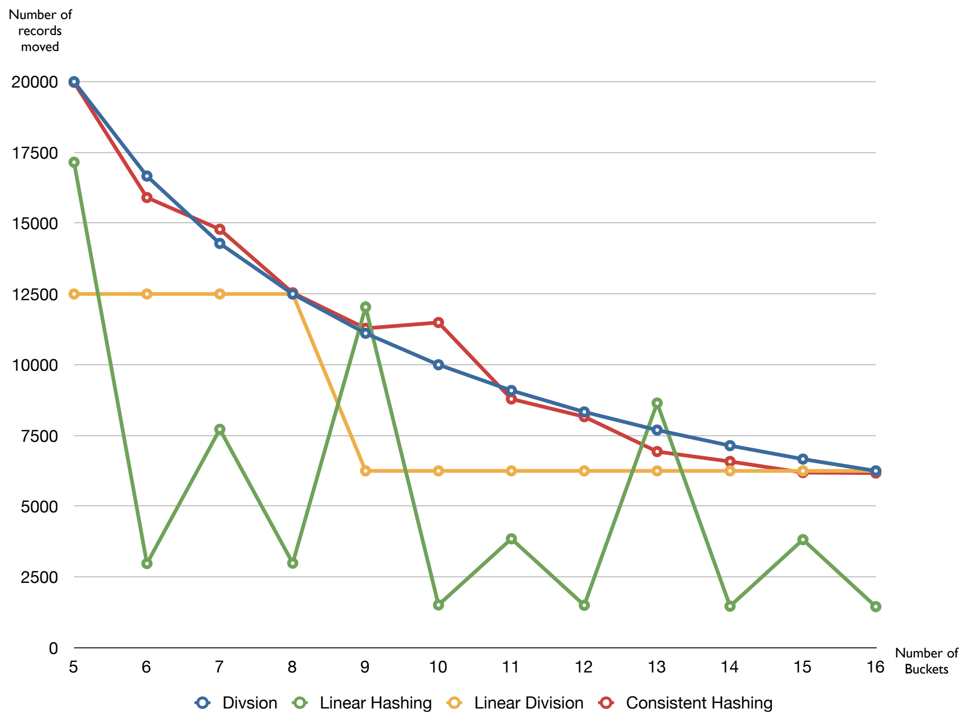

Figure 5.5: Records moved removing a bucket

Chapter 6

Data Replication

Looking at distributed systems, in particular distributed databases, reliability and availability stand out as two very important topics. In a distributed system composed by many machines, there is always a chance that a machine may fail. What we want to provide is a system that can cope with the failure of one machine at a time, being able to fully restore all the lost records. The only way to offer these characteristics is by introducing replication of data.

6.1 Introduction

In this work we study data replication to allow the system to overcome fail over scenarios where we consider a crash-recovery fault model. A cloud does not fail permanently but some of its parts may fail temporarily. If we purchase many virtual machines in the cloud, it is possible that some might fail while the system is being used. This way we are going to address replication techniques that allow the system to continue available in case one node fails and also allow to restore the failing node once it is up again.

For the system to stay available when a node fails all the queries have to be redirected to the replicated data in the other machine(s), until the respective node is recovered. The machine(s) holding the replica are also responsible for restoring the data in the recovered node, so the more distributed the replica is the less cost for each machine. However we must not forget that we are working in a dynamic system where data can frequently be relocated, a fact that may add some interesting challenges when dealing with replicated data.

In this Chapter we research how to add replication to the distribution algorithms from Chapter 5. For ease of understanding we refer to one set of data as the *original*

data and to the other as *replicated data*. This intends to provide an easier and clearer explanation of the algorithms. There is no difference between the two sets of data and they may both be available to answer queries if there is any advantage in that.

When dealing with replication in a distributed system some important questions must be addressed. How fragmented should the replicas be? Should they be distributed for how many machines? Should all the original data and the replicas be placed in different machines? If we want to use the same machines to hold both the replicas and the original data, where should each replica be placed? This are some topics that must be considered in order to obtain a good replication model.

We try to achieve the following characteristics:

- There is one replica of every record in the system;
- The replicated record is on a different machine than the original one (otherwise in case of a failure it would be impossible to recover the data);
- The data should be distributed as evenly as possible through all machines;
- Aim for the most efficient way to do adapt the data to system changes - move the least data possible;

6.2 Two buckets: original and replica

The most trivial replica system is to simply have a replica of each bucket in the system. For the positioning of these replicas we can use an n+1 approach, where there is a sequence of machines and each replica would be put in the next machine. In this simple approach, each machine would just have two buckets, one with the original data and another with the replica of the previous machine. This may be considered naive but it would allow to easily find and maintain all the replicated data.

However there is not much distribution introduced by this technique, as each replica of one machine is completely kept in another machine. In the process of removing and adding nodes, the workload of dealing with the replicas is mostly put on the shoulders of two machines. Furthermore this technique implies moving more data than the optimal minimum to keep the system balanced. This happens because every time the system changes, it is necessary to move full blocks of replicas in order to keep the n+1 assignment correct.

