Automatic and transparent selection of cloud storage services

António Manuel da Silva Ferreira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Rodrigo Seromenho Miragaia Rodrigues
Dr. João de Almeida Varelas Graça

Examination Committee

Chairperson: Prof. João António Madeiras Pereira
Supervisor: Prof. Rodrigo Seromenho Miragaia Rodrigues
Members of the Committee: Prof. João Carlos Antunes Leitão

November 2017
Acknowledgments

I would like to thank my family and friends for all their support and for all the fun and joyous moments.
I would also like to acknowledge my dissertation supervisors Prof. Rodrigo Rodrigues and Dr. João Graça for their insight, support and sharing of knowledge that has made this Thesis possible.
Last but not least, to "Estrela", my best friend for the last thirteen years. You were the best cat anyone could have asked for. Thank you.
To each and every one of you – Thank you.
Abstract

Cloud storage services are in demand nowadays. They allow companies to have access to almost unlimited resources without the need for expensive infrastructures. However, choosing the right storage service is a complex task, given the variety of options that exist. Additionally, the right storage service changes with time. Based on the real-world requirements of an Internet company called Unbabel, we present a system that, given a workload description, can choose the best storage service in an automatic and transparent way, and adapt this choice as time passes and the workload changes.

Keywords

Cloud storage; Storage services; Data migration; Automatic;
Resumo

Serviços de armazenamento na cloud são muito procurados hoje em dia. Permitem às empresas accedêrem a uma quantidade praticamente ilimitada de recursos, sem a necessidade de estas manterem infraestruturas dispendiosas. No entanto, escolher o serviço de armazenamento correto é uma tarefa complexa, dada a enorme oferta existente. Além disso, a escolha ideal de armazenamento na cloud varia com o tempo. Com base nos requisitos do caso real da empresa Unbabel, nós apresentamos um sistema que, dada uma descrição duma carga de trabalho, consegue escolher o serviço de armazenamento mais adequado duma maneira automática e transparente, e adaptar esta escolha à medida que o tempo passa e a carga de trabalho varia.

Palavras Chave

Armazenamento na nuvem; Serviços de armazenamento; Migração de dados; Automático.
Contents

1 Introduction 1
  1.1 Motivation ................................................................. 3
  1.2 Problem ......................................................................... 4
  1.3 Objectives ....................................................................... 4
  1.4 Outline .......................................................................... 5

2 Related work 7
  2.1 Cloud Storage ................................................................. 9
    2.1.1 Major providers and current offer .................................... 9
    2.1.2 CloudCmp: Comparing Public Cloud Providers [1] ............ 14
  2.2 Minimizing cost by exploiting cloud pricing schemes ............... 15
    2.2.2 Cost-aware Multi Data-Center Bulk Transfers in the Cloud From a Customer-Side Perspective [3] ....................... 16
  2.3 Minimizing cost by automating the provisioning of storage configurations ........................................ 17
    2.3.1 Traveling to Rome: QoS specifications for automated storage system management [4] ................................................. 17
    2.3.2 MINERVA: an automated resource provisioning tool for large-scale storage systems [5] ..................................................... 18
    2.3.3 Hippodrome: running circles around storage administration [6] ................................................................. 18
    2.3.4 scc: Cluster Storage Provisioning Informed by Application Characteristics and SLAs [7] .................................................. 19
  2.4 Minimizing cost by combining cloud providers ......................... 20
    2.4.1 SPANStore: Cost-Effective Geo Replicated Storage Spanning [8] ............................................................. 20
    2.4.2 CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services [9] ........................................... 21
    2.4.3 Orchestrating the Deployment of Computations in the Cloud with Conductor [10] ..................................................... 22
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>System view with all the different subsystems that constitute it</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of the generations mechanism</td>
<td>39</td>
</tr>
<tr>
<td>5.1</td>
<td>Put operation latency test results</td>
<td>45</td>
</tr>
<tr>
<td>5.2</td>
<td>Get with PK operation latency test results</td>
<td>46</td>
</tr>
<tr>
<td>5.3</td>
<td>Get without PK operation latency test results</td>
<td>46</td>
</tr>
<tr>
<td>5.4</td>
<td>Delete with PK operation latency test results</td>
<td>47</td>
</tr>
<tr>
<td>5.5</td>
<td>Delete without PK operation latency test results</td>
<td>47</td>
</tr>
<tr>
<td>5.6</td>
<td>Create operation throughput test results</td>
<td>48</td>
</tr>
<tr>
<td>5.7</td>
<td>Get operation throughput test results</td>
<td>49</td>
</tr>
<tr>
<td>5.8</td>
<td>Delete operation throughput test results</td>
<td>49</td>
</tr>
</tbody>
</table>
List of Tables
List of Algorithms
Listings
Introduction

Contents

1.1 Motivation ......................................................... 3
1.2 Problem .......................................................... 4
1.3 Objectives ......................................................... 4
1.4 Outline ............................................................ 5
1.1 Motivation

Cloud computing is a type of Internet-based computing in which services are delivered over the internet. It enables companies to consume a computational resource, such as a virtual machine, storage or an application as a utility, rather than having to build and maintain computing infrastructures in house.

Many organizations, nowadays, resort to these cloud services in order to store and process large volumes of data. These services allow, among other things, for almost infinite scalability, without the need for organizations to host large and expensive infrastructures which in turn would entail a big management cost. For example, if an organization needed to handle a large volume of data, instead of going through the trouble of getting the necessary infrastructures ready and maintain them, it could simply choose one of the many cloud storage services available, offered by the many different providers, and delegate the problem of the infrastructure (and other problems as well, like replication policies, availability and security guarantees) to these providers. Another advantage of this kind of services is that it allows the organizations, clients of these services, to potentially save a lot of money, in things such as maintenance of the infrastructures, and time (since they don’t need to build the infrastructures themselves; they “just” choose the service and are ready to go).

Typically, the clients choose the service, or services, that best meet their needs (which is, in itself, a complex task, given the variety of offers that exist in this area and the complex pricing policies of such offers), based on the actual demand and workload, on the goals and requirements, on the type of the compute resource, on the available budget, and other, more specific and individualized variables. However, things normally change and predicting the future is something impossible to do. This is also true in the case of outsourcing computations and data using cloud services. The perfect solution (“perfect” usually meaning the one that meets the requirements while incurring the lowest cost possible) yesterday might not be so good today, and we must adapt as the situation demands in order to fully take advantage of this type of services. For example, data that was previously stored, using some cloud storage service, might become less frequently accessed and thus it might be better, from a financial point of view, to store this data in storage meant for data that is less frequently accessed.

In the real world, what usually ends up happening, in the case of the organizations that resort to these cloud services, is that after an initial assessment of the requirements and of the environment, and consequent selection of the best storage method, the conditions change, and with them the ideal storage method changes as well, and thus a transition must be made in order to adapt to the new conditions. However, this transition between the old services and the new ones might be an onerous task, from a software development point of view, being both time and money consuming.
1.2 Problem

In this paper, we address this problem in light of the real-world use case of Unbabel, an Internet company for crowd sourced translation. This company, currently, stores their data in a MongoDB cluster. The cluster is managed by the company but it is deployed on Amazon EC2 instances, i.e., in virtual machines running in one of Amazon's data centers.

The existing model for data storage worked fine until recently. However, given its recent expansion, the volume of data handled grew as well, and the previous model revealed itself to be expensive. As such, there is a need to change the model used.

With that in mind, it would be convenient to allow for a substantial fraction of their data to be gradually stored in an external database (external meaning not managed by the company itself, i.e., managed by third-parties), using cloud storage services. This external database would work like an archive and should be substantially cost-effective. However, it is still necessary to keep certain data (specifically data that has a higher probability of being accessed in the near future) in the database managed by the company.

1.3 Objectives

The main objective of our work is to address this problem, motivated by the real-world experience of Unbabel, in an automatic and transparent fashion (automatic meaning without the user having to do extra work, when compared to what he already did prior to our solution, and transparent meaning with zero or little changes to the existing client code).

This will be achieved by implementing a new data model. The core principle behind this new data model is to categorize data in one of two possible categories, depending on the usage of that same data, and to store that data in different databases according to that categorization. To support this strategy, we will need to implement a mechanism that is capable of monitoring and classifying the data, automatically and transparently. This will have to be implemented on the client side of the system, since data accesses and manipulations are done, or at least issued, by the application code running in the client's machine.

Since data in this new model isn’t limited to being categorized the same way all the time, data migrations between databases (a consequence of data changing categories) must also be possible. Thus, we have to implement a mechanism that supports these migrations (and these migrations should be transparent to the user). This mechanism will also be implemented in the client's side.
1.4 Outline

In this document we present and discuss, in Chapter 2, various works that have been done aiming at minimizing the cost associated with cloud storage. In Section 3 we present and describe the architecture of our solution, describing its individual constituents. In section 4, we present the system design and implementation, explaining in great detail how we implemented our solution. In Section 5 we present the experimental results of our solution and we reflect upon those results. Finally, we end the document by presenting, in Section 6, the conclusions of our work.
2 Related work

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Cloud Storage</td>
</tr>
<tr>
<td>2.2</td>
<td>Minimizing cost by exploiting cloud pricing schemes</td>
</tr>
<tr>
<td>2.3</td>
<td>Minimizing cost by automating the provisioning of storage configurations</td>
</tr>
<tr>
<td>2.4</td>
<td>Minimizing cost by combining cloud providers</td>
</tr>
<tr>
<td>2.5</td>
<td>Discussion</td>
</tr>
</tbody>
</table>
In this chapter, we present various systems that try to minimize the cost associated with cloud storage. Each of these systems tries to do so on its own, unique way, but, generically speaking, these techniques can be grouped into three categories, depending on how they try to minimize cost:

- by exploiting pricing models of cloud storage services
- by automating the provisioning of storage configurations
- by combining cloud providers

We start the section by giving some insightful information about the current market of cloud storage services and its respective offer, and subsequently we survey the related approaches, structured as separate subsections, one for each of the above categories.

