Università di Firenze, Università di Perugia, INdAM consorziate nel CIAFM

DOTTORATO DI RICERCA
IN MATEMATICA, INFORMATICA, STATISTICA

CURRICULUM IN INFORMATICA
CICLO XXXIII

Sede amministrativa Università degli Studi di Firenze
Coordinatore Prof. Paolo Salani

Enhancing cache content management in a data lake architecture using Reinforcement Learning

Settore Scientifico Disciplinare INF/01

Tutore
Prof. Marco Baioletti
Prof.ssa Valentina Poggioni

Dottorando:
Mirco Tracolli

Referente INFN
Daniele Spiga

Coordinatore
Prof. Paolo Salani

Anni 2017/2020

Abstract

In the past few years, data dimensionality has become so high and complex that a specific field has been created: Big Data. Besides the size of the data, that is continuing to grow in each sector, from business to scientific domains, the advent of IoT (Internet of Things) and data from sensors, introduces a large volume of information that is not simple to manage and to extract valuable knowledge.

The process to extract useful information and value from such data is mainly composed of two phases: first, the processing, and then the data access.

One of the main requirements for data access is fast response time, whose order of magnitude can vary a lot depending on the specific type of processing as well as processing patterns. Therefore, besides the specific optimization of algorithms and software processes, there are several aspects that involve the infrastructure level of the analysis environment that could be enhanced. From this point of view, the optimization of the access layer becomes more and more important while dealing with a geographically distributed environment where data must be retrieved from remote servers of a Data Lake.

From the infrastructural perspectives, caching systems are used to mitigate latency and to serve better popular data. Thus, the role of the cache becomes key to effective and efficient data access.

In this thesis, we will explore how to make a cache autonomous and adaptable to improve the performances of a system in terms of data management with the aim of reducing the cache costs, such as the amount of data written and the amount of data read from the cache memory.

Dedication

I dedicate this work to my family, that sustained me during these years.

Declaration

I herewith formally declare that I, Mirco Tracolli, have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. This thesis has not been handed in or published before in the same or similar form.

Acknowledgements

I want to thank all the people that supported me during the difficult moments and that let me stand up against the current.

Contents

Page

List of Figures iii

List of Tables vii

0 Introduction 1

1 Background on Data Lakes 5

1.1 Cloud service models 6

1.2 Big Data and the Cloud 8

1.3 Data caching in cloud systems 11

1.4 Data lakes 12

2 Background on Machine Learning 15

2.1 Machine learning 16

2.2 Reinforcement Learning 20

2.2.1 Concepts and terminology 21

2.2.2 Approach details 24

2.2.3 Algorithms 27

3 Related works 31

4 Problem description 35

4.1 Definition and targets 36

4.2 Context 39

4.2.1 Data-sets 44

4.3 Simulation environment 50

5 Smart Caching for Data Lakes **59**
5.1 Exploring file scoring 60
5.2 An agent for a better addition 62
5.3 Smart caching with double agents 66

6 Experiments **73**
6.1 Evaluation metrics 75
6.2 Tests and results 81
6.2.1 Selection by file score 83
6.2.2 Cache composition by SCDL 87
6.2.3 Cache management by SCDL2 90
6.2.4 Extreme use cases 93
6.3 Evaluation 95
6.3.1 Metrics correlation 95
6.3.2 Notes on agent learning 95
6.3.3 Notes on selected features 97
6.3.4 Result considerations 98

7 Future steps **103**
7.1 Q-Learning improvements 104
7.2 Deep Q-Learning 105
7.3 Integration 106

8 Conclusions **113**

Bibliography **117**

A Appendix **125**
A.1 Cloud services 125
A.2 Data Analysis pipeline 127
A.3 Synthetic datasets 128
A.4 Result view through the simulation analyzer 130

List of Figures

1.1 Cloud service model management details, with the responsibilities of the various roles 7
1.2 New cloud service model schema in comparison to IaaS model . . 8
1.3 Data lake model schema with a focus on the internal organization of the lake 13

2.1 Machine Learning engineering cycle, from problem issue to the deployment of a solution 20
2.2 Generic reinforcement learning interaction schema between the agent and an environment 22

4.1 Action schema that depicts when the cache retrieves a file on miss 37
4.2 Action schema that shows when the cache is used as a proxy to serve a missed file 37
4.3 Basic environment schema with focus on throughput aspect, a metric to evaluate the caching behavior 38
4.4 Projections of the increase of data storage at CMS 40
4.5 WLCG Data Lake schema example with focus on internal components 42
4.6 Activity composition in a task of a data analysis user 44
4.7 Data general statistics of Italian requests in 2018 showing the number of files, requests, jobs, tasks, users, and sites during the year 45
4.8 Data general statistics of United States requests in 2018 showing the number of files, requests, jobs, tasks, users, and sites during the year 46

4.9 File size month distribution of Italian requests in 2018 showing the most common sizes for the file requested in each month of the year 47
4.10 File size month distribution of United States requests in 2018 showing the most common sizes for the file requested in each month of the year 48
4.11 Data type composition of Italian requests in 2018 showing insight of domain-specific information about the file type 48
4.12 Data type composition of United States requests in 2018 showing insight of domain-specific information about the file type 49
4.13 Illustration of environment implementation with the focus on the components selected to solve the problem 50
4.14 Simulation environment schema showing the several aspects taken into account and the units measured 52
4.15 Golang cache interface used in the simulator 55
4.16 Simulation environment configuration file example in YAML . . . 57
4.17 Result dashboard example screenshot 58

5.1 Schema of weight function idea to select the file to store 60
5.2 Schema of the environment of single-agent approach that uses the information of client request to accept or not a file into the cache 63
5.3 Reinforcement Learning schema of the double agent approach, where the AI choose both, the addition and eviction of a file into the cache memory 67