6.2.1 Adding a machine

When a new machine is added, it will always be placed at the end of the sequence of machines. This way, in order to keep the n+1 characteristic correct, the replica in the first machine has to be moved to the new (last) machine. Then, data from all machines has to be moved to the new machine according to the distribution algorithm. Then the respective replicas need to be updated to match the deletion of data from their originals. Finally the data that was moved to the added machine has also to be replicated, which is done on the first machine.

This happens because the first machine is always responsible for storing the replica from the last. Therefore this operation always stresses the first machine, which is not desirable since we try to balance the work between all the participants as much as possible.

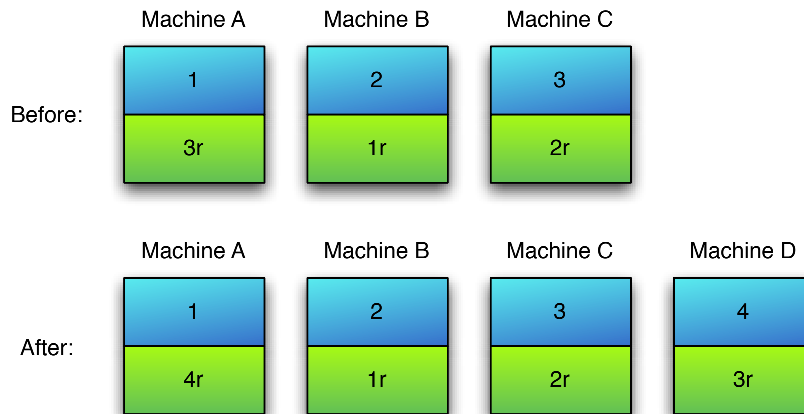

Figure 6.1: Adding a machine using the n+1 approach for the replicas

In Figure 6.1 we can see the state of each machine before and after one machine has been added. First bucket 4 is created with data moved from buckets 1, 2 and 3. As this data is moved to bucket 4 their replicas have also to be deleted in buckets 1r, 2r and 3r. Then 3r has to be moved to *Machine D* in order to keep the n+1 locations correct. Finally bucket 4r is created in *Machine A* with a replica of all the data present in bucket 4.

6.2.2 Removing a machine

To remove a machine from the system, we start by removing its replica from the next machine. Then we move the replica stored in the machine we are removing to the next machine. Finally the original data has to be distributed among the

remaining machines, being created at the same time the replicas of each record in the respective n+1 machine.

6.2.3 Considerations

Two things can be done in order to overcome the issue of overloading always the first machine: we can keep a pointer, going around sequentially, allowing to insert the new machines always in different places of the sequence; or we can keep and index that maps the original data to each machine where its replica is stored, allowing for the replicas to be kept in any place. This two approaches would grant that the work of moving replicas would not always be addressed to the same machine, however it would still mostly overload only two machines during each redistribution.

This algorithm allows to easily recover from a machine failure. When this happens we just have to copy back the replica located in the next machine and replicate again the original data in the previous one.

Another good thing about this replication technique, besides simplicity and ease of maintenance, is the fact that it can also easily provide a solution to improve performance of the distributed database. Instead of each machine storing the two buckets (original and replicated data), we can separate them into different machines. Looking into the previous picture 6.1, we could use four machines to store bucket 1, 2, 3, 4 and another four machines to store the replicas 1r, 2r, 3r, 4r. The correspondence between originals and replicas works the same way but this system can use the replicated data to give a performance boost in querying processing.

6.3 Three buckets: original, replica and replica fragment

With the previous algorithm the workload was not so distributed, being only two machines responsible for most of the work. Hence, we looked for an implementation where the workload regarding replication could be more distributed over the whole system. In this approach we also try to move less replicas around whenever the system changes.

In Figure 6.2 we show a representation of an algorithm where each machine is holding three buckets: one with the original data (blue); one with a complete replica of the data from another machine (green); one with a fraction of a replica (grey). The first step is to load the data where there are two possible outcomes. If the

number of machines is even then all the machines only have two buckets with data (blue and green). If the number of machines is odd then all odd machines have their replicas in the next machine and all even machines have their replicas in the previous machine. The last machine, which is obviously odd, has its replica spread among all the other machines (grey bucket).