2.1 Cloud Storage

In this section we explore some of the offers that exist today for cloud storage. We start by reflecting about the complexity of cloud storage pricing and analyze some of the pricing models of the leading companies in this area. We finish this section by presenting a tool that can be used to compare different options of cloud storage.

2.1.1 Major providers and current offer

There are three major cloud service providers nowadays - Google, Amazon Web Services (AWS) and Microsoft. Each of these providers offer, among other services, storage as a cloud service, which in turn comprises different options of storage. This variety of storage types arises from the fact that there isn’t a single best solution for everything. Different applications and workloads require different storage and database solutions. Furthermore, since the customers’ storage needs vary with time, so thus the optimal solution. For example, sometimes we want data to be readily available because we frequently access it, other times we might want to archive files, and thus durability is our main concern. With this complexity in storage options comes an equally complex variety of pricing models, with different providers offering different rates and applying different pricing models (discounts may apply after certain thresholds, for example).

To give a more concrete view of this complexity, next we provide several examples of the offer of cloud storage services and their pricing models. Google’s cloud storage offer, through Google Cloud Platform [12], comprises four types of storage (each one represents a unique class of storage): multi-regional, regional, nearline, and coldline. In Google Cloud Storage, whatever the type of storage you choose, you need to create a bucket to store your data. A bucket is an entity that has three properties
that you specify when you create it: a globally unique name, a location where the bucket and its contents are stored, and a default storage type for objects added to the bucket. All storage classes offer the same throughput, latency, and durability. They differ by their availability, minimum storage durations, and pricing for storage and access. Pricing is based on a flat rate for storage and a usage rate for network (calculated in GB), with no minimum fee and a pay for what you use model. Storage and bandwidth usage charges accrue daily but billing is done at the end of the billing period. Multi-regional storage is the most expensive and offers the highest availability, followed by Regional storage, Nearline storage, and Coldline storage (these last two offer the same availability). Because Nearline and Coldline storage are intended for infrequently accessed data, additional rates for data access and early deletion apply. These two types of storage also contemplate a minimum charge period (30 days for the Nearline storage and 90 days for the Coldline storage). All of these storage classes support the same tools and APIs to access data, including the XML API and JSON API, the command-line gsutil tool, the Google Cloud Platform Console, and the client libraries.

Through AWS [13], Amazon’s offer encompasses the following cloud storage solutions: Amazon Simple Storage Service (S3), Amazon Elastic Block Storage (EBS), Amazon Elastic File System (EFS) and Amazon Glacier. Also through AWS, Amazon offers a variety of cloud databases solutions [14]. These solution include Amazon RDS, Amazon Aurora, Amazon DynamoDB, Amazon Redshift and Amazon ElastiCache.

Amazon Simple Storage Service (Amazon S3) [15] is a key-based object storage, meaning it stores data as objects and assigns to each of these objects a unique key (these keys can be any string and can be used to retrieve the data). These objects are stored within resources called "buckets", which have a globally defined unique name and an AWS Region (where you want the bucket to be created). Each AWS account is limited to 100 buckets (though this limit can be increased). A client can store as many objects as he wants within a bucket, and write, read, and delete objects in his bucket. Buckets also allow for versioning of every object they store, although this is not the default behavior and has to be enabled. Objects can be of different classes, depending of the intended use, and have a maximum allowed size of five terabytes (and for a single PUT the maximum object size is five gigabytes). The existing storage classes are Amazon S3 Standard, Amazon S3 Standard - Infrequent Access and Amazon Glacier and each has its intended use cases. Amazon S3 Standard is optimal for frequently accessed data while Amazon S3 Standard - Infrequent Access is intended for data that is accessed less frequently, but requires rapid access when needed. Amazon Glacier is prime for long-term data archiving. To manage and access Amazon S3 resources, clients can use the AWS Management Console, which is a web-based interface, or use one of the many available AWK SDKs. Amazon S3 follows a “pay only for what you use” model, with no minimum fee. Prices are based on the location of the client's Amazon S3 bucket. Storage and bandwidth consumption are calculated in GB. Storage, on a monthly basis, gets cheaper
after the first 50 TB, and when it surpasses 500 TB further discounts are applied. Storage pricing also
varies from storage type to storage type. Standard storage is the most expensive, followed by Infrequent
Access Storage, and finally by Glacier Storage, which is the cheapest.

Amazon Elastic Block Storage [16] is a type of block storage that allows for the creation of storage
volumes and their posterior attachment to Amazon EC2 instances (afterwards, these volumes can be
used as one would use block storage). Upon a volume's creation, one has to decide its capacity and
volume type (it is possible to modify the volume configuration in the future). The maximum allowed size
for a volume is 17 TB and the minimum is one GB. Note that a volume can only be attached to one
instance at a time, but many volumes can be attached to a single EC2 instance, and the volume has
to be in the same availability zone (which is basically a geographic region) as the EC2 instance. EBS
volumes can be created and attached to EC2 instances using the Amazon EC2 console or through the
command line. EBS provides a range of options that allow you to optimize storage performance and cost
for your workload. These options are divided into two major categories: SSD-backed storage and HDD-
backed storage and are used for transactional workloads and intensive workloads, respectively. SSD-
backed volumes are further categorized into Provisioned IOPS SSD (io1) and General Purpose SSD
(gp2), while HDD-backed volumes include Throughput Optimized HDD (st1) for frequently accessed,
throughput intensive workloads and the lowest cost Cold HDD (sc1) for less frequently accessed data.
With EBS, the client only pays for what he uses (similarly to S3). The costs vary from region to region and
based on the type of volume chosen (SDD-based volumes are more expensive then HDD-based ones).
Generally, all types of volumes are charged per GB-month of provisioned storage with the exception of
io1 volumes which are also charged per provisioned IOPS-month (input/output operations per second).

Amazon Elastic File System [17] is a fully-managed (meaning minimal setup and little to no mainte-
nance to the clients), distributed, file storage service for use with Amazon EC2. To create a file system,
the client has various options. The AWS Management Console, the AWS Command Line Interface (AWS
CLI), the Amazon EFS API, or even through various language-specific SDKs, are all possible ways of
creating a file system which then needs to be mounted on EC2 instances. This last step can be achieved
by using NFSv4.1, which is supported by Amazon EFS. Following the same philosophy as the other so-
lutions offered by Amazon Web Services, with Amazon EFS a client pays only for the storage used by
his file system per month, which is measured in GB per month, and there is no minimum fee or setup
cost and no charges for bandwidth or requests. Pricing varies solely based on the region.

Amazon Glacier [18] is a cloud storage service for data archiving and long-term backup. Glacier
stores data in "archives". The data stored in these archives can be of any type. You can upload a
single file as an archive or aggregate multiple files into a TAR or ZIP file and upload as one archive. A
single archive can have a maximum size of 40 terabytes and there is no limit to the number of archives
a single client can have. Upon creation, each archive is assigned a unique archive ID and the content
of the archive is immutable, i.e., after an archive is created it cannot be updated. Amazon Glacier uses "vaults" as containers to store archives. Through the AWS Management Console, a client can check his vaults and through the AWS SDKs he can create or delete a vault (among other operations). Vaults also support access policies. There is a limit of 1000 vaults per AWS account. With Glacier, the client only pays for the storage used, measured in GB per month, and there is no minimum fee (the price varies from region to region). The pricing for archive retrieval depends of the type of retrieval used. Glacier supports three different ways of retrieving archives, which vary in access time and cost requirements. Expedited retrievals allow for quick data accesses to a subset of archives, being the fastest but the most expensive. With Standard retrievals, archives typically become accessible within 3 – 5 hours and is cheaper than the expedited option. The third option, bulk retrievals, is the most cost efficient but also the slowest. To upload or retrieve data, one can use the AWS SDKs or the Amazon Glacier API.

Amazon Relational Database Service (RDS) [19] is a managed relational database service. Amazon RSD allows for the creation of different database instances. Upon the creation of these database instances, we can specify the DB engine and version, the license model, the instance type, the storage type and amount, and the master user credentials. These instances can be created, deleted and managed through the AWS Management Console, the Amazon RDS APIs, or the AWS Command Line Interface. Each client is limited to 40 Amazon RDS DB instances, but additional ones can be requested. Amazon RSD follows the same pricing policy of pay only for what you use with no minimum fee. Billing is done based on DB instance hours, storage (per GB per month), I/O requests per month, Provisioned IOPS per month, Backup Storage, and Data transfer.