6.1 Projection of the metrics on the simulation environment that shows the several aspects controlled to study the cache behavior 75
6.2 CPU efficiency difference between a local and remote served file 80
6.3 Explanation of the experiments' targets compared to the cache model used 81
6.4 Correlation matrix of the various metric used in the experiments 96
6.5 Result investigation of *Read on hit ratio* for both datasets with a cache that has infinite memory 100

6.6 This plot shows the difference between SCDL2 with no eviction and LRU, using a cache of size 500T. The red line is the average gain of SCDL2 102
6.7 This plot shows the difference between SCDL2 with no eviction and a cache size of 500T versus a cache of size 1000T that uses LRU. The red line is the average gain of SCDL2 102

7.1 Cumulative number of miss on delete for period B of the year (Apr-Jul) using the Italian dataset 105
7.2 Frequency distribution of miss on delete an miss on skip for SCDL in period B (Apr-Jul 2018) for the Italian dataset 106
7.3 Frequency distribution of miss on delete an miss on skip for SCDL2 with no eviction in period B (Apr-Jul) for the Italian dataset 106
7.4 Schema of a machine learning typical pipeline 109
7.5 DODAS architecture schema showing how the stack is implemented 110
7.6 XCache plugin schema for AI integration which shows where it is possible to intervene and extend the framework 111

A.1 Cloud service models and abstraction level 125
A.2 Typical data analysis pipeline 127
A.3 The dataset generator dashboard 128
A.4 Synthetic dataset configuration example in JSON 129
A.5 Throughput comparison using the simulation analyzer 130
A.6 Cost comparison using the simulation analyzer 131
A.7 ϵ decay comparison between SCDL and SCDL2 131

List of Tables

4.1 Simulation parameters 56

6.1 Test results grouped by function family. 85
6.2 Test results of IT region with cache size of $100T$ and $10Gbit$ bandwidth. 87
6.3 Test results of IT region with different cache sizes and $10Gbit$ bandwidth. 88
6.4 Test results of US region with different cache sizes and $10Gbit$ bandwidth. 89
6.5 Test results of IT region with cache sizes of $100T$ and $10Gbit$. . . . 90
6.6 Test results of IT region with different cache sizes and $10Gbit$ bandwidth. 92
6.7 Test results of US region with different cache sizes with $10Gbit$ bandwidth. 93
6.8 Results of extreme big size synthetic dataset with a cache of 100GB and 10Gbit of bandwidht. 94
6.9 Results of extreme small size synthetic dataset with a cache of 100GB and 10Gbit of bandwidht. 94
6.10 Results of the experiments with a different ϵ decay rate. The table shows the average gain in performance for each metric compared to the ϵ value chosen in the thesis experiments 97
6.11 Results of the experiments without the feature size that show the gain in performance for each metric 97
6.12 Comparison of SCDL and SCDL2 in IT region with cache size of $100T$ and $10Gbit$ bandwidth. 98

6.13 Comparison of weight functions in IT region with cache size of $100T$ and $10Gbit$ bandwidth. 99
6.14 Comparison of SCDL and SCDL2 in IT region with different cache sizes and $10Gbit$ bandwidth. 99
6.15 Result comparison of IT region with different cache sizes and $10Gbit$ bandwidth. 101

Introduction

In the last decade, data volume has grown beyond any expectation in many sectors, from science to business.

Nowadays, it is customary to manage and handle data, whose size was regarded as unimaginable in the past years by means of small-size devices like our smartphones.

Plenty of information is generated every day by sensors, peripheral systems, and applications. This gigantic quantity of information is processed via network-based cloud systems, where users do not have to regard the way their data is maintained as a burden.

Cloud systems are usually disseminated all around the globe, for satisfying the demand for access to the services they offer. For this reason, the role played by the network infrastructure is fundamental. The cloud systems a user can access can be of various kinds: public, private, community-maintained, hybrid, or even on-demand.

What is currently being done, is not only storing data onto cloud platforms, but these resources are also being shared throughout a steeply increasing number of web services. For instance, thanks to cloud systems, it is possible to easily access the computational power that would be otherwise unachievable by a single individual.

The data manageability through cloud systems gave birth to the concepts of Big Data and Data Lake. The former concept embodies the possibility of managing data of big dimensions, either in terms of quality, in terms of quantity, or both. The latter concept, roughly speaking, embodies the idea of storing and analyzing an enormous amount of data in a single place. The put into the place of such a concept may occur in different fashions.

In this thesis, we use the concept of Data Lake in the same spirit as WLCG [1, 2] (Worldwide LHC Computing Grid), which defines a Data Lake as a set of data and computing centers that are interconnected by means of the network infrastructure. As in the WLCG idea, this work aims to optimize the resources, improve performance and efficiency, as well as simplifying and reducing operation costs. This thesis proposes an autonomous approach to achieve that in the Data Lake context focusing on cache content management. Despite the study's environment regards High Energy Physics, the aims of the project are to be as agnostic as possible. As a consequence, this work will not use any specific domain information. Moreover, as a method to manage the cache memory, it uses a Reinforcement Learning (RL) technique. The RL approaches are generally used in dynamic contexts and problems, where an agent has to adapt itself as the environment changes without any human intervention. Also, even if they do not guarantee the optimality of the solution found, often they are more efficient than the classical optimization algorithms.

RL solutions are growing in number and are also used in the Big Data context [3, 4]. Nevertheless, this encourages us to experiment with the reactions of an agent in a complex environment such as the cache content management in a Data Lake architecture. Furthermore, the approach is tested in a real Data Analysis context, with information taken from the High Energy Physics (HEP) workflow. Thus, it is also questioning how to measure the different behavior of the cache respects the standard policies and how we can evaluate the cached content for the specific domain.