6.3.1 Adding a machine

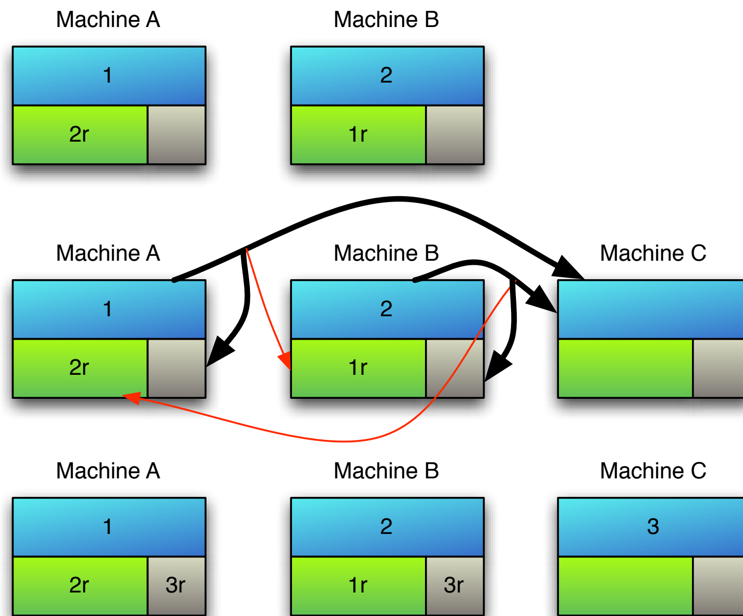

Figure 6.2: Adding a machine with replication using the three buckets per machine approach

In the process of adding a new machine to the system this technique would proceed in the following way. As we can see in Figure 6.2 each machine is divided in three compartments. This schema shows how the distribution of the data evolves as new machines are added. We start with a system of two machines and we want to grow the system to three machines. This way original data from buckets 1 and 2 has to be copied to *Machine C* in order to create bucket 3. This same data is also moved locally in each machine originating the replica of bucket 3 in two fragments. Finally the previous replicas of the data moved have to be updated in bucket 1r and 2r. In picture 6.2 the black thicker arrows represent the data moved to create bucket 3 and replicas 3r and the thiner red arrows represent the messages sent to the previous replicas to update their data.

Adding a 4th machine would imply as before moving original data from every machine to the new one and updating the corresponding replicas. Then the fragment replicas 3r would be moved into the new *Machine D* and the replicas of bucket 4 would be stored in *Machine C*.

We observed that when going from an even number of machines to an odd number we only need to move few data, just a portion of each machine's original to form the new one. However we pay the price when we go from an odd number of machines to an even number. Here we start by also moving a fragment of all the originals to form the new one. Then we have to move an entire replica to the new machine that also comes from all the other machines. Finally we need to replicate the original data that was moved to the new machine.

Overall we see that the amount of data moved in the worst case is 1 original and two replicas, the same as in the n+1 algorithm. However this is done here in a much more distributed manner.

6.3.2 Removing a machine

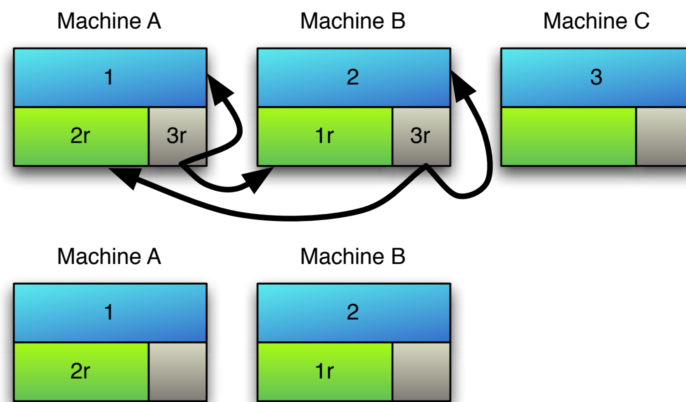

Figure 6.3: Removing a machine with replication using three buckets per machine in a setting with an odd number of machines

This algorithm works linearly, which means that it only allows to remove and add the last machine in the sequence. Like we mentioned in the beginning of the Chapter the fault model adopted is crash-recovery, where any machine can temporally fail, but it eventually returns and is reintegrated in the system. When recovering from a fail, even without keeping any administration, we can easily know where is the replica of each machine and which data has to be replicated again. The only exception

occurs if the system had an odd number of machines and one machine, that not the last, fails. In this case we would have the added work to replicate and fragment the original data from the last machine to all the others again, because we cannot know which specific fragment was erased.

The easiest way to remove a machine is by removing the last one in the sequence of machines. When doing this we may consider two situations depending if the number of machines is odd or even.

Looking at Figure 6.3 we see the algorithm removing a machine from an odd setting. First each fragment of replica on corresponding to the original being removed are locally merged with the original data of all the remaining machines. The fragments that were moved locally have also to be copied to other machine in order to create the respective replicas. Finally the last machine and respective replica fragments can be removed.

The procedure for an even number of machines is a bit different. It works by first evenly distributing the original data present in the last machine among all the others and also creating the respective replicas. Then the replica stored in the last machine is also distributed over the others (except to the one that has the corresponding originals, that is the previous machine). Finally the replica of the machine being removed is deleted and the machine is decoupled.

6.3.3 Fault Tolerance

To think about fault tolerance we have again to consider two different scenarios: odd and even number of nodes. We can see that the original and replicas end up stored in couples, so we can easily locate and restore a failed node. We now that the 1st machine has its replica in the 2nd and the 2nd machine has its replicas in the 1st. This goes on for the whole system, the 3rd machine has its replicas in the 4th and the 4th machine has it replicas in the 3rd.