Amazon Aurora [20] is a relational database engine. Aurora is one of the possible database types that we can use with Amazon RDS. The minimum storage capacity of Aurora is 10GB and the maximum is 64TB. To launch an Aurora DB instance, we can use the AWS Management Console or the Amazon RDS APIs. Pricing can be done by an on-demand policy, paying for the database usage by the hour with no long-term commitments or upfront fees, or a reserve policy, paying in advance for a fixed amount of storage for a fixed period of time and is cheaper that the on-demand option. With the latter, the client commits to the entire duration of the term and pays an upfront charge (the client is billed for each hour of the term regardless of whether any usage has occurred).

Amazon DynamoDB [21] is a fully managed NoSQL database service. To use this service, a client can use the AWS Management Console or the Amazon DynamoDB APIs. Amazon DynamoDB’s data model consists of items, which have attributes, and that are stored in tables. Tables, which are collections of data items, and since DynamoDB is a non relational DB service, are schema-less (i.e. the data items in a table need not have the same attributes or even the same number of attributes), with each table having to have necessarily a primary key. The primary key can be a single attribute key or a “composite” attribute key that combines two attributes. The attribute(s) designated as a primary key must exist for
every item, since primary keys uniquely identify each item within the table. An Item is composed of
a primary or composite key and a flexible number of attributes. There is no explicit limitation on the
number of attributes associated with an individual item, but the aggregate size of an item, including all
the attribute names and attribute values, cannot exceed 400KB. Each attribute associated with a data
item is composed of an attribute name and a value or set of values. Individual attributes have no explicit
size limit, but they count for an item’s total size and thus are bounded by the 400KB maximum value.
Pricing follows a pay only for the resources you provision with no minimum fee policy. Billing wise, the
client is charged by each DynamoDB table’s hourly throughput capacity (after the free tier is exceeded).
In addition, DynamoDB also charges for indexed data storage as well as the standard internet data
transfer fees.

Amazon Redshift [22] is a fully managed data warehouse that organizes the stored data by column.
A client can use the AWS Management Console or the Amazon Redshift APIs to create an Amazon
Redshift data warehouse cluster. Upon the initial configuration, the client specifies the preferred Avail-
ability Zone (optional), the number of nodes, node types, a master name and password, security groups,
preferences for backup retention, and other system settings. There are two possible configurations for
Redshift’s data warehouses. The single node one and the multi-node one. The first is faster to setup
and can scale up to a multi-node configuration. The multi-node configuration is a more complex con-
figuration that requires a leader node that manages client connections and receives queries, and two
compute nodes that store data and perform queries and computations. The nodes themselves can by
of two types. Dense Storage (DS) nodes or Dense Compute nodes (DC). Dense Storage nodes allow
for the creation of very large data warehouses using hard disk drives (HDDs) and are cheaper than the
DC type. Dense Compute nodes allow the creation of very high performance data warehouses using
fast CPUs, large amounts of RAM and solid-state disks (SSDs). An Amazon Redshift data warehouse
cluster can contain between 1 and 128 compute nodes, depending on the node type. Pricing for Ama-
zon Redshift can follow an on-demand policy of a reserve in advance policy. The first has no upfront
costs, no commitment minimum term, and the client is charged an hourly rate based on the type and
number of nodes in his cluster. The latter offers three different options. No Upfront, where there are no
upfront costs, and the client commits to pay hourly over the course of one year at a 20% discount over
on-demand. Partial Upfront, where the client pays a portion of the reserved resources upfront, and the
remainder over a one or three year term. The discount over on-demand is up to 41% for a one year term
and up to 73% for a three year term. All upfront where the client pays for the entire reserved resources
(one or three years) with one upfront payment, with a discount of up to 42% for a one year term and up
to 75% for a three year term compared to on-demand.

Microsoft offers cloud storage through Microsoft Azure [23]. Similarly to Google and Amazon, it
follows a “pay only for what you use” plan, with no upfront costs and no termination fees. Storage prices
vary based on the type of storage chosen, the redundancy options, and the amount of data stored, on a monthly basis (after the first 100 TB prices lower, and after the first 1000 TB prices get even lower). Pricing for data over 5000 TB/month isn’t specified, with the client having to contact Microsoft directly. Pricing also varies with the type of storage chosen.

2.1.2 CloudCmp: Comparing Public Cloud Providers [1]

Almost all public cloud providers differ in their approach to infrastructure, virtualization, and software services. From the pricing models to the services offered, each cloud provider has its unique scheme of doing things. This diversity in cloud services makes it difficult to choose the right one, i.e., usually the cheaper one who can meet our requirements. We need to be able to, given two cloud providers, compare them and tell which one performs the best. It seems pretty straightforward but considering that each cloud provider is different in its own way, finding a common criteria to compare them is challenging.

CloudCmp is a tool that compares the performance and cost of cloud providers. It helps the customer choose which cloud provider is best suited for his needs, by providing cost and performance information about several providers. The customer then decides which one suits him the best. CloudCmp is also useful for the providers. It tells them which of their services are under-performing, compared to their competition (it actually goes further and explains why those discrepancies exist between providers, enabling them to improve their own services). One important characteristic of CloudCmp is fairness, i.e., it uses the same benchmarks, workloads, and metrics to compare all the different providers. This in turn has the disadvantage of lowering the number of services tested because only services offered by all the providers can be compared. The common functionalities compared include elastic compute cluster, persistent storage, intra-cloud network, and wide-area network. The metrics used for each one of these are: benchmark finishing time, cost, and scaling latency for the compute clusters; operation response time, time to consistency (the time between when a datum is written to the storage service and when all reads for the datum return consistent and valid results), and cost per operation for the storage services; path capacity, and latency for intra-cloud networks; optimal wide-area network latency for wide-area networks.

CloudCmp also tries to limit measurement cost. It trades a more complete comparison (one in which all cloud providers are continuously measured across all their data centers) for a lower measurement overhead and reduced monetary costs. It also restricts the cloud providers that are compared (compares the ones with more customers, i.e., popular, and that represent different models). It’s important to note that CloudCmp complies with cloud providers’ use policies, making sure not to disrupt cloud infrastructures and other customers applications.
2.2 Minimizing cost by exploiting cloud pricing schemes

In this section we present two different systems that try to minimize the cost of cloud storage, from a customer's point of view, by taking advantage of some particular characteristics of the pricing models applied to this type of cloud services.


In cloud computing, resource provisioning can be done in two different ways. Either using a short-term on-demand plan or a long-term reservation plan. Usually, the on-demand plan is charged in a pay-per-use basis. This type of plan can be bought anytime for short periods of time. The reservation plan is charged once, e.g., one a year, with the clients reserving resources a priori. Both of these plans have their advantages and disadvantages. The short-term on-demand plan allows for a dynamic provisioning of resources. If, for example, I, as the customer, suddenly need more storage space, I can simply buy more using this type of plan. The disadvantage of this type of plan is the price when compared with the long-term one, with the short-term plan being more expensive. The disadvantage of the long-term reservation plan comes from the very fact that makes it cheaper, that is, the advancement of resources provisioning. If a client is reserving resources for a time in the future, he cannot possibly know if those resources will be enough (or they can be an overkill, resulting in underusage of resources). This is called the underprovisioning problem (and the overprovisioning problem, respectively). We could say that a simple way to solve the underprovisioning problem is to simply use an on-demand plan when we need more resources. This however is more expensive. Thus, it is on the customer's best interest to minimize on-demand plans, since these are more expensive plans, and underprovisioning and overprovisioning of resources, so that the total cost of resource provisioning is minimized.

To address the problem of optimal cloud resource provisioning (OCRP) an algorithm is proposed which minimizes the total cost for provisioning resources in a certain time period. This algorithm takes into account resources offered by multiple cloud providers. A stochastic programming model is formulated, representing the problem of optimizing resource provisioning (the objective function is to minimize the cloud consumer's total provisioning cost). Considering the system model of cloud computing environment consisting of four main components, namely cloud consumer, virtual machine repository, cloud providers, and cloud broker, in which computing resources have to be provisioned from cloud providers, in order to meet the cloud consumer's jobs demand, and to obtain these resources, consumers first create VMs integrated with software required by the jobs, which are then stored in the VM repository. The cloud broker, located in the cloud consumer's site, is then responsible for provisioning resources for hosting the VMs (it can also allocate the VMs originally stored in the VM repository to appropriate cloud providers). The OCRP algorithm is implemented by this cloud broker. The broker receives a set of VM
classes (each one representing a type of job) that require, each, an amount of resources for running.

With this resource requirements, the broker can reserve resources, from cloud providers, to be used in the future according to the actual demand (additional resources can be provided if the reserved ones aren’t enough, through on-demand plans). These two types of resource provisioning plans (on-demand and reservation) are used in different time intervals, known as provisioning phases. Three of these phases exist: reservation, expending, and on-demand. In the reservation phase, the broker unaware of the consumer’s actual demand, provisions resources in advance, using the reservation plan. In the expending phase, the price and demand are realized, and the reserved resources can be used. If it is observed that the resources aren’t enough, the broker can use the on-demand plan and buy more resources, thus beginning the on-demand phase.

2.2.2 Cost-aware Multi Data-Center Bulk Transfers in the Cloud From a Customer-Side Perspective [3]

Many cloud applications rely on the ability to geo-distribute data across multiple data centers as a means to increase their performance (by lowering latencies), availability and reliability (by backing up data and replicating it). Other applications, such as software distribution, virtual machines cloning, distributed databases, and data warehousing, depend, as well, of this ability to geo-distribute data. And at the center of this feature, lies the large volume of data that must be transferred between data-centers. With the propagation of such volume of data, comes a very high cost, consequence of the expensive inter data center bandwidth.