The idea of this thesis is to develop an independent service, a helper tool for the caching management that supports caching decisions. The main goal is to optimize the content of the cache layer compared to the data stored in the lake and the user requests. As a consequence, a better storage use should increase user task throughput. Moreover, reliable networking can be used to reduce disk replica requirements via caching, despite this will push up the network capacity requirements. In turn, the desired behavior is to write fewer data into the cache layer and store only the files more important and more used. For this reason, it is expected to have a better cache composition to serve at best the user requests.

This thesis is structured as follows: first, the background on data lakes Chapter will cover all the notions about modern Data Lakes and Big Data management. Then, in Chapter 2, it is presented the techniques used in the project approaches, focusing on the theory part to understand the further application. In Chapter 3, several related works are presented and compared with the thesis solution. Chapter 4 presents the addressed problem and also has a follow-up section on the project context. Then, in Chapter 5 there are the exploitation of the thesis approaches used in the experiments. Chapter 6 depicts the metrics used to test several techniques and presents tests and results. Finally, in Chapter 7, it is presented the future evolution of this project before the conclusion.

Background on Data Lakes

Contents

1.1 Cloud service models 6
1.2 Big Data and the Cloud 8
1.3 Data caching in cloud systems 11
1.4 Data lakes 12

This chapter is an introduction to the Data Lakes' world, starting from the infrastructure to the recently adopted models. Furthermore, it is illustrated in the domain field of the project. Also, there will be an insight into data caching, especially in distributed storage with a particular focus Data Lake context.

1.1 Cloud service models

With the evolution of the World Wide Web, the services changed during the years, and with them, also the level of abstraction from the hardware.

Cloud Computing is the possibility of using computer system resources on-demand, like data storage, CPU calculus, GPUs, etc., without direct and active management of such resources by the end-user.

It is explicitly stated in the NIST definition of Cloud Computing [5]:

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Thus, the cloud infrastructure resources are provided as a service, which means they are well-defined functions, self-contained, and does not depend on the context or state of other services. This software designs kind, named Service-Oriented Architecture (SOA [6]), allows access over the network to components through a specific protocol. The NIST's cloud computing has three canonical service models, i) Infrastructure as a Service (IaaS), ii) Platform as a Service (PaaS), iii) and Software as a Service (SaaS),that are illustrated in Figure 1.1 with their different level of abstraction (for more details see appendix Appendix A.1). A lower level of abstraction means that the provider of the service has to manage a small part of the service components and the client is left with a lot of control over the resources.

The different models also represent a different cost of consumption for the users. A higher abstract level, of course, has a higher cost because providers have to maintain the whole stack service. Instead, in a low-level abstraction service, the end-users have to manage more aspects of the infrastructure and pay only the resources asked for. Of course, the payment is on consumption for each kind of service and the costs depend also on several aspects, such as the type of hardware required, the number of instances of the service, the location of the service (geographically speaking), and more.

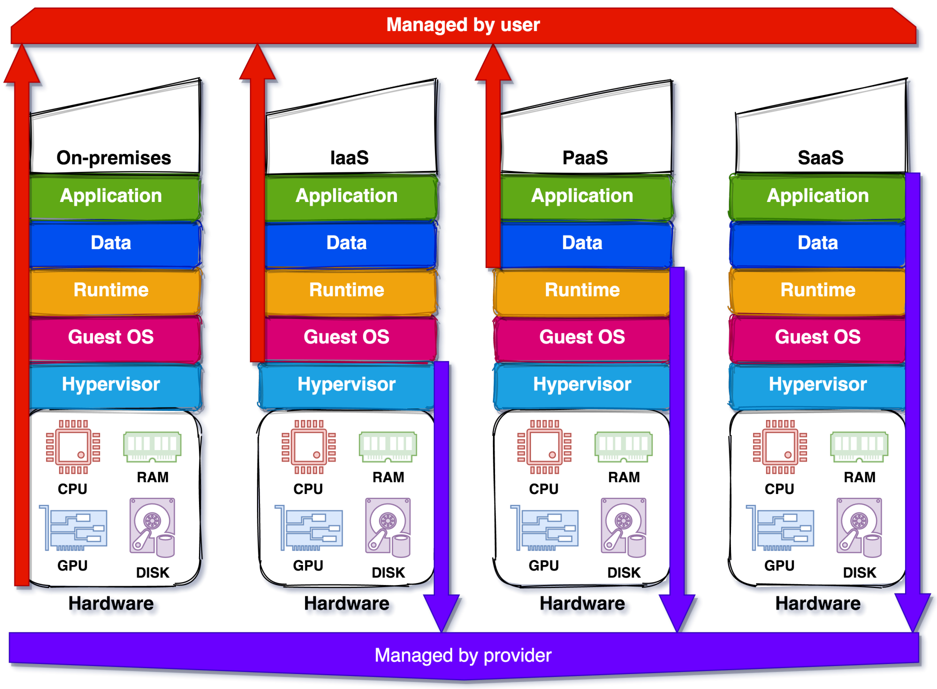


Figure 1.1: Cloud service model management details, with the responsibilities of the various roles

The above three models described are part of the base of "XaaS"[6] and they are only the main ones used. In fact, the "X" stands for everything as a service and that means it can be replaced with whatever letter corresponds with the service provided. For example, there are also Database as a Service (DaaS) or Backup as a Service (BaaS). Recently, two new models appeared in this cloud infrastructure organization as a service, and they became so popular because they are a good compromise between control and responsibility. In particular, they allow the users to create a very specific task with a high customizable setting that is also scalable and reproducible with ease. They are the Container as a Service (CaaS) and the Function as a Service (FaaS) model.