The one exception is when we have a setting with an odd number of machines and the last one fails. In this case we just need to retrieve the original data from the replicas spread over all the other machines. Also for an odd setting, the failure of any other machine forces the originals in the last node to be replicated again over all the others.

6.4 Completely Distributed

We have discussed a simple technique to replicate data, the n+1 algorithm, and a more distributed and complex technique using three buckets per machine. Now we are going to discuss a way to have all the machines involved in the replication of the original data in each machine.

In this technique the system is perfectly balanced, having all the machines always the same amount of data. Moreover every time a change occurs in the system, only the necessary data to fill the new machine is moved, corresponding to the least data possible to achieve a balanced system. When adding a new machine, $\frac{1}{n}$ records will be moved from every existent machine into the new one. When removing a machine, all its data is distributed among the every remaining machine. Replication is handled in a much more distributed manner comparing to the solutions previously presented. Here when readapting the system all the machines have to participate in the process contrasting with the previous approaches where most of the times only two or three machines would be involved.

We have seen that using algorithms that do not care about the content of the data allow a very efficient work when readapting to a new system configuration. However not having any knowledge on the specific records held by each bucket causes trouble when dealing with replication. We need to guarantee that two fundamental requirements are being fulfilled: first not having the original data and its replica in the same machine; and second in case of a machine failure it should be possible to identify where the duplicates of the data are. It is necessary to identify at least to which original correspond each replica fragment.

The algorithm we present is illustrated in Figure 6.4 where we have a system composed by two machines each one with two buckets. The original data in bucket 1 of *Machine A* is replicated in the bucket 1r of *Machine B* and the original data from bucket 2 in *Machine B* is replicated in bucket 2r of *Machine A*. If we add a new machine to the system 1/3 of the original and replicated data from each machine has to be moved to the new one in order to keep the system balanced. If we remove one node all the data from that machine has to be distributed to the others.

6.4.1 Adding a machine

In Figure 6.4 we can see how the data distribution evolves when the system grows. Here we have a set of keys from 1 to 12 and we start with them distributed and replicated over two machines. Now we want to grow the system to three machines.

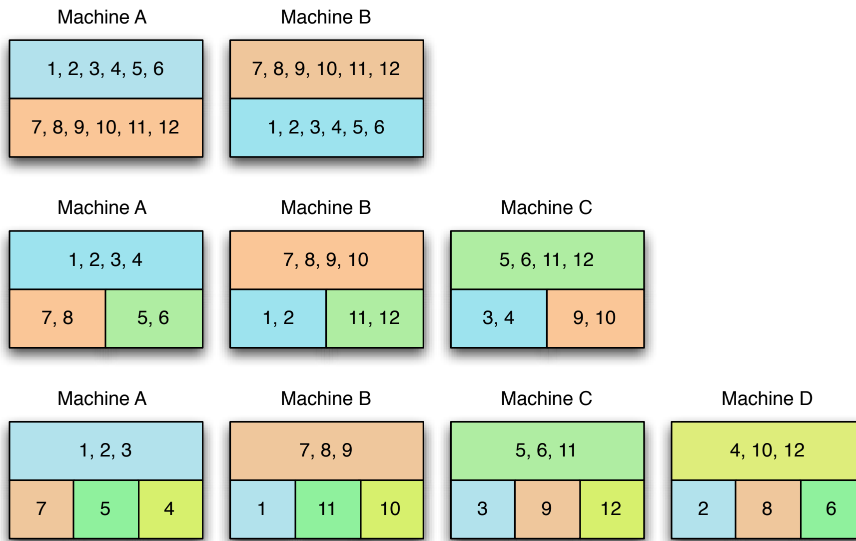

Figure 6.4: Adding a machine with replication using a completely distributed approach

First we have to take $\frac{1}{n}$ keys from the original data in each machine, ship it to the new machine and update the respective replicas. In this case we have to take one third of the original data in Machine A and B, copy it to Machine C and update the replicas in A and B. This fragments of data that were copied are then moved locally to another bucket creating the replicas of C. Finally we equally distribute the records between each replica fragment. In this case we have to move two keys from the replica of Machine A (keys 3 and 4) and two keys from the replica of Machine B (keys 9 and 10), to the new Machine C. We do not care about which data is in each replica fragment, however each replica fragment must be identified to which original it corresponds.

Adding another machine repeats the process. First we take one fourth of the original data in Machine A, B and C and copy it to Machine D, updating the respective replicas in A, B and C. The data fragments that were copied are then moved locally to another bucket creating the replicas of D. Finally we balance the replica fragments corresponding to each machine. Here we have to move the keys 2, 8 and 6 to the new Machine D.