CloudMPcast is an overlay system that executes in each data center where the application is deployed. It exploits two particular aspects of the models that Cloud Service Providers use to charge their customers, with the goal of minimizing cost (from a CSPs customer perspective). This is done without compromising transfer times (so that applications’ performance isn’t affected). Considering that data transfers are charged by volume of data transferred, by counting the GBs each customer transfers from each data center during a month and multiplying it by its cost, there are two aspects that allow CloudMPcast to exploit pricing policies. The rates that apply are based on the location of the source data centers (and in the Amazon EC2 case, also on the location of the receiver data centers) and on the total volume of data that a customer transfers in a month per data center (when in a data center the volume of data transferred exceeds a certain threshold, discounts are offered for that data center).

The main component of CloudMPcast is the routing planner, which is responsible for finding the most cost-efficient route, while monitoring the transfer time, for any request. This problem is a specific case of the multicast problem in which the solution has to, at least, meet the transfer time of the trivial solution (that in which the source data center transmits directly to all the destinations), and the edge costs are not fixed (they depend on the volume of data transferred). The routing planner models this
problem as an integer linear program. It starts by modeling it without considering volume discounts, while trying to minimize the cost (given by the objective function). It searches for possible routes, only considering paths that are, at least, as good, in terms of end-to-end transfer times, as the trivial solution. When volume discounts are applicable, it has to take into account that the cost for a given request now depends on both the routing decisions made for prior requests (because it has to consider the potential data centers that are “discounted”) and the current volume to transmit.

2.3 Minimizing cost by automating the provisioning of storage configurations

In this section we present four different systems that try to minimize the cost of cloud storage by observing that storage system design and management is a very complex task, and by trying to automate certain aspects of this task.

2.3.1 Traveling to Rome: QoS specifications for automated storage system management [4]

Storage systems are getting bigger and bigger. Data centers with hundreds or thousands of logical volumes and file systems, hundreds of terabytes of disk drives, and tens to hundreds of gigabytes per second of storage traffic will be nothing new in a nearby future. With this increase in size, these systems get more complex and, as such, their design, support and management get harder. Add to this the fact that human intervention usually results in errors and the scarcity of competent system administrators, and we get an area that is perfect for automated design and management techniques, with huge potential for cost savings. A key point of such techniques is the representation of storage system QoS.

Rome is the information (or object) model that the Storage Systems Program at HP Laboratories has developed to tackle this challenge. It is described as an “information bus” used to “tie together our storage system design, configuration, and monitoring tool”. It works like a common language used to described everything important about storage systems, like, for example, the workloads and the QoS goals of the system. The Rome information model is based on a very simple idea: everything is an object, and attributes are added to those objects to describe additional properties. For example, objects could represent disk arrays, or part of a workload and attributes could represent internal components, such as I/O controllers on a disk array. Each object is introduced by a single declaration, has a unique name, and has a set of attributes. In turn, these attributes are modeled as objects in their own. The Rome object model has two layers. The lower one is known as the shallow semantics and occupies a middle ground between the syntax of the representation language and the deep semantics. Shallow-
semantic objects are objects that are well standardized across the different tools, such as, the cost. The other level is known as deep semantics and is responsible for defining the remaining objects types and their attributes. The interpretation of these objects varies from tool to tool.

2.3.2 MINERVA: an automated resource provisioning tool for large-scale storage systems [5]

It is difficult to design enterprise-scale storage systems. From the hundreds of host computers and storage devices to the tens of thousands of disks and logical volumes, the complexity that arises from all the possible choices is enormous. Now imagine that these systems are traditionally designed by hand, and we can easily understand why it is usually a slow (from several weeks to several months) and tedious process, resulting, most of the times, in solutions that perform poorly or are over-provisioned. Having a way of automatically designing storage systems could lower the costs of storage systems design, by quickly providing a well provisioned design.

MINERVA is a suite of tools for designing storage systems automatically. Its main objective is to rapidly design a storage system that meets workload requirements as cost effective as possible. MINERVA addresses the complex problem of designing storage systems by dividing it into three steps. First, it chooses the right set of storage devices for a particular workload, i.e., it selects a set of devices that can satisfy the resource requirements of the workload. Second, it configures those devices. Thirdly, it maps the data onto the devices. After these three steps, a final optimization step takes place. Unused resources are cut of and a re-assignment is performed, attempting to balance the load across the remaining devices. MINERVA takes as input descriptions of the workload of the system being designed and of the capabilities of the available storage devices. It outputs an assignment, i.e., a selection of storage devices and their correspondent configurations, and a mapping of the pieces of the workload onto those storage devices. The two key ideas behind MINERVA are fast analytical models and heuristic search techniques. The heuristics ensure that the design space doesn’t have to be exhaustively searched, and the models allow for a quick evaluation of each candidate solution chosen by the heuristics, determining if the proposed solution satisfies all workload requirements and doesn’t overload any device.

2.3.3 Hippodrome: running circles around storage administration [6]

Storage systems at the enterprise scale are extremely difficult to manage. From their size, to all the possible configuration choices, managing this kind of systems, at this level of magnitude, is complex. Additionally, if we consider that, traditionally, these systems are configured and managed by experts, that are scarce and expensive, in a process that is time-consuming and error prone, we can understand
why storage management is challenging.

Hippodrome is a system that automates the design and configuration process of storage systems. It follows an iterative approach to storage system management, while trying to minimize the resources used. The approach consists of three steps: analyze workload, design new system, implement design. Hippodrome is composed by four different components which work together in order to implement this loop. The analysis component, the performance model component, the solver component and the migration component. The analysis component is responsible for the first step of the loop and it starts by analyzing a running system in order to learn about its workload. It receives as input a trace of the workload’s I/O references and a description of the storage system, and outputs a summary of this trace. To implement the second step of the loop, both the performance model and the solver components are used. The performance model takes as input the output from the analysis model and a potential storage system design from the solver. It outputs the utilization of each component in the storage system. The solver receives as input the workload description generated by the analysis component, efficiently searches the large space of storage system designs, and outputs a balanced (minimal utilization variation across the resources), valid and minimal (as few resources as possible) design that fulfils the workload’s performance requirements. Finally, the migration component is responsible for implementing the last step of the loop. It takes as input the design output by the solver, and changes the existing configuration to the new design, thus migrating the old system to the new design.

2.3.4 scc: Cluster Storage Provisioning Informed by Application Characteristics and SLAs [7]

Nowadays, application providers can choose from a myriad of different storage options to provision the infrastructure for cluster-based applications. This provisioning is usually based on rules of thumb and best practices. Applications are roughly categorized and deployed on clusters, and when their loads increase, more storage is simply added. This kind of “one size fits all” approach makes it so that these deployments aren’t really that optimized for the applications, failing to take advantage of the diversity of available storage services, and resulting in a variety of inefficiencies. As these deployments get bigger, the inefficiencies start to pile up, increasing the overall cost of the deployments and inflating expenses.

scc is a storage configuration compiler that automates cluster configuration decisions based on formal specifications, while seeking to minimize cost. It takes as input a model of application behavior (its implementation and the target workload) provided by the application’s developer, information about the available hardware provided by the infrastructure provider, and application performance metrics (i.e. Service Level Agreement). For SLA specification, two types of application classes are considered, batch and interactive. The former requires the SLA to have two attributes, the job size and the required execution time. The latter requires that each type of request has an associated SLA. For the hardware
specification, three types of elements are considered: storage units, CPU cores, and servers (each of these elements has a cost attribute associated). The implementation of the application is described in terms of storage and compute components, and their interaction. Properties of the expected workload can be provided by the application's developer, by measuring the compute and I/O characteristics of an application's components. After receiving these inputs, scc proceeds to determine a cost-effective cluster configuration that satisfies the SLAs. To do so, scc has to determine the architecture of the cluster (for each dataset determine the type of media it should be stored in) and to identify its respective scale (determined by the number of servers, storage units, and CPUs). scc exhaustively searches the configuration space for the optimal configuration, representing each point in this space by the assignment of storage unit types to datasets. scc implements a procedure that receives as input an assignment of storage types to data sets and outputs a cost-effective set of resources to meet the target SLAs. Using this procedure as it traverses the search space, scc can determine a cost-effective configuration.

2.4 Minimizing cost by combining cloud providers

In this section we present four different systems that try to minimize the cost of cloud storage by employing a model where, instead of only considering one cloud service provider, different providers are considered at the same time, and their respective services are all viewed as if they were provided by only one provider.

2.4.1 SPANStore: Cost-Effective Geo Replicated Storage Spanning [8]

Nowadays, several cloud providers offer storage as a service and they provide it in several geographically distributed data centers. As such, applications should be able to exploit these distributed locations for storage to offer low-latency to their clients. However, in most cases, these services offer an isolated pool of storage per data center, meaning that there isn’t a global view of these storage services. For example, if an application wants to replicate some data to all the data centers, the application itself is responsible for issuing a PUT to each data center. But sometimes we don’t need, or want, to replicate data to all the data centers. Maybe we’re only interested in a specific region, or want to prioritize the cost. It’s up to each application to reason about how data should be replicated.