These new types of models introduced the concept of a container, a standardized unit of software that packages up code and all its dependencies. This package has the property to run quickly and reliably from one computing environment to another. Due to these latest models, the Data Lakes architecture becomes feasible and started to be explored as the new paradigm for Big Data

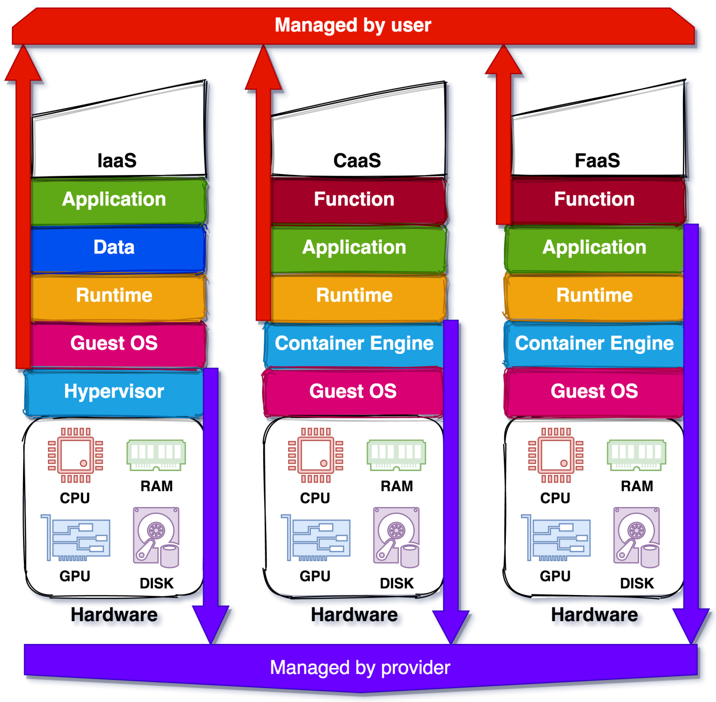


Figure 1.2: New cloud service model schema in comparison to IaaS model

management.

1.2 Big Data and the Cloud

The current cloud services allowed the management of a huge number of tasks and the processing of bigger and bigger volumes of data. The Big Data management could be very different without them today's services, not to mention the frameworks and paradigms specially created to support those kinds of infrastructure described in Section 1.1. In fact, stating also to the Gartner glossary, Big Data is not only how the volume of information is increased, but it concerns also how the data is managed:

> **Big Data** is high-volume, high-velocity, and/or high-variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision-making, and process automation.

The central point of Big Data is that they are more expensive in terms of management, access, and of course, distilling information. Because of that, the cloud infrastructure has a fundamental role in data analysis, and it also affected the way analysis tasks are performed.

Generally, there are four points that concern the Big Data information [7]:

i) **volume**: the amount of data accumulated;

ii) **velocity**: data can pile up really fast, but a desirable feature also processes and examine those data with a high speed;

iii) **variety**: the types of collected information can be very diverse, from structured data (like database information) to unstructured data (such as emails, images, videos, etc.);

iv) **veracity**: being certain that the collecting data contains useful information that can be separate from the noise and the poor quality data.

Usually, it is possible to find a fifth point named **value** but, due to the nature of Big Data, it is easy to guess that the true value of the data is accomplished when the right information is captured. As a consequence, the Cloud infrastructure goes well with the Big Data points and help the users. For example, it lowers the upfront costs with a more flexible and cheap structure (from the client point of view), increasing the accessibility, reliability, and the tasks' automation.

To clarify, manipulating petabytes is not trivial and perhaps, to maintain an infrastructure that supports those levels of volumes is not affordable for everyone. In particular, the cost of elaborate information is much higher than the cost of storing it. Furthermore, CPU and GPU calculus is necessary to extract value from data and Cloud computing enables fast scalability with a readily-available infrastructure that can scale on demand. The pay-as-you-go model contributes to less wasting of resources and lets the users focus more on creating insights and, at the same time, reduces the cost of analytics because the infrastructure is managed by the provider.

As well as that, users can be more creative thanks to the direct access to raw data (unstructured information), and business processes like continuous integration and disaster recovery are made easier by cloud technologies.

Howsoever, the flexibility of the cloud is not the only reason why Big Data is well fitted to that environment. Since they are composed of unstructured information, the paradigm with which they are accessed is not comparable to a query of a database, where the data have a precise number of features, and they are ordered in memory. In Big Data a standard de facto method to extract information is the MapReduce [8] programming model. Shortly, it is a mechanism to apply a function to all entries and then filters depending on some constraints to produce the result. This practice has influenced so much the developer community also thanks to a famous framework born to accomplish that task, today named Apache Hadoop [9]. In detail, Hadoop is a collection of open-source software utilities that allow solving the problem that involves a massive amount of data and computation. Its distributed storage is prepared to be used with the MapReduce paradigm and the tools Apache Spark is the Swiss knife that allows the analysis of Big Data in a distributed system.

The popularity of HDFS, the Hadoop file system, encouraged the search for new systems to store and analyze large quantities of files. Unlike databases, the main storage types in Big Data are the following:

1. **file storage**: where data is stored as a single piece of information inside a folder;

2. **block storage**: where the original data is split into several pieces with a unique identifier, and they are stored according to rules of convenience;

3. **object storage**: is a flat structure in which files are broken into pieces and spread out among hardware.

The latter has become the most used solution in the past years. Also, big companies like Amazon implemented their own solutions, for example, the S3 storage system of Amazon. There are open source solutions too, like MinIO, that use the block storage method to manage the data. In conclusion, both the storage and the computing have obtained more simplicity, performance, and scalability from the cloud environment and that is the reason why nowadays Big Data works closely with this ecosystem [10].

1.3 Data caching in cloud systems

Since the initial spread of the WWW, several solutions to enhance the performances and face the exponential growth of the internet has been hypothesized, tested, and put into practice [11]. In a network, there are several actors that can take advantage of using a cache system, starting from the servers to the clients, but also intermediary objects like the proxies or load balancers that redirect the traffic.

Despite the field of application and the type of data, the benefits of using a caching system impact both, the end-users, and the infrastructure. Thus, the WWW adopted this method to mitigate the constant growth of web contents (E.g. [12]), and the main earning points that makes so attractive the use of a cache [13] are:

- it reduces network bandwidth usage;
- it reduces user-perceived delays;
- it reduces loads on the origin server.