6.4.2 Removing a machine

To remove a machine we just have to do the opposite we did to add one. Figure 6.5 shows the removal of Machine B. We start by merging locally each replica fragment

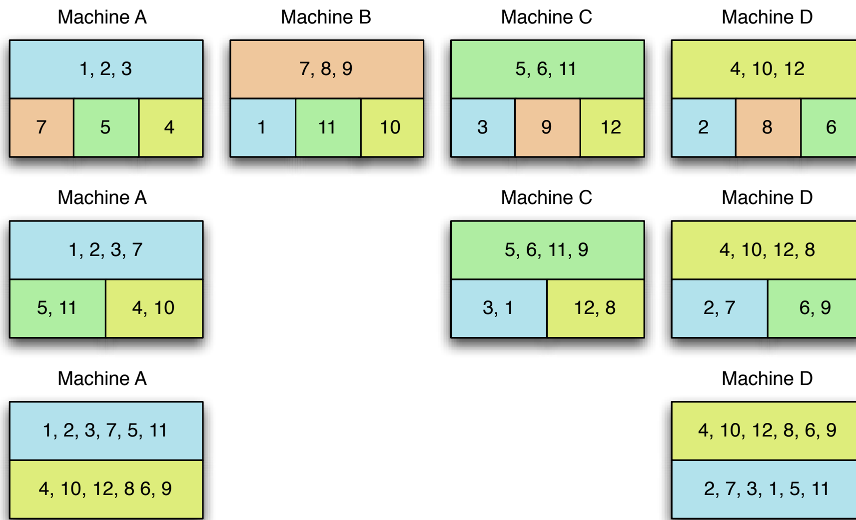

Figure 6.5: Removing a machine with replication using a completely distributed approach

of the data being removed, with the original data in each machine. In the figure we move locally to the originals the keys 7, 9 and 8. After doing this we also copy the keys that were moved to the respective replicas of each machine. Finally we take all the replica fragments in Machine B and we move them to the other machines in the system, making sure that the replica's fragments end up all balanced.

6.4.3 Fault Tolerance

We saw that in this technique when adding and removing machines from the system, we avoid putting original data and the respective replica in the same machines without knowing the exact data that is being moved. This is possible if we update the replicas of the original data moved before redistributing the replicas. Every time there is a change in the system we transfer the least amount of data possible to keep the system balanced.

However in case one machine fails the scenario becomes more complicated. Looking again to the top of Figure 6.5, let's suppose that Machine B fails. In this case we are able to precisely recover the originals that are gone just by putting together all the respective replica fragments. However we have no clue on which data was in the replica fragments of the failed machine. The system would easily be up and running with the correct originals, but it would require the whole data to be replicated again over all machines.

An easy solution to overcome this problem, implies keeping some more administration. It consists on fragmenting the replicas by ranges and keeping a mapping between each range and the machine storing it. This way when a machine fails, the system can easily only copy the fragments of replicas missing in order to restore the machine that is down. With this solution we just need to move the minimum data possible to restore the system.

6.5 Original data and replicas in different machines

Finally one last way to go, would be using different machines for holding the original data and the replicas. This would create two systems that could easily grow and shrink independently. This can lead to a more efficient process of adding and removing machines from the system, but at the same time can also mean waste of computer resources as the machines with the replicated data would just be storing data.

In this approach it is possible to add only new machines for holding the original data, being the replicas handled in different ones. The replicas can even be only handled in the same machine just being moved locally to a different bucket, in order to allow a faster redistribution. Imagine that the user wants to add some more machines in the cloud just to speed up the computation of some hard work queries and afterwords return to the initial set up. This way the data is still kept replicated and the distribution of the original data is done faster and more efficiently.

Algorithms	**Workload Distribution**	**Replicas Relocation**
Two Buckets	Bad	Large
Three Buckets	Medium	Medium
Completely Distributed	Good	Minimum
Independent Systems	Good	Minimum

Table 6.1: Replication comparison in data distribution algorithm

Two important characteristics that differentiates this from the previous algorithms are the simplicity of implementation and independency between original data and replicas. Here we can also think about using different kinds of machines for the original data and for the replicas, being the ones that hold the replicas oriented for storage and aiming primarily for big capacity and fast data access. Moreover

there is also the possibility of using the machines holding the replicas to query data. This may be taken into account in some special cases for examples when queries in the original data are taking too long and the replicas are used as additional query machines.

In Table 6.1 we have a comparison between the algorithms just presented, concerning the distribution of workload and the amount of replicas relocated when the system is modified. We see that the two last algorithms hold the best characteristics where more distribution can be introduced in the replicas and just the minimum amount of data need to be moved when changing the system setting.

Chapter 7

Conclusion

In this thesis we have explored the possibilities of distribution in a database system regarding a cloud setting. Cloud computing has been gaining more and more popularity along the way, providing flexibility, ease of use and low costs to the purchase of computational resources. This service can greatly enable many fields of computer science, going towards a world where all wastes are minimized: computer resources, time, money. However, deploying a distributed database in a distributed system as dynamic as this is not an easy task. We focused on the study of data distribution, finding that it comes with lots of data shipping, which can become complex depending on the distribution algorithm approached.