SPANStore is a key-value store that tries to address this problem, by presenting a “unified view of storage services in geographically distributed data centers”, thus simplifying application development, while trying to minimize the cost associated with latency-sensitive applications (and at the same time respecting applications’ latency goals and requirements for consistency and fault tolerance). To achieve this “unified view”, SPANStore sits between the applications and the distributed storage services, serving PUTs and GETs, enabling applications to interact with only one storage service, SPANStore itself, which
in turn uses several other distributed storage services in the background.

With the objective of minimizing the cost, three principles rule the design of SPANStore. First, instead of limiting itself to the data centers of just one cloud provider, SPANStore uses data centers of different providers. This has two advantages. First, it allows for lower latencies because different providers have data centers located in different places (and so, by using data centers from different providers, SPANStore is capable of covering a wider and denser area), and it allows SPANStore to exploit to the fullest the price differences between the different providers (because usually the cost of these storage services differ from provider to provider); Second, for every object that it stores, SPANStore determines where to replicate it and how to perform this replication, based on that object’s workload, i.e. accesses, the latency, consistency and fault tolerance requirements of the application and the pricing of the storage services; Thirdly, SPANStore minimizes the compute resources necessary to offer a global view of storage.

2.4.2 CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services [9]

In almost every application, user-perceived latencies are crucial. Even the smallest of delays can mean a significant loss in profit. Higher user-perceived latencies lower revenue, which naturally makes the task of lowering them one of the utmost importance. This becomes even more important if we consider the fact that, nowadays, both fetching and storing data, using cloud services, are associated with high latency variance (caused mostly due to isolated latency spikes). These high latency events are problematic for two types of applications. For applications where even 1% of the traffic corresponds to a large volume of requests, and for applications where a single request requires the application to fetch several objects and user-perceived latency is constrained by the last object fetched.

To address this problem, CosTLO employs a very well known strategy for reducing latency variance, augmenting GET/PUT requests through redundancy. Although using redundant requests to reduce latency variance isn’t nothing new, CosTLO is the first system to combine different forms of redundancy while trying to minimize cost. This combination of different forms of redundancy has the consequence of producing a large number of possible combinations. This, in conjunction with the complex architecture of cloud storage services and all the different pricing policies, makes choosing the right configuration a difficult task. In order to address this complex problem, CosTLO estimates the latency offered by any configuration, as well as the cost associated with the configuration, and searches the configuration space for a cost-effective one which meets the desired latency variance. This search starts as if CosTLO wasn’t being used, i.e. by issuing a single request to the closest data center. Afterwards, CosTLO iterates through configurations in the increasing order of cost. From a pool of possible configurations, it considers the minimum cost one (this cost is computed as the sum of expected costs for storage, VMs,
requests, and bandwidth). If this lowest cost configuration doesn’t satisfy the intended latency variance goals, it’s rejected and all of it’s neighbours (configurations that differ by one parameter) are added to the pool of possible configurations. This process is repeated until a configuration that satisfies the latency variance requirements is found.

2.4.3 Orchestrating the Deployment of Computations in the Cloud with Conductor [10]

Cloud computing allow for almost unlimited computation resources to programmers. This makes it so that organizations don’t have to invest in IT infrastructures to perform jobs locally. They simply move it to the cloud. This ease of scalability isn’t, however, without some challenges. When some computation is deployed in the cloud, it is necessary to account for the time it takes to finish and its respective cost. This in conjunction with a never-ending offer of different services with different prices, that are always fluctuating, and different performance characteristics, results in a huge number of possible choices. Additionally, consider hybrid deployments (a combination of cloud computing and local computing), and the difficulty that is to predict the performance characteristics of different services, and we get a highly complex problem.

Conductor is a system that supports cloud customers, enabling them to make better decisions regarding which cloud services to select, and orchestrates the execution of MapReduce computations on the cloud automatically. It frees the customers from having to reason about the different services and their trade-offs, by devising an execution plan, and deploying it. For each computation, optimization goals, such as minimizing monetary cost or completion time, are specified by the customers. These goals influence which services are used and how they’re used. Conductor takes as input a computation, a set of cloud services that can be used, and a set of optimizations goals. With these inputs, Conductor starts by finding the best execution plan that fulfills these requirements. This is done by modeling both the computation and the set of cloud services available (and their cost and performance), using a dynamic linear programming. Using the CPLEX solver and giving it as input the linear program modeled previously, it automatically determines an optimal execution plan. This solver is however bounded in its solving time, with a maximum of three minutes. After this time, the best solution computed so far is chosen. This solution is then deployed and its execution monitored. If deviations are detected, a new plan is computed and deployed using the same strategy but with updated conditions.
2.4.4 CDStore: Toward Reliable, Secure, and Cost-Efficient Cloud Storage via Convergent Dispersal [11]

Cloud storage allows for organizations to host backups off-site in a cost efficient manner. However, this naturally raises some concerns, from the clients’ perspectives. Outsourcing all the data in a single cloud raises some reliability concerns, such as vendor lock-in (imagine the cloud storage provider going out of business and all your data disappears), and some security concerns as well. Your data is now handled by third-parties, which makes confidentiality and integrity of such data important aspects to consider. Combining multiple cloud storage services improves some of these aspects. Various independent cloud services means no vendor lock-ins and no single points of failure. Achieving this redundancy of outsourced data while being concerned with its confidentiality and integrity, can be accomplished using secret sharing algorithms. However, traditional secret sharing algorithms don’t allow for storage savings by means of deduplication (which is usually used for reducing costs since backups contain substantial redundant information).

CDStore is a multi-cloud storage system that aims to provide a unified view of cloud storage while striving for reliability, security, and cost minimization. It follows a client-server architecture. CDStore targets backup workloads, which usually contain significant identical content (making deduplication useful). The core idea behind CDStore is a secret sharing scheme called convergent dispersal, which consists of a secret sharing scheme that instead of using random inputs, as it is traditionally done, uses deterministic cryptographic hashes derived from the original data. Only someone who knows the full data set can compute these hashes. CDStore tolerates cloud failures by reconstructing the original secrets and then rebuilding the lost shares. CDStore ensures confidentiality and integrity of data as long as a tolerable number of clouds aren’t compromised. Deduplication is used for reducing costs. It works by identifying identical pieces of data and storing only one of them, referring all the identical ones to that one copy (through small-size references).

2.5 Discussion

To the best of our knowledge, there is no prior work that focuses on automatically and transparently selecting the most appropriate cloud storage service, based on the access patterns of the data by the applications, and on dynamically adjusting this choice as time goes by and the environment changes, while focusing solely on minimizing cost. In this aspect, we believe the design we propose is unique and original, providing some insightful information, concretely on the particular use case of Unbabel.

All of the research presented in this chapter focused on trying to reduce cost associated with cloud storage services, which is one of our objectives for this work. Nevertheless, the vast majority of this research employs complex models and algorithms that aren’t that interesting to the work that we aim
to produce (either too complex and/or their scope is usually larger than what we want, i.e. they focus on optimizing more aspects than the ones that we are interested in, and thus they become, at least for us, unnecessarily complex). Another shortcoming of existing proposals is that most of them are only demonstrated through a research prototype, which is often not available as open source, and/or is far from being production-grade code.

CloudMPcast and the OCRP algorithm, that use a strategy of minimizing cost by exploiting cloud pricing models, are far too complex to be relevant in our case, using complex mathematical models to solve the cost optimization problem. Thus, the possible benefits do not justify the overall complexity of the resulting systems. However, they both provide some interesting features, like, for example, OCRP’s optimization of resource provisioning, that exploits the two possible ways for resource provisioning, and CloudMPcast’s exploit of bulk transfers.

Rome, MINERVA, Hippodrome, and scc are all systems that try to minimize cost by automating the provisioning of storage configurations. Even though they all provide some interesting information and useful insights about storage systems, they fall off of our intended scope for this work. They focus mainly on automatically designing storage systems, which isn’t our objective. These systems are, as well, highly complex.

The approach of minimizing cost by combining cloud providers is exploited by four different systems, SPANStore, CosTLO, Conductor, and CDStore. The combination of cloud providers seemed a desirable feature at first but, to simplify the overall complexity of the solution, we will only consider one provider, even though it is possible for us to migrate from one provider to another, from time to time (however we will only use one provider at a time). Furthermore, each of these systems focuses on optimizing other aspects than just cost, which in turn entails a greater complexity.

SPANStore, in particular, is the one that gets the closest to our intended solution. Its architecture is similar to what we aim for, since it sits between the applications and the storage mechanisms, however there are some factors that prevent it from being a good match for our scenario. As previously mentioned, we will only consider one cloud storage provider. Furthermore, it also tries to minimize latency in data accesses, which results in an even more complex solution, that relies on inputs correctly provided by the programmers.

CloudCmp is an interesting tool that allows customers to compare different cloud providers. It could be used, in theory, to choose the best option of cloud storage, however this comparison must be handled by the customers themselves, who have to compare the results output by CloudCmp (this tool only provides the measurement results). Since the comparison has to be done by the programmer, we will not be considering this work for our solution (we strive to take the burden off the programmer as much as possible).
3 Architecture overview

Contents

3.1 MongoDB ................................................................. 27
3.2 Client ................................................................. 28
3.3 Data migration ......................................................... 29
In this chapter, we will present the architecture overview of our final solution to the automatic and transparent selection of cloud storage services problem. We will describe and explain the main parts of our solution, including the different systems that make up the entire system, namely the clients, the Mongoengine DOM, the PyMongo driver and the mongoDB databases.