However, the main problem of cache content management is the replacement strategy [14] and this applies in all the contexts where a cache is used. Furthermore, the problem becomes more complex if we talk about Big Data, where there are numerous heterogeneous files with possibly huge sizes and a great number of characteristics.

The main policies adopted for the caching content management are LRU (Least Recently Used, which serves as the golden standard) and LFU (Least Frequently Used). These policies are known to be easy to implement, and they do not require large computational calculations (that's why they are preferable in embedded environments). Although, they are known to under-perform in the Big Data environments because they were initially designed for evicting fixed-size pages from buffer caches [15].

Another type of algorithms used for their simplicity are those based on file sizes and choose to remove a file from the biggest (Size Big) or the smallest (Size Small) file stored in the cache.

Nevertheless, the main characteristics that influence the replacement process are still the same [16, 17, 18] and they are related only to the file (or object) requested f:

- **recency**: time of (since) the last reference to f
- **frequency**: number of requests of f
- **size**: the size of f

In conclusion, though the cache concept was born from a specific environment with different targets, we apply this paradigm in the most varied contexts to mitigate the high demand for a huge amount of data. This goes perfectly with the Big Data concept and the current world of cloud computing, where the information size explosion seems to have no end.

1.4 Data lakes

Data lakes emerged in the last decade to solve the problem of growing data. The future of data management has to deal with Big Data, a huge mole of information from several sources and various formats. Organizations need a scalable solution with a low cost and a unique place where to put their data to analyze later. The data lake seems like the answer: an ideal way to store and analyze enormous amounts of data in one location. However, the most collected information for decades was structured data that do not fit well with the new approach used in data lakes, where it is preferred to semi-structured or unstructured memorization of the data.

About the original meaning of the name, according to Dixon [19], it is a large body of water, into which new water streams from many channels, and from which samples are taken and analyzed. The water represents ready to use data where users have a unique access point. This concept has changed during the years due to the evolution of cloud computing and data store technologies and, as King said in [20], it becomes a more complex infrastructure that manages also unstructured data ignoring the information content.

The main difference between a data lake and a data warehouse is that in a data warehouse, the data is preprocessed and categorized to a fixed structure when it's stored. Instead, a data lake stores information of any type and structure in their native formats and make those data available for future reporting and analysis. Hence, a key point to better understand this type of architecture is to think that it enables a comprehensive way to explore, process, and analyze petabytes of information arriving from multiple data sources.

The way to access and process information is more dynamic in a data lake [21, 22], especially for data analysis patterns. As a consequence, if the data stored has no value or has a bad quality, the data lake can quickly become a data swamp: unorganized pools of data that are difficult to use and understand.

The design of a data lake may be different depending on the domain it is applied to and the background technologies used.

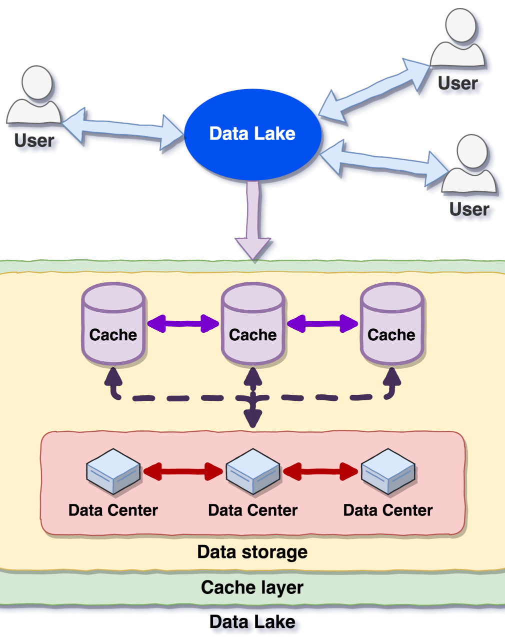


Figure 1.3: Data lake model schema with a focus on the internal organization of the lake

In the Figure 1.3 there is a schematic view of a data lake architecture used in this work. In that picture, several data centers, that could be geographically distributed, are the core of the system. The main storage is surrounded by a cache layer composed of several caches that serve the user requests.

The cache layer could be distributed in distinct regions in which each cache instance has not to have duplicate files within its neighbors' caches. Therefore, managing the cached content is a key problem in this data lake architecture.

In terms of caching data management, this project wants to solve a problem that has many affinities with a Content Delivery Network (CDN [23]) and also with the web content caching (especially with video file streaming [23]).

Background on Machine Learning

Contents

2.1 Machine learning 16
2.2 Reinforcement Learning 20
2.2.1 Concepts and terminology 21
2.2.2 Approach details 24
2.2.3 Algorithms 27

This chapter provides a global picture of Machine Learning, and more specifically, on the Reinforcement Learning technique used in this work.

2.1 Machine learning

The Machine Learning (ML) term was coined by Arthur Samuel in 1959 to represent the field where machines can learn how to do a specific task from experience, without being explicitly programmed. The process is quite similar to the real meaning but, as we can say that Artificial Intelligence is not intelligence, Machine Learning is not learning at all [24]. ML consists of building and adapting models with the support of data analysis. As a result, it creates a useful experience that improves the ability of the machine to make predictions. The adoption of ML techniques in the past 10 years dramatically increased to improve machine problem-solving.

ML improvements have also a direct impact on Artificial Intelligence because it is strictly related to learning automatically through experience. In fact, the subtle difference of Artificial Intelligence is that the machine has to understand and react autonomously, especially when it faces new situations, but it still remains tied to what it has learned previously. That is the insoluble link between the two fields and ML could be considered the first brick of the process because it tries to extract insight from data.

Because of the current computing power and the spread of the information and tools through the Internet, in modern days ML affects millions of people and is present in a significant number of fields. The presence of open source tools contributes to the evolution of the techniques, bringing unimaginable results to theories of about 50 years ago, such as neural networks (1957) and nearest neighbor algorithm (1967).