7.1 Results and Overview

We started this work by investigating further the concept of cloud, also going deeper into virtualization, the technique that makes it possible. We wanted to know what takes to efficiently deploy a database in the cloud. Here, one of the most important issues is how to distribute the data and how to efficiently redistribute it in order to follow possible modifications in the system.

Concerning data distribution we have looked into previous related work that addresses the issue of efficiency and data balancing when readapting the system. While some of them presented plausible solutions for data distribution, none appeared to be designed for the elastic characteristics of a Cloud. The intermittent growth and shrinkage of the cloud size challenges most algorithms in their efficiency regarding data movement.

Some of the database products available in the market also claim the ability to adapt the database to the clients needs in the cloud. However not much is said

further on this topic and the available ways to do table partitioning let us thinking that the data distribution is based on some kind of hash.

Our approach to distribution focused on techniques to redistribute data efficiently in case changes occur in the system setting, aiming at a balanced system. The various algorithms we investigated presented different solutions in terms of data balancing, amount of records that need to be relocated and administration size and complexity. We have chosen to implement the solutions that would readjust the system with less data shipping.

The implementation of four algorithms have put us closer to the issues and results one can find dealing with data distribution. The two most interesting algorithms end up being Division and Consistent Hashing where we achieved a fairly balanced system, with good performance, just by moving the minimum or almost minimum data possible. Two aspects that clearly stand out during our study were: the less data being moved the faster the algorithm performs; the disregard for the location of each records does not add that much of performance. These observation do not come as a surprise, however, for the database system to perform maximally, a well balanced distribution of the data is vital. Avoiding any unnecessary move of data here, means a win in performance of the actual grow or shrink operation.

Introducing replication on top of data distribution showed to be a complicated action when applied as alternative of having replicas at the bucket level. We have looked into four different ways to address replication over distributed data, trying to distribute the workload and move the least records possible. The solutions we presented enabled the system to keep working with a failure of one machine and eventually restore the data.

7.2 Future work

Inserting a database in a cloud environment is a complex subject where many issues have to be addressed. We studied the problematic of data distribution, but even here space is still open for further work and research.

Cluster of Buckets One improvement that can be made approaching data distribution is to consider the addition and removal of clusters instead of single buckets. This would perhaps be a more realistic approach, enabling the system to expand and shrink in a more effective and significative way. Some adaptations to the algorithms would be required in order to minimize the data moved

in each operation.

Minimize Down Time When the system is growing or shrinking and data is being moved, the clients are not able to query the database. One other topic to address would be how to minimize the down time for clients, whenever the system setting changes. Making use of the replicas could be a wise approach.

Cloud Experiments It would be interesting to set up a cloud environment and deploy various instances of MonetDB to simulate a distributed database. Then, with an application, manage the data and perform changes in the system setting to see what are the real costs of shipping data inside the cloud.

Querying Compare the performance of querying data using map reduce and a traditional DDBMS. See if query performance would be lost using a distribution technique like Division that does not keep the location of each record.

Updatable Database In this study we considered a static database where data was not being updated while we were working. If we consider a database where data is being updated, can we still use the same algorithms? What are the changes that have to be made? Which consistency issues have to be addressed?

Appendix A

Gentoo Configuration

The Gentoo operative system was built following the documentation in the respective website and the components chose to be installed were: stage, portage, genkernel, system logger, dhcp client and grub. Before compiling the kernel it was set the following flags:

/etc/locale.gen: en_US ISO-8859-1 en_US.UTF-8 UTF-8

/etc/make.conf: USE="-nls -alsa unicode -fortran -X -qt -gtk -doc -ssl"

The kernel was compiled using genkernel by running the command *#genkernel –menuconfig* which allowed to select what would it have support for. With the OS set it was also installed MonetDB. At this point the system was occupying 2.0GB of disk space and we still wanted it slimmer. So, it was made an inventory of all the main directories and their corresponding sizes which allowed us to see that most of the space was being taken by the directory */usr* (1.8GB), more precisely in */usr/lib* (226MB), */usr/portage* (662MB), */usr/share* (145MB) and */usr/src* (691MB). The content of */usr/lib* is indispensable for the well functioning of the system and MonetDB, so it cannot be removed. On the other hand, in */usr/portage* is kept the Portage tree which could be erased and easily reconstructed at any time only be doing *emerge –sync*. We also had the */usr/share* folder where some important files for applications are kept, not allowing to remove it completely. At last in */usr/src* there is the source code for compiling the kernel, which could be deleted, and even if it necessary in the future it is always possible to get it back using portage. Having this said, the idea was to reduce the OS size, putting it in a state that could easily be updated and maintained.