We start off the section by presenting an overview of the system as a whole, with its different components, some making up the client and others making up the database with which the client communicates, and by explaining how these different components interact between themselves to work like a concrete and coherent system (here we use the term "system" to refer to the totality of all the systems that we are working with, either it being the client or the database. We will continue to use this term throughout the document to refer to all the systems that we are dealing with).

We end the section by tying all the pieces together and by giving a general idea, without getting into the technical details (we save that for the next section), of how we modeled our solution.

### 3.1 MongoDB

This is the database of choice for our system. MongoDB is a document-oriented database. Our system runs with two traditional mongoDB instances, each with a purpose of its own. One of the instances is responsible for storing data that is deemed "live", meaning that is data that was manipulated recently. We designate this mongoDB instance by "live system", alluding to the very nature of the data that it stores (the same logic process is used to name the other mongoDB instance). The second mongoDB instance is responsible for storing data that is considered "old", i.e., data that was accessed for the last time a long time ago. We named this instance "archival system", since its function is pretty much to serve as an archive to mongoDB's documents. From now on, we will use the names "live system" and "archival system" when referring to the first and second instances of MongoDB, respectively. Data stored in the live system cannot be present in the archival system at the same point in time, and vice versa, meaning that these two databases are mutually exclusive when talking about the data that they store. This also means that data that isn't in either one of these databases isn't considered to be stored in our system (there are no other data storages in our system).

It's important to note that even though mongoDB offers different solutions regarding data storage and management, our instances of MongoDB are simple databases. We also don’t make any assumptions regarding the deployment of these databases in the cloud but, generally speaking, the live system would be deployed using a cloud storage service that allows for fast data accesses and the archival system would be deployed using a cloud storage service that allows for very cheap data storage as long as that data isn’t accessed often, and even though our solution works with two running instances of MongoDB, its architecture could be easily adapted to work with more.
3.2 Client

The client side of our system is comprised of three parts. The Mongoengine, the PyMongo and the client application. All of these components run locally, on the client machine. We will be looking, further ahead in the document in subsequent sections, mainly into the Mongoengine component since that was the place where we primarily focused our attention and efforts.

3.2.1 Application

Each client communicates with both the live system and the archival system to read and write data to the system, where the exact system that is contacted depends on the decision made by the underlying layers that we explain next, depending on the data that it is handling. This communication is transparent to the application itself and is handled by the Mongoengine DOM and the PyMongo driver. All that is visible to the application is the interface exported by the Mongoengine, which allows it to work with the MongoDB databases. This means that, to the application, data that is stored in the live system is, for all intents and purposes, treated the same as data stored in the archival system. This is possible because Mongoengine and PyMongo do all the heavy lifting of handling the logistics behind the management of this data to one of the two databases.

3.2.2 Mongoengine

This is a Document-Object Mapper (DOM) for working with MongoDB from Python. It exports an interface, in python, which allows for applications to use different functions to manipulate (create, delete, update, query, etc) data from/into MongoDB.

Our focus was primarily directed towards this tool. Through modifications to its source code, we created new functions and modified existing ones to support the same functions it already did but with automatic handling of the two database systems and their respective data. In the next chapter, we will go into detail about these modifications.

3.2.3 PyMongo

This is a python distribution for interacting with MongoDB databases from Python. It sits beneath the Mongoengine DOM and is the one that actually communicates with the MongoDB instances. Similar to Mongoengine, it offers a set of methods and functions to interact with MongoDB from python, although it is on a much lower level when compared with Mongoengine. We didn’t modified this tool and instead chose to modify Mongoengine, albeit we could have. In reality, modifying one tool or the other would yield basically the same results (one would be modified while the other would stay relatively unchanged,
Figure 3.1: System view with all the different subsystems that constitute it

simply forwarding the requests and replies). We opted for Mongoengine since it is relatively higher level, and thus the modifications appeared simpler. This choice had its own limitations which we further elaborate in section 4.5.1.

Figure 3.1 shows the overall architecture of our system.

3.3 Data migration

In order to cope with the problem of storing data in the most efficient way possible, depending on either it being “live” or not, we defined the following strategy using the systems previously mentioned.

In Mongoengine, the programmer typically defines the schemas for the different data that he wants the system to handle. In these schemas, the programmer includes different fields that represent the different characteristics of the data. Our solution automatically includes a special hidden field in these schemas. We named this field “liveness” and it represents the state of the data (and since the state of the data represents the database in which the data is to be stored in, this field basically represents the database in which the data must be stored).

If this field is set to true (since we were working with two databases, we chose to use a boolean value
for this field), it means that the data is to be stored in the live system. If, otherwise, the liveness field has value false, the data is to be stored in the archival system.

It is this liveness field, and its respective value, that changes through the course of the data’s life, that models the different states of the data. This evolution in the data’s state translates into migrations between the live and the archival systems.

Our approach to the migration of data between the two databases is rather simple. When data is first created, the liveness field is automatically set to true and thus the data is stored in the live system. When a predetermined time period has passed (we further elaborate all of this in the next section), that data is no longer considered as being "live" and so it is moved to the archival system, i.e., it is stored in the archival database and deleted from the live database. Our approach is simple because we don’t consider accesses to data. This makes it so that after creation, even if accessed afterwards, the data migrates to the archival system after the predetermined time period has passed. We also don’t consider migrations from the archival system to the live system (although the same mechanisms used to migrate from the live to the archival would still apply to a possible migration from the archival to the live system).

When data is deleted, independently of if being "live" or not, it is removed from the respective database.
System design and implementation

Contents

4.1 Read protocol ................................................................. 33
4.2 Write protocol ............................................................... 36
4.3 Delete protocol ............................................................. 37
4.4 Data migration protocol ................................................... 37
4.5 Supported operations ....................................................... 40
In this chapter, we will present the design and implementation of our system.

As previously mentioned, the bulk of our efforts was concentrated in the Mongoengine DOM tool. Our objective was to modify Mongoengine such that it would work with both databases, the live system and the archival system, transparently and automatically, from a user's perspective. The original Mongoengine tool already supported multiple MongoDB databases but transitioning from one to the other had to be done manually. The modifications that we made allow for this transition to be done automatically, without the user knowing that they were made, and without the user having to manually do them.

We start off the section by explaining in detail the read, write and delete protocols, since these three operations are at the core of data storage. It is worth noting that there are various functions in Mongoengine that allow for the creation, for the querying and for the removal of data but the underlying logic is the same across them all, with differences existing at the implementation level. Mainly, Mongoengine allows for data manipulation through two object classes and our work consisted of modifying both of them.

The first one is the Document class which is the class used for defining the structure and properties of collections of documents stored in MongoDB (i.e. its instances represent documents stored in MongoDB). This class exports many methods for manipulating data, such as, save, update, reload, and delete. The second class is the BaseQuerySet class. This class wraps a MongoDB cursor (a cursor is a pointer to the result set of a query that can be iterated to yield the results), providing Document objects as the results. This class exports, as well, various operations for manipulating data, such as create, delete, update, count, get, and insert. The most complex one to modify was the BaseQuerySet class since within the Document class itself we have the document we're interested in available to us (because Document instances represent data stored in MongoDB).

We also lay out, in a much more technical and detailed way, the data migration protocol that we touched upon in the previous section.

Throughout the chapter, we explain the data structures maintained by our system, which are needed to its correct operation and to the correct functioning of all the protocols we mentioned.

We end the section by enumerating the various data manipulating operations that our system supports and by pointing out some limitations of our own design and implementation.

4.1 Read protocol

Mongoengine allows for the querying of data in a multitude of ways but all of them are through the BaseQuerySet class (even though the Document class exports a reload method, which allows for attribute reloading from the database, this is just a wrapper for BaseQuerySet). From queries based on the primary key of a document to queries by a field attribute, the possibilities are almost endless.
Traditionally, the read protocol in Mongoengine is pretty straightforward. By using the BaseQuerySet object, a user can query the database against a specific collection of data. However, Mongoengine doesn’t support multiple databases automatically.

In order for us to support multiple databases as if there was only one, some modifications had to be made to the original Mongoengine. The strategy is rather simple. Every time there is a read from the database, we have to instead read from all the databases and produce a result that would be the logical aggregation of the results of the different databases.

At the implementation level what this means is that instead of working with one MongoDB cursor corresponding to querying on specific database, we worked with two, and thus we had to adapt Mongoengine’s code and structure to work with an array of two cursors.

We should mention that producing a result that is the result of combining the different single results from the multiple databases and that is logically equivalent to the one we would get from a single database is rather difficult and, in some operations, it is even impossible, at least at the Mongoengine level - we further elaborate on this topic in section 4.5.1 and explain why this happens.

In the read case it is somewhat easier since we basically just return an aggregation of single database results. For example, if we're searching for all the documents that have a certain value for a certain field which isn't the primary key field, we have to query both databases since there might be some documents that match the query in the live system and some others in the archival database and we want to return them all. Once we get the results from this search across all the databases, we return them to the application that issued the request.