Therefore, we can describe the ML as a process of solving a particular problem using some raw data (taken from nature, handcrafted, or generated algorithmically) to build a statistical model that, somehow, will be used to solve the practical problem. The key concept remains trying to pull out knowledge that is not obvious from looking at the data.

Generally, ML uses statistics but lies at the intersection of computer science, engineering, and statistics and also can appear in other disciplines. Because it is applied in many fields there is a tremendous amount of data created by humans that can be used, and also nonhuman sources of data are emerging quickly. Data from sensors are coming as a deluge these days, just think of the several sensors we have in a mobile phone and the related data that they generate. As a consequence, ML will be important in the next future, especially in the current jobs. Quoting Hal Varian, the chief economist at Google:

> The ability to take data—to be able to understand it, to process it, to extract value from it, to visualize it, to communicate it—that's going to be a hugely important skill in the next decades, not only a the professional level but even at the educational level for elementary school kids, for high school kids, for college kids.

Apart from this, there is a common terminology that identifies the Machine Learning field and that it is useful to understand the involved processes [25]:

- **feature**: called also *attribute*, it represents a measured value, and it is related to the object it is analyzed, for example its weight or color. Features can be numeric, binary, or sort of enumeration;
- **instance**: a collection of features that identify a specific sample or object. If we consider the columns of a database as the features, an instance represents a single row;
- **classification**: one of the basic tasks of ML. It solves the problem to distinguish an object from another, predicting what *class* an instance of data should belong using the features available;
- **regression**: the task of predicting a numeric value. It is another common task in ML;
- **training set**: example set of quality data used to train the machine. It is composed of several examples that are instances of the desired objects. In this set, the *target variable* (attribute) is the machine goal, the desired prediction. For example, in classification, it is the object class;
- **test set**: a sample of data with missing target variables. They should be predicted by the machine;
- **knowledge representation**: the measure of what the machine has learned. This can depend also on the algorithm or the technique used. For

example, in classification, could be the accuracy, that is how wrong the machine has predicted the classes.

Hence, Machine Learning has many algorithms that solve different tasks with respect to problem goals. Usually, we can classify the ML tasks into three categories represented by how much feedback or information they have from the analyzed system:

- **unsupervised**: here the machine tries to create a model that takes several features and transforms them into valuable information starting from a collection of unlabeled example (e.g. clustering method corresponds to assign a cluster id to a set of features without knowing a priori information about their membership). Thus, starting from an unlabeled input the machine has to create a structure out of the inputs on its own;
- **supervised** (and also semi-supervised): in this case the model has access to a collection where entries have information about their class. The machine has to learn how to distinguish the differently labeled data to infer from features of the correct class. In practice, it creates a mathematical function that relates input variables to the preferred output variables. In the semi-supervised learning we have some entries without labels, used to improve the predictive ability during the training phase;
- **reinforcement**: in this subfield of Machine Learning we have a machine that interacts directly with an environment and receives rewards from it with the goal to learn the best policy (action) to do in a specific situation. This mechanism of rewards and punishments makes the machine explores some solutions and exploit what it has already learned to find a balance that makes to reach the goal.

In this work are used techniques related principally to the last class but in the next chapter, there will be a detailed discussion on Reinforcement Learning algorithms. Nevertheless, we have to mention that ML offers different approaches to solve a wide variety of problems, such as neural networks, deep learning, cluster analysis, decision tree learning, support vector machine, etc [24]. Also, those techniques are often combined to solve a specific task, but we will not cover this topic here.

Certainly, it is not trivial to choose the right algorithm or technique, and it depends on several problem factors. Nevertheless, the steps present during the development of Machine Learning applications are [25]:

1) **collect data**: the first action, of course, is to collect samples and take measures;
2) **prepare input data**: process the data to prepare them for the algorithm used. It could be necessary to filter or remove attributes, to scale or normalize values and also simply change the data format. This task is also called *pre-processing*;
3) **analyse the input data**: an optional step where it is possible to check if there is no garbage coming or if the data representation is what it should look like. Usually, it is a step consumed by humans to verify the data processing before the training phase;
4) **train the algorithm**: where the Machine Learning takes place. This will not be done in unsupervised learning because there is no target value;
5) **test the algorithm**: the phase where the knowledge learned is put into action and were to evaluate the algorithm;
6) **deploy**: when the algorithm is used in a real use-case and not only with the test set data. In short, it put into practice the results of previous steps.

All of these stages compose the basic Machine Learning Engineering (MLE) cycle [26], where a complex computing system is build using traditional software engineering and ML tools and techniques. The cycle covers all steps from data collection to model deployment for end-user customers.

Besides, there are several steps not mentioned here that encompass the management of the ML as a service and that fall in the field of *DevOps*. These involve the phases to maintain and serve the final application, and they are at the same level as the deployment step. Indeed, we are focusing on the data analyst point of view and not on service development. The process steps