To see all the applications integrating the system it was installed the *gentoolkit*. This way, by doing *#equery list -i* all the installed packages were shown. Than, after analyzing that list it was possible to sort out some applications that are not going to be needed and which absence will not hurt system. The following components were erased using the command *#emerge -C*:

app-misc/mime-types-8 (mail)
sys-apps/man-1.6f-r3 (manuals)
sys-apps/man-pages-3.22
sys-apps/man-pages-posix-2003a
sys-devel/gcc-4.3.4
sys-kernel/genkernel-3.4.10.906
sys-kernel/gentoo-sources-2.6.31-r6
sys-libs/timezone-data-2009s

Before removing gcc, it was created a package with it by doing *#quicpkg gcc*. This will allow to easily recover gcc at any time. Unfortunately there was still much space being used and as portage was not going to be needed anymore, the content of */usr/portage* was completely removed. In */usr/source* the kernel tree was still there taking 267MB. This tree would only be necessary for installing programs that compile against it. As we do not intend to do this it can also be removed. The next target was to inspect */usr/share* and see what could be disposed. It was deleted the content of the following directories:

9M /usr/share/i18n
11M /usr/share/gtk-doc
16M /usr/share/doc
12M /usr/share/man
3.5M /usr/share/locale

Finally the temporary folders */tmp* and */usr/tmp* were emptied. The resulting OS with MonetDB installed has the size 454MB. Perhaps a smaller system can yet be obtained with a better kernel configuration.

Appendix B

Data Schema

```
CREATE TABLE "ontime" (
    "Year" smallint DEFAULT NULL,
    "Quarter" tinyint DEFAULT NULL,
    "Month" tinyint DEFAULT NULL,
    "DayofMonth" tinyint DEFAULT NULL,
    "DayOfWeek" tinyint DEFAULT NULL,
    "FlightDate" date DEFAULT NULL,
    "UniqueCarrier" char(7) DEFAULT NULL,
    "AirlineID" decimal(8,2) DEFAULT NULL,
    "Carrier" char(2) DEFAULT NULL,
    "TailNum" varchar(50) DEFAULT NULL,
    "FlightNum" varchar(10) DEFAULT NULL,
    "Origin" char(5) DEFAULT NULL,
    "OriginCityName" varchar(100) DEFAULT NULL,
    "OriginState" char(2) DEFAULT NULL,
    "OriginStateFips" varchar(10) DEFAULT NULL,
    "OriginStateName" varchar(100) DEFAULT NULL,
    "OriginWac" decimal(8,2) DEFAULT NULL,
    "Dest" char(5) DEFAULT NULL,
    "DestCityName" varchar(100) DEFAULT NULL,
    "DestState" char(2) DEFAULT NULL,
    "DestStateFips" varchar(10) DEFAULT NULL,
    "DestStateName" varchar(100) DEFAULT NULL,
    "DestWac" decimal(8,2) DEFAULT NULL,
    "CRSDepTime" decimal(8,2) DEFAULT NULL,
```

