Automatic and transparent selection of cloud storage services

ANTÓNIO FERREIRA, Instituto Superior Técnico, Universidade de Lisboa, Portugal

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.

Additional Key Words and Phrases: cloud storage, storage services, data migration, automatic

1 INTRODUCTION

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 for almost infinite scalability without the need for organizations to host and manage large and expensive infrastructures which in turn would entail a big management cost. They also have the particularity of being ready to use as soon as the client subscribes to the service.

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 variables. However, the most suitable solution yesterday might not be the best today, and the clients must adapt as the situation demands in order to fully take advantage of this type of services, in order to minimize costs.

In a real life scenario, what usually happens 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, prompting a transition 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.

2 PROBLEM

We address this problem of adapting to the changing environment 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 current model.

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.

3 OBJECTIVES

The main objective of our work is to address this problem, motivated by the real-world experience of Unbabel, in an automatic (i.e. without the user having to do extra work) and transparent fashion (i.e. with zero or little changes to the existing client code).

This will be achieved by implementing a new data model. The idea 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. This will also be implemented in the client’s side.

4 RELATED WORK

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 real-world use case of Unbabel.

CloudMPcast [Li et al. 2010] and the OCRP algorithm [Chaisiri et al. 2012], 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 [Wilkes 2001], MINERVA [Alvarez et al. 2001], Hippodrome [Anderson et al. 2002], and scc [Madhyastha et al. 2012] 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 [Wu et al. 2013], CostLO [Wu et al. 2015], Conductor [Wieder et al. 2012], and CDStore [Li et al. 2016]. The combination of cloud providers seemed like a promising feature at first but to simplify the overall complexity of the solution, we ended up consider only one provider. 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, 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 [Li et al. 2010] is an interesting tool that allows customers to compare different cloud providers. It could be used 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.

5 ARCHITECTURE

Our system is composed by a client, running locally on the client machine, and by two MongoDB databases. We describe next in more detail each of these components and their respective constituents (we use, throughout the rest of the paper, 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).

Figure 1 shows the overall architecture of our system.

5.1 MongoDB

This is the database of choice for our system. MongoDB is a document-oriented database. Our system runs with two 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 designated this mongoDB instance by “live system”, alluding to the very nature of the data that it stores. 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”. 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.

We 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.

5.2 Client

The client side of our system is comprised of three parts. The Mongeengine, the PyMongo and the client application. All of these components run locally, on the client machine.

5.2.1 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 contact酿造 depends on the data that it is being handled. This communication is transparent to the application itself and is handled by Mongeengine and PyMongo. All that is visible to the application is the interface exported by Mongeengine, 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.

5.2.2 Mongeengine. 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.

5.2.3 PyMongo. This is a python distribution for interacting with MongoDB databases from Python and it sits beneath the Mongeengine DOM. Similar to Mongeengine, 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 Mongeengine. We didn’t modify this tool and instead chose to modify Mongeengine, 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, simply forwarding the requests and replies). We opted for Mongeengine since it is relatively higher level, and thus the modifications appeared simpler.

5.3 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 Mongeengine, the programmer typically defines the schemas for the different data that he wants the system to handle. In these
As previously mentioned, the bulk of our efforts was concentrated (although the same mechanisms used to migrate from the live to the made allow for this transition to be done automatically. The modifications that we supported multiple MongoDB databases but transitioning from one to the other had to be done manually. The original Mongoengine tool already in the Mongoengine DOM tool. Our objective was to modify Mon- goengine 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.

6 DESIGN AND IMPLEMENTATION

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.

We start by explaining the read, write and delete protocols. 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 class is the Document class which is 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.

6.1 Read protocol

Mongoengine allows for the querying of data in a multitude of ways but all of them are through the BaseQuerySet class. 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.

In the read case, we 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. 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. 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, and it is the only feasible one since we don’t know which data is stored in which database.

6.1.1 PK queries optimization. We implemented an optimization for reads that we think will be useful. We identified a possible opportunity to save time when querying the databases, since we normally have to query both 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 know, before querying the databases, in which MongoDB database that specific document was stored, we could save query time. 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 problem. It was too expensive, memory-wise. 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 [Tarkoma et al. 2012] 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 check 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.

If the check returns true, we have to be careful with bloom filters false positives. 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.

When a document, that was originally in the live system, migrates to the archival system, 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 [Tarkoma et al. 2012]. 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.

6.2 Write protocol

The write protocol is pretty straightforward. All documents, independently of the method used to create them, are saved in the live system when created. This happens because, logically, a document that was just created is recent.

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).

6.3 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.

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 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).

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.
6.4 Data migration protocol

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 append and pops on either end. Figure 2 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, 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), not forgetting to update the directory of the document ids that are stored in the live system. After this step, we have successfully transferred all the documents that were in the older generation from their original database to the other database.

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.

6.5 Supported operations

Mongoengine supports many different operations. 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 thought 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. Since Mongoengine sits on top of PyMongo it uses, naturally, the methods and structures provided by PyMongo for data manipulation when working with MongoDB. 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. These two objects implement various data manipulation functionalities on which Mongoengine relies.

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
- Limit
- Skip
- Distinct
- Only
- Exclude
- Sum
- Average
- Item frequencies
- Rewind
- Update
- Update one
- Upsert one
- Modify
- With id
- In bulk
- None
- Reload
- All
- Filter
- First
- Get
- Create
- Insert

6.5.1 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.

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

6.5.2 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

7 EVALUATION

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.

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

7.1 Latency

For these tests, we ran each operation 1000 times and then we took the average of those results for each test (in milliseconds). Figures 3, 4, and 5 show the results of these tests.

7.2 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 6, 7 and 8.

7.3 Discussion

In terms of latency, the results are what we expected. Our modified version is substantially slower than the original one. This was expected to some degree since 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 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 need to query both databases.
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. 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, 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.

8 CONCLUSION

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.

8.1 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.

ACKNOWLEDGMENTS

The author would like to thank Prof. Rodrigo Rodrigues for his invaluable guidance and support, his family and friends for all the fun and joyous moments and, last but not least, to “Estrela” for being the best cat anyone could have asked for during the last thirteen years.

REFERENCES