However, this is much more complex if, instead, of a simple get operation we apply other operations on top of it. For example, imagine a simple query by some attribute upon which we further apply a limit operation, a skip operation, and finally an order by operation. As we can easily imagine, even though the basic get operation is simple, the extra conditions imposed by the other methods, make this a very complex query. Our implementation strategy was based on the original Mongoengine implementation in which these operations and constraints are applied on top of the MongoDB cursor (it is the cursor that applies them, Mongoengine only proxies the requests). Since in the original version there was only one cursor at a time, we had to adapt all of the BaseQuerySet methods that were ready for a single cursor, to work with two instead (and this, in some cases was impossible to do since the cursor object encapsulates some information that we can’t modify at the Mongoengine level). This meant that we had to implement functionalities, that were originally implemented in the cursor, at the Mongoengine level (for example, the count method originally invoked the count on the cursor and returned the result. We had to, at the Mongoengine level, call the count operation in both cursors and sum the results - originally there was no sum performed at Mongoengine for this operation).

Even though the implementation itself is complex, the strategy of querying both databases separately
is simple (we present next an optimization that should considerably improve the read performance in
some specific cases), and it is the only feasible one since we don't know which data is stored in which
database, and so we are forced to query both the live and the archival systems, and because the fact
that Mongoengine supports complex queries (by complex queries we mean queries that aren't based on
the primary key of the document we're querying for - we see next why this is a simpler case).

4.1.1 PK queries optimization

We implemented an optimization for reads that we think will be useful (the results can be seen and
compared in the evaluation section). We identified a possible opportunity to save time when querying
the databases, since we normally have to query the two databases to return the correct results. This
opportunity arises when the query contains the primary key of the document that we're searching for.
If we could, somehow, know, before querying the databases, in which MongoDB database that specific
document was stored, we could, theoretically, save query time since we would know from the get-go
which database to query. In order to know this kind of information, some type of structure, mapping the
documents' identifiers to their respective database, would be needed.

In order to implement this optimization strategy, we started off with a simple dictionary, using the
identifiers as the keys and the database's identifier as the value. However, this strategy had one prob-
lem. It was too expensive, memory-wise. Since the documents stored in our systems can go up to
the thousands, having a dictionary containing all these identifiers would be impractical. We needed a
structure capable of storing a high volume of data while keeping memory usage to a minimum. This
structure should have, as well, a fast and inexpensive operation for checking, given a document's id, the
database we want to query.

Given these constraints, we chose to use bloom filters [24] as the data structure for this optimization,
since they are a space-efficient data structure and they can be used to check, in constant time, whether
an element is a member of a set or not. We had to implement the mapping document's id to database's
id using the simpler bloom filter structure, which is basically an array of bits. To do so, we implemented
a directory by using a bloom filter that would contain the identifiers of the documents stored in the live
system, that is, documents with the attribute field "liveness" set to true. So, upon querying by primary key,
we look into this directory of ours, which, basically, translates to checking if the identifier is in the bloom
filter. If the check returns false, we can be sure that the document, to which the identifier pertains, isn't
stored in the live database and so we query the archival database. Note that the document could have
been deleted and, consequently, not be present in any of the databases. We could have used a second
bloom filter for the documents stored in the archival system but the memory requisites would have been
doubled. We considered that querying the archival database even when the document doesn’t exist was
a good compromise between time efficiency and space efficiency (queries by primary key of deleted
documents shouldn’t be frequent either).

If the check returns true, we have to be a little more careful since false positives are possible with bloom filters. In this case, we query the live database as if false positives weren’t possible. If this query doesn’t return any results, we have to query the second database since we can be in the presence of a false positive. This strategy should work fine and reveal itself to be efficient, since false positive rates can be significantly low.

We still have to think about what happens when we query, with the primary key, some document that was originally in the live system but then migrated to the archival system. In this case, we have to delete the corresponding identifier from the bloom filter. But there is one problem. Traditional bloom filters don’t support deletions. The answer to our problem is a counting bloom filter [24]. By using a counting bloom filter instead of a regular one, we can remove the identifiers of the corresponding migrated documents and thus checking if a migrated document id is in the bloom filter should return false. In case it returns true (false positive) we simply proceed as previously explained.

4.2 Write protocol

The write protocol is pretty straightforward. All documents, independently of the method used to create them (Mongoengine has various methods that support, in one way or another, the creation of documents as we already saw), are saved in the live system when created. This happens because, logically, a document that was just created is recent.

Data creation is possible through the Document class or through the BaseQuerySet class and the implementation varies, naturally, from one class to the other. The latter allows for data creation through the create method, the insert method (for bulk inserting of documents - all the other methods allow for only one document to be saved at a time), the update method with the upsert flag true, the upsert one method, and the modify method with the upsert flag true. The former allows data creation through the save method and the update method with the upsert flag true.

Even though the write protocol is pretty straightforward, there are some subtleties we have to account for to make sure everything works just as expected. The first one is making sure we don’t forget to save the identifier of the newly created document in the directory (the bloom filter representing the “live” documents), thus, ensuring our optimization based on the primary key works correctly. The second subtlety is to make sure we save a timestamp of the creation of the document (we further elaborate and explain this on the subsection pertaining data migration).
4.3 Delete protocol

In Mongoengine we can delete data mainly by two ways. If we have the document instance we want to delete, we can call the delete method directly over that instance. If we don’t have the document instance (or instances) that we want to delete we need to query the databases, through the query object, and we call the delete method over the returned results. Internally, these two ways of deleting data work pretty similarly to one another. They both search for the document in the database, the difference being that the first way always searches for the document by primary key while the second can query for any other attribute or combination of attributes (thus allowing for deleting documents in bulk and not just one document at a time).

Depending on which way we’re deleting data, our system handles the removal of data from the databases in two different ways. If we’re calling the delete method directly with the document instance we want to remove, we can use our primary key optimization strategy to delete the intended document from the correct database right away. If we’re not calling the delete method with the document instance, we will need to query all the databases and delete all the documents that match the query from both the live and archival systems (of course, if the query is by primary key we can deploy, once again, our primary key optimization strategy).

As happened with the write protocol, we have to consider some subtleties referring to the directory of identifiers. When we delete documents we have to delete the corresponding identifiers from the bloom filter that holds the ids of the “live” documents. This is pretty simple since, as we already mentioned, we used a counting bloom filter that supports delete operations. If the delete operation was issued on a query by a non primary key attribute, we simply retrieve the documents corresponding to that query prior to the delete operation, thus getting their primary key values (we only remove from the directory if the delete operation is successful) and then we remove those keys from the bloom filter.

4.4 Data migration protocol

An essential part of our solution is data migration, done automatically, between the two database systems. We must be capable of transferring documents between the live system and the archival system since our idea revolves around the idea of archiving “old” data in the archival system.

In order to implement this data migration protocol we started out by the very basic. We needed a save operation that could, depending on the value of some field (which we named “liveness”) of the document’s model, save a document in the live system or in the archival system. We created this method using the save operation that Mongoengine already exported through the Document class. This new save would need to, prior to saving the document to the database, check the value of the liveness field of that document. If the value of this attribute was true, the document was stored in the live system.
In contrast, if the liveness field value was false, the document was stored in the archival system. With this new save operation we could, by manipulating the value of the documents’ liveness field, choose in what database to store the data.

The next step was about registering the time of creation of the documents. This was important because we needed to know, somehow, which documents had been created a certain time period ago and eventually change their databases. We decided that we would, upon creation, save the timestamps of the documents. We decided not to use traditional timestamps (a mapping between identifier and timestamp) since we would have to keep a table that, in the worst case scenario, would contain all the identifiers of all the data in the system. This wasn’t practical nor scalable. Instead, we chose to use a mechanism based on generations that would allow for a more efficient usage of memory and a more scalable solution.

The idea of the generations was to have a fixed, maximum number of generations in which the older ones would be deleted from to make space for the new ones. Following this logic, if we had a list with a maximum allowed size, in which each position represented one of these generations, we could start out with the empty list (no generations) and progressively fill it out. When we eventually tried to create a new generation but the list was already at maximum capacity, we would then pop the highest indexed generation (i.e. the oldest generation) from the list, shifted it and then insert a new generation at the beginning. The data structure that we needed for this operation was a queue, since we would be following a FIFO policy, removing the first element in and inserting in the beginning (an operation that isn’t efficient in traditional lists). Thus, we implemented this idea with a deque, a python’s collection with fast appends and pops on either end. Figure 4.1 shows the underlying principle of the generations mechanism.

We initialize the queue with a maximum capacity of elements, meaning that we will support at most that number of simultaneous generations. A new generation is spawned at regular time intervals, in the fashion discussed above. The value of the time intervals times the maximum capacity of the queue yields the time it takes for one generation to go from newly created to deleted and, consequently, for the documents whose identifiers were in that generation to be migrated from the live system to the archival system. By manipulating the number of generations and/or the time interval to spawn a new generation, one can change the time it takes for one document to be moved from one database to the other, either making it longer or shorter. Bigger time intervals between spawning new generations, given a same length queue, means that each generation will hold more information regarding document identifiers and that the generations’ queue will take longer to fill up and consequently data will take longer to migrate. Bigger queues, given the same time period for generation spawning, mean more generations, which translates into a bigger time for a document to go from the live database to the archival, and more memory usage. Generally speaking, if one wants to hold more information regarding generations, he/she
can either increase the size of the queue or increase the time period between generation spawning (or even a combination of both). The latter might lead to coarse-grained generations while the former requires more generation spawning cycles.