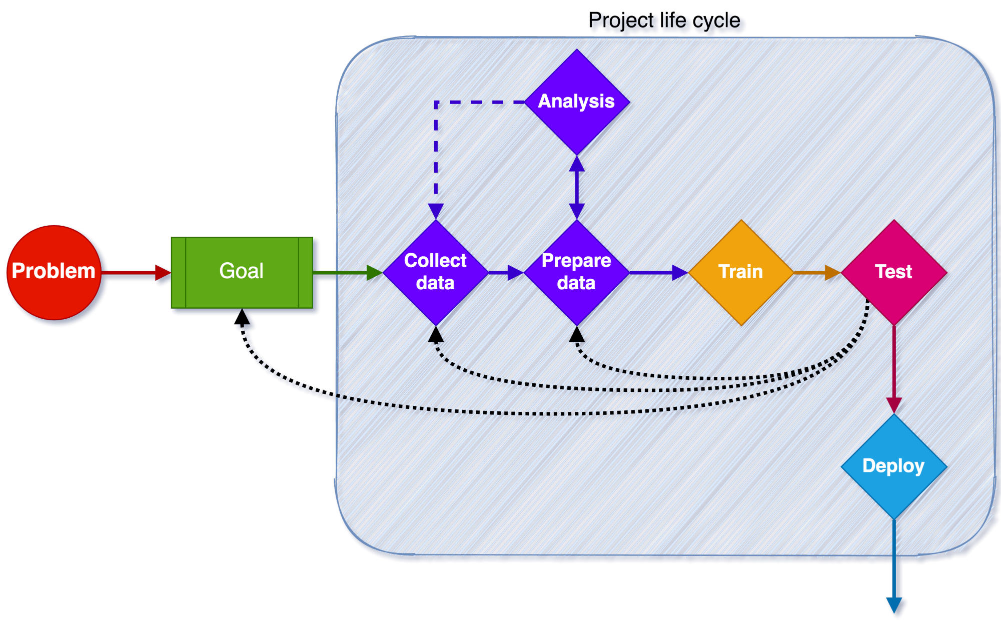


Figure 2.1: Machine Learning engineering cycle, from problem issue to the deployment of a solution

mentioned before are schematized in Figure 2.1. In the schema are also visible the possible interaction between the phases and the cyclic paths to improve the model on each iteration. In fact, it is normal to arrive at a model that triggers some modification of the previous phases because its evaluation in the test phase was poor.

2.2 Reinforcement Learning

In our lives, we experience the process of cause and effect that put the base of our knowledge since we are born. Clearly, it is easy to understand that the consequences of our actions give us information about ourselves and the environment in which we are. In the field of Machine Learning, this aspect is used to produce an agent that can interact with a particular environment, and this kind of learning is named Reinforcement Learning (RL).

The RL approach is different from the other types because it puts the learner in a situation of trial and error, where the consequences of its actions have an impact on the environment and also on the problem's goal. Furthermore, the agent is punished or rewarded on the basis of its behavior, with the idea that, in the future, it will prefer good actions and forego unwanted behaviors.

As a consequence, RL is focused on goal-directed learning from interaction. For this reason, it differs from Supervised Learning because it does not use a set of labeled examples provided by a knowledgeable external supervisor. Supervised Learning has the object to extrapolate or generalize the responses to acts correctly in situations not present in the training set.

Furthermore, RL differs from Unsupervised Learning because it does not search for hidden structures in data collections. Still, it is quite similar because it does not rely on examples of correct behavior.

Thus, RL takes place in a specific problem domain where we have a dynamic environment and a clear goal to achieve. The knowledge the agent builds for himself through the experience is, of all the forms of machine learning, the closest to the kind of learning that humans and other animals do. Consequently, many of the core algorithms of RL were originally inspired by biological learning systems. Nevertheless, the challenges that arise in reinforcement learning, and not in other kinds of learning, is the trade-off between exploration (that is trying new things) and exploitation (namely applying what was learned). The balance between them remains an unresolved problem and one of the most delicate parameters to set.

Apart from this, RL has fruitful interactions with other engineering and scientific disciplines and can fit a variety of problems (e.g. [27]), also because it could be an independent component of a larger behaving system.

2.2.1 Concepts and terminology

The main actors involved in a Reinforcement Learning approach are the *agent* and the *environment* [28]. Briefly, the environment is the world where the agent lives and interacts with. In each step of interaction with the world, the agent observes the state of the environment and decides the action to take. Those actions can modify the state of the environment but, the world can also change on its own.

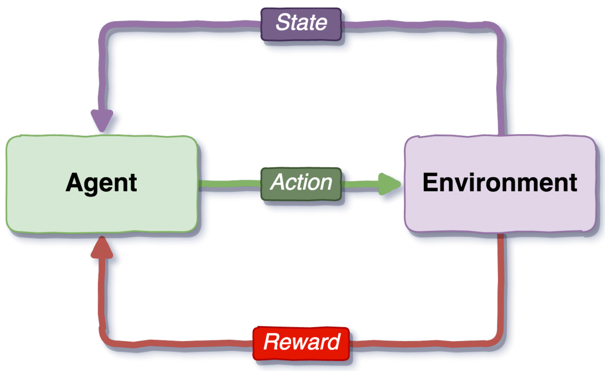


Figure 2.2: Generic reinforcement learning interaction schema between the agent and an environment

As a result, the agent perceives some rewards from the environment, which tells the agent how good or bad is the current state of the world. In Figure 2.2 there is a simple schema of the actors' interactions.

The agent main goal is to maximize its cumulative rewards, named *return*. Thus, a RL method it's just a way with which the agent can learn behavior that leads to the problem goal.

Starting with this big picture, it is possible to define better all the components and the concepts behind it:

- **state**: it represents a full description of the world in a particular step of the interaction. Usually, it is indicated with s and there is no hidden information, the agent knows all the world attributes. When the state is a partial description of the world, thus there is some information omitted, it is better know as an *observation* o. In this case, the world is partially observed instead of fully observed;

- **actions** and *action spaces*: depending on the world, there are several actions that the agent can do. The set of actions allowed by the environment is named *action space*. The actions' space can be discrete, thus described by a finite number of actions, or continuous, where the actions' value is represented by a real-value and there is a finer grane of control over the action impacts;