"DepTime" decimal(8,2) DEFAULT NULL,
"DepDelay" decimal(8,2) DEFAULT NULL,
"DepDelayMinutes" decimal(8,2) DEFAULT NULL,
"DepDel15" decimal(8,2) DEFAULT NULL,
"DepartureDelayGroups" decimal(8,2) DEFAULT NULL,
"DepTimeBlk" varchar(20) DEFAULT NULL,
"TaxiOut" decimal(8,2) DEFAULT NULL,
"WheelsOff" decimal(8,2) DEFAULT NULL,
"WheelsOn" decimal(8,2) DEFAULT NULL,
"TaxiIn" decimal(8,2) DEFAULT NULL,
"CRSArrTime" decimal(8,2) DEFAULT NULL,
"ArrTime" decimal(8,2) DEFAULT NULL,
"ArrDelay" decimal(8,2) DEFAULT NULL,
"ArrDelayMinutes" decimal(8,2) DEFAULT NULL,
"ArrDel15" decimal(8,2) DEFAULT NULL,
"ArrivalDelayGroups" decimal(8,2) DEFAULT NULL,
"ArrTimeBlk" varchar(20) DEFAULT NULL,
"Cancelled" tinyint DEFAULT NULL,
"CancellationCode" char(1) DEFAULT NULL,
"Diverted" tinyint DEFAULT NULL,
"CRSElapsedTime" decimal(8,2) DEFAULT NULL,
"ActualElapsedTime" decimal(8,2) DEFAULT NULL,
"AirTime" decimal(8,2) DEFAULT NULL,
"Flights" decimal(8,2) DEFAULT NULL,
"Distance" decimal(8,2) DEFAULT NULL,
"DistanceGroup" tinyint DEFAULT NULL,
"CarrierDelay" decimal(8,2) DEFAULT NULL,
"WeatherDelay" decimal(8,2) DEFAULT NULL,
"NASDelay" decimal(8,2) DEFAULT NULL,
"SecurityDelay" decimal(8,2) DEFAULT NULL,
"LateAircraftDelay" decimal(8,2) DEFAULT NULL,
"FirstDepTime" varchar(10) DEFAULT NULL,
"TotalAddGTime" varchar(10) DEFAULT NULL,
"LongestAddGTime" varchar(10) DEFAULT NULL,
"DivAirportLandings" varchar(10) DEFAULT NULL,
"DivReachedDest" varchar(10) DEFAULT NULL,

```
"DivActualElapsedTime" varchar(10) DEFAULT NULL,
"DivArrDelay" varchar(10) DEFAULT NULL,
"DivDistance" varchar(10) DEFAULT NULL,
"Div1Airport" varchar(10) DEFAULT NULL,
"Div1WheelsOn" varchar(10) DEFAULT NULL,
"Div1TotalGTime" varchar(10) DEFAULT NULL,
"Div1LongestGTime" varchar(10) DEFAULT NULL,
"Div1WheelsOff" varchar(10) DEFAULT NULL,
"Div1TailNum" varchar(10) DEFAULT NULL,
"Div2Airport" varchar(10) DEFAULT NULL,
"Div2WheelsOn" varchar(10) DEFAULT NULL,
"Div2TotalGTime" varchar(10) DEFAULT NULL,
"Div2LongestGTime" varchar(10) DEFAULT NULL,
"Div2WheelsOff" varchar(10) DEFAULT NULL,
"Div2TailNum" varchar(10) DEFAULT NULL,
"Div3Airport" varchar(10) DEFAULT NULL,
"Div3WheelsOn" varchar(10) DEFAULT NULL,
"Div3TotalGTime" varchar(10) DEFAULT NULL,
"Div3LongestGTime" varchar(10) DEFAULT NULL,
"Div3WheelsOff" varchar(10) DEFAULT NULL,
"Div3TailNum" varchar(10) DEFAULT NULL,
"Div4Airport" varchar(10) DEFAULT NULL,
"Div4WheelsOn" varchar(10) DEFAULT NULL,
"Div4TotalGTime" varchar(10) DEFAULT NULL,
"Div4LongestGTime" varchar(10) DEFAULT NULL,
"Div4WheelsOff" varchar(10) DEFAULT NULL,
"Div4TailNum" varchar(10) DEFAULT NULL,
"Div5Airport" varchar(10) DEFAULT NULL,
"Div5WheelsOn" varchar(10) DEFAULT NULL,
"Div5TotalGTime" varchar(10) DEFAULT NULL,
"Div5LongestGTime" varchar(10) DEFAULT NULL,
"Div5WheelsOff" varchar(10) DEFAULT NULL,
"Div5TailNum" varchar(10) DEFAULT NULL
) ;
```

Bibliography

[1] http://monetdb.cwi.nl/.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[4] D. M. Choy, R. Fagin, and L. Stockmeye. Efficiently extendible mappings for balanced data distribution. 1994.

[5] Todd Deshane, Zachary Shepherd, Jeanna Matthews, Ben-Yehuda Muli, Amit Shah, and Rao Balaji. Quantitative comparison of xen and kvm. 2008.

[6] Irfan Habib. Virtualization with kvm. *Linux J.*, 2008(166):8, 2008.

[7] Sanjay Ghemawat Jeffrey Dean. Mapreduce: Simplified data processing on large clusters. 2004.

[8] A.S.Szalay J.Gray A.R.Thakar P.Z.Kunszt T.Malik J.Raddick C.Stoughton J.vandenBerg. The sdss skyserver - public access to the sloan digital sky server data. 2001.

[9] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. 1997.

[10] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. pages 225–230, 2007.

[11] Witold Litwin. Linear hashing : a new tool for file and table addressing. 1980.

[12] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. Lh* — linear hashing for distributed files. 1993.

[13] Patrick Valduriez M. Tamer Ozsu. *Principles of Distributed Database Systems (2nd Edition)*. Prentice Hall, 1999.

[14] Jun Nakajima and Asit Mallick. Hybrid-virtualization—enhanced virtualization for linux. pages 87–96, 2007.

[15] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of Cloud Computing and Its Applications*, October 2008.

[16] Lucas Nussbaum, Olivier Mornard, Fabienne Anhalt, and Jean-Patrick Gelas. Linux-based virtualization for hpc clusters. 2009.

[17] Ethan L. Miller R. J. Honicky. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. 2004.

[18] Research and Innovative Technology Administration (RITA) Bureau of Transportation Statistics. http://www.transtats.bts.gov/.

[19] Sloan Digital Sky Survey Sky Server. http://cas.sdss.org/dr7/en/.

[20] European Comission Information Society and Media. The future of cloud computing - opportunities for european cloud computing beyond 2010.

[21] VMware. A performance comparison of hypervisors. 2007.

[22] VMware. Understanding full virtualization, paravirtualization, and hardware assist. 2007.

[23] John Paul Walters, Vipin Chaudhary, Minsuk Cha, Salvatore Guercio Jr., and Steve Gallo. A comparison of virtualization technologies for hpc. Washington, DC, USA, 2008.

[24] Sage Weil, Scott Brandt, Ethan Miller, and Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. 2006.