Having created the generations mechanism, we now simply needed to, upon a document creation, insert its identifier in the newest generation possible (i.e. insert it in the beginning of the queue). Once a generation gets deleted, we pop all the identifiers that it contained. With these identifiers, we fetch, from the respective databases, the documents. Once we have the documents, we first delete them from their old databases and then we invoke our created save operation (note that we change the values of the liveness field of the documents prior to the save operation call so that the save operation saves them in the other database). After this step, we have successfully transferred all the documents that were in the older generation from their original database to the other database (in our case this will always be a transition from the live system to the archival system but the mechanisms apply either way).

As we already mentioned in section 4.1.1, when we migrate a document from one database to the other, we can’t forget to update the directory of the document ids that are stored in the live system.

Regarding documents that are deleted before they’re migrated, instead of removing the respective identifiers from the respective generations (which would be difficult since we wouldn’t know in which generation that identifier was stored), we simply ignore them. When a generation is popped and we search for the documents with the respective identifiers, if any of these identifiers correspond to a document that was deleted, that search will simply return nothing and we discard that result.
4.5 Supported operations

Mongoengine supports many different operations (we already mentioned some of them previously). These operations correspond to methods of the BaseQuerySet class and of the Document class. Some of them are simple, others not so much, and they can generally be chained together to yield more complex queries. For example, we can limit the number of documents returned by a query through the limit method.

Our objective was to integrate all of these operations in our modified version of Mongoengine and have them working as they logically would with only one database. However, and even though we tried to the best of our capabilities to support the majority of these operations in our two database system, there were some that we couldn’t adapt correctly.

The main problem with why we couldn’t adapt some operations was because Mongoengine is a relatively high-level tool (we could possibly have done it if we had modified PyMongo also). Since Mongoengine sits on top of PyMongo it uses, naturally, the methods and structures provided by PyMongo for data manipulation when working with MongoDB (obvious that Mongoengine adds functionalities to PyMongo). When talking about data manipulation, Mongoengine relies heavily on the cursor object and the collection object exported by PyMongo. The cursor objects is a generator that yields the queries results and the collection object represents a MongoDB collection (i.e. a grouping of MongoDB documents, the equivalent of an RDBMS table) on which we can perform operations. These two objects implement various data manipulation functionalities on which Mongoengine relies. For example, when we use the limit operation to limit the number of results returned by a query, what really happens is that we call limit on the cursor object of that query and thus it isn’t Mongoengine that truly limits the results but rather PyMongo. To implement this operation on our system, we had to call the limit operation on both cursors and add extra logic, that wouldn’t normally exist, in Mongoengine to deal with the problem of having to work with two databases as if they were one. If we naively called limit on both cursors and did nothing more, we could have had the following (incorrect) scenario. A request to limit the results to five documents. The first database has ten total documents that matched the query and thus returns the first five and the second database does the same thing. It is obvious that this is wrong. Limit should limit both databases as if there was only one and thus, in this example, only the first five documents from the first database should have been returned.

We had to add this extra logic in Mongoengine in order to cope with the two databases, while keeping PyMongo intact.

We list next the operations from the original Mongoengine that our system supports (the behavior is equivalent to what it would be if we were considering one single database).

- Count
4.5.1 Partially supported operations

There are some operations supported by Mongoengine that our modified version doesn’t support, i.e., they don’t work with two databases as if there was only one. For example, the order by function should consider all the documents matching the query, regardless of the database in which they were stored, and then order them, but what actually happens in our implementation is that the order by function orders the results from each database independently ordered. For example, consider a model with an
age field. We have some documents in the live system with values 10, 12 and 13 for this field. In the archival system we have two documents which have values 11 and 14 for this age field. If we wanted to order the documents by the age field, from lowest to highest, we would expect the result to be 10, 11, 12, 13 and 14, and this is what would happen if the documents were all stored in one single database. However, in our system what would be returned would be 10, 12, 13 (corresponding to one database), 11 and 14 (corresponding to the other database) since the order by function is applied separately to each database.

This happens because some of these operations execute in the server side (exec js, for example) or simply because Mongoengine operates at a too high of a level. The first case is self explanatory. Since the operation is executed on the server side, in each database independently, we can’t combine the results as if there was only one database (we could but we would have to implement the functionality in the client side). All we can do is present the results by database. The second case happens because Mongoengine sits on top of PyMongo and relies heavily on it. Sometimes it isn’t possible to manipulate PyMongo’s output, which is specific to each database.

We list next the set of operations that are partially supported in our system:

- Order by
- Exec js
- Map reduce
- Aggregate

### 4.5.2 Not supported operations

We list next the operations from the original Mongoengine that we didn’t support in our modified version.

- Search text
- Hint
- Batch size
- Ensure index
5 Evaluation

Contents

5.1 Latency ....................................................... 45
5.2 Throughput .................................................. 48
5.3 Discussion ................................................... 48
We designed and run tests to measure the latency and throughput of our modified version of Mongoengine and compared the results with the original, unmodified Mongoengine.

All tests were run in a Virtual Machine running on a Windows 10 64-bit PC with an Intel Core i7-5500U CPU @ 2.40GHz, 8.00 GB of ram Ubuntu 16.04 (64-bit), and with two graphics cards, Intel(R) HD Graphics 5500 and NVIDIA GeForce 920M. The Virtual machine was running Ubuntu(64-bit) 16.04 with two processors, 4096MB of base memory, and 12MB of video memory.

First, we present the latency tests and then the throughput ones. Both tests were run for the get, put, and delete operations, for both the original and the modified versions of Mongoengine.

We discuss the obtained results at the end of the chapter.

5.1 Latency

For these tests, we ran each operation 1000 times and then we took the average of those results for each test (in milliseconds). For our modified version of Mongoengine, we ran each operation for the live system and for the archival system, i.e., we issued a get for a document stored in the live database, a get for a document stored in the archival database, a delete for a document stored in the live database, and a delete for a document stored in the archival database (for the put operation it wouldn’t make sense to do this distinction). We also tested both the get and delete operations with and without our primary key optimization strategy.

Figures 5.1, 5.2, 5.3, 5.4, and 5.5 show the results of these tests.
Figure 5.2: Get with PK operation latency test results

Figure 5.3: Get without PK operation latency test results
Figure 5.4: Delete with PK operation latency test results

Figure 5.5: Delete without PK operation latency test results
5.2 Throughput

For these tests we followed the next sequence of steps. We launched N different clients, each one doing consecutive requests (get, create, or delete). Each client ran during ten seconds. We recorded the total number of operations (get, create, or delete) executed by all the N clients. We divided this total number by ten and obtained the number of operations done per second. We then repeated these steps while varying the number of clients, for both our version and the original one.

The obtained results are shown in figures 5.6, 5.7 and 5.8.

5.3 Discussion

In terms of latency, the results are what we expected. Our modified version is substantially slower than the original one (it is roughly ten times slower across all the operations tested). This was expected to some degree since performance wasn’t really a concern of ours and our solution is substantially more complex than the original version.

Analyzing only the results of our modified Mongoengine, there are some things worth noting. Our optimization involving the primary keys revealed itself successful. Comparing the results of a Get operation with and without the primary key field, we can clearly see in the results that our optimization idea roughly halved the response time (it is still significantly slower than the original Mongoengine version).

It is also worth mentioning the fact that the response times for all the operations measured were practically the same for the main database (the live system) and the secondary database (the archival system). This happens most likely because when we don’t have a query based on the primary key, we
Figure 5.7: Get operation throughput test results

Figure 5.8: Delete operation throughput test results
need to query both the databases and so we always make two accesses in these types of situations.

In terms of throughput, the results are also in line with our expectations. Our modified version of Mongoengine has a lower throughput across all the operations tested, being the biggest discrepancy found in the delete operation and the smallest in the get operation. As we mentioned, this lower throughput was expected since our modifications introduced additional complexity in the Mongoengine tool.

It's worth mentioning that we run these tests with very limited hardware which downgraded the overall latency and throughput of the tool. However, since all tests, for both the modified and the original versions of Mongoengine, were run in the same conditions, hardware wise, the relations between the data obtained are still valid and we can draw the same conclusions that we would if we had more capable hardware.
6 Conclusion

Contents

6.1 Future work ............................................................... 53
In this document, we presented our solution to the problem of automatically and transparently handling the migration of data between two databases. Our work started out more directed towards the specific case of Unbabel's data problem but then branched out into a more generic solution, considering MongoDB instances and the Mongoengine DOM tool.

It was quite challenging implementing the many operations supported by Mongoengine in our system since this is an highly complex tool. Nonetheless, our modifications were successful in a variety of operations. Some modifications weren’t successful and this happened mostly because we didn’t modified PyMongo. If we had modified both Mongoengine and PyMongo, we could have implemented all the functionalities originally supported by Mongoengine. However, this would have been extremely complex since PyMongo is, by itself, as complex (or more) as Mongoengine. Our solution was a good compromise between complexity and operations supported by our version of Mongoengine, since it wasn’t as complex as a solution involving both systems but it didn’t implement all of the functionalities of the original Mongoengine.

### 6.1 Future work

The most obvious feature to implement in the future would be to modify PyMongo and, in doing so, implementing the missing operations of our solution. We could also strive for a better performance, since our solution had a considerable overhead in terms of performance.

Future work might also include concurrency and consistency, since our solution didn’t touch on these aspects at all.

One interesting aspect would be to consider more than two databases, and generalize this system for N databases.
Bibliography