- **policies**: characterize the brain of an agent because they indicate how an agent reacts, specifying which action to take. These rules identify the agent behaviors. If the rule is deterministic it is indicated with the relation $a_t = \mu(s_t)$, where the rule μ is used to select the action a from the state s at time t. Another type is stochastic rules denoted with π and indicated with the following relation: $a_t \sim \pi(\cdot|s_t)$. In the latter case the function π outputs a probability action distribution over the state s at the time t;
- **trajectories**: a trajector τ is a sequence of states and actions in the world: $\tau = (s_0, a_0, s_1, a_1, \dots)$. The initial state s_0 could be any of the state available in the start-state distribution (a set of all possible start-state) and usually is randomly sampled;
- **state transition**: indicates what happened from state s at time t and the state s' at time $t+1$. This involves the environment natural laws and the application of the agent's action, that can be done in a deterministic way: $s'_{t+1} = f(s_t, a_t)$, where the transition function f applies the action a to the state s at time t and we have the next state s' at time $t+1$ as result. Otherwise, if it is stochastic, the relation is the following: $s'_{t+1} \sim P(\cdot|s_t, a_t)$. It should be noted that the stochastic approach is often used when the environment is uncertain, and it is not possible to choose an action directly;
- **reward** and **return**: a reward r at time t is a value obtained by the function R that is dependent on the current state s_t, the next state s_{t+1} and the action a_t: $r_t = R(s_t, a_t, s_{t+1})$. As mentioned before, the **return** is the sum of all rewards taken by the agent. It is possible to indicate that with $R(\tau)$, because the agent tries to maximize the reward in a trajectory. Thus, it is possible to distinguish two kind of return:
    1) **finite-horizon undiscounted return**: that is the sum of rewards in a fixed size window, denoted as $R(\tau) = \sum_{t=0}^{T} r_t$;
    2) **infinite-horizon discounted return**: where there is a discount factor $\gamma \in (0, 1)$ that emphasize the recent rewards: $R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$.

The notations explained above are useful to understand the environment formalization. Consequently, the RL world obeys the Markov property, where the transitions only depend on the most recent state and action, with no prior history. In fact, the RL can be seen as a Markov Decision Process (MDP), that is a 5-tuple $\langle S, A, R, P, p_0 \rangle$ defined as follows:

- S is the set of all valid states of the world;
- A is the set of the valid actions;
- R is the reward function that goes from $S \times A \times S$ to $\Re$;
- P is the transition probability function that gives the probability to arrive in state s' if the action a is applied to the state s: $P(s'|s,a)$;
- p_0 is the starting state distribution

2.2.2 Approach details

Independently of the return measure and the policy, the target of a RL is to select a policy that makes an agent behave in a way to maximize the expected return. If the transitions and the policy are traited as stochastic, we can define the probability of a trajector τ of lenght k as follows:

$$P(\tau|\gamma) = p_0(s_0) \prod_{t=0}^{k-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t) \tag{2.1}$$

Consequently, the expected return J over the policy π can be defined as:

$$J(\pi) = \int_{\tau} P(\tau|\pi)R(\tau) \tag{2.2}$$

The previous equation can be denoted also as the expected return E:

$$\underset{\tau\sim\pi}{E}[R(\tau)] \tag{2.3}$$

Therefore, the optimal policy π^*, that is the central optimization problem in a RL approach can be written as:

$$\pi^* = arg\max_{\pi} J(\pi) \tag{2.4}$$

To monitor and have feedback on the agent behavior it is often used a *Value Function* to evaluate the state or the state-action pair. The output value is the expected return E when starting with a certain state or state-action and then follow a particular policy forever after. There are four main functions to mention that are useful to get the value of a particular state and also its optimal value when following a policy:

- **on-policy value function**: denoted as $V^{\pi}(s)$ it gives the expected return E if starting from the state s and then always follow the policy π:

$$V^{\pi}(s) = \underset{\tau \sim \pi}{E} [R(\tau) | s_0 = s] \tag{2.5}$$

- **on-policy action-value function**: denoted as $Q^{\pi}(s, a)$ it gives the expected return E if starting from the state s, take the action a and then always follow the policy π:

$$Q^{\pi}(s, a) = \underset{\tau \sim \pi}{E} [R(\tau) | s_0 = s, a_0 = a] \tag{2.6}$$

- **optimal value function**: denoted as $V^{*}(s)$ it gives the expected return E if starting from the state s and then always follow the optimal policy in the environment:

$$V^{*}(s) = \max_{\pi} \underset{\tau \sim \pi}{E} [R(\tau) | s_0 = s] \tag{2.7}$$

- **optimal action-value function**: denoted as $Q^{*}(s, a)$ it gives the expected return E if starting from the state s, take the action a and then always follow the optimal policy in the environment:

$$Q^{*}(s, a) = \max_{\pi} \underset{\tau \sim \pi}{E} [R(\tau) | s_0 = s, a_0 = a\ ] \tag{2.8}$$

The value function equations respond to a dynamic programming method that breaks the decision problem into smaller sub-problems, in particular, they follow Bellman's principle of optimality that is [29]:

> Principle of Optimality: An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

If we consider the stochastic process of interaction with the environment and that principle, value functions could be rewritten. If we defined the following shorthand:

— $s' \sim P = s' \sim P(\cdot|s,a)$ (where the next state s' is sampled from the environment's transition rules)

— $a \sim \pi = a \sim \pi(\cdot|s)$

— $a' \sim \pi = a' \sim \pi(\cdot|s')$

It is possible to write the on-policy value function Equations (2.5) and (2.6) as:

- **on-policy value function**:

$$V^{\pi}(s) = \underset{\substack{s' \sim P \\ a \sim \pi}}{E} [r(s,a) + \gamma V^{\pi}(s')] \tag{2.9}$$

- **on-policy action-value function**:

$$Q^{\pi}(s,a) = \underset{s' \sim P}{E} \left[r(s,a) + \underset{a' \sim \pi}{E} [Q^{\pi}(s',a')] \right] \tag{2.10}$$

Respectively, the optimal value functions Equations (2.7) and (2.8) should be:

- **optimal value function**:

$$V^{*}(s) = \max_{a} \underset{s' \sim P}{E} [r(s,a) + \gamma V^{*}(s')] \tag{2.11}$$

- **optimal action-value function**:

$$Q^{*}(s,a) = \underset{s' \sim P}{E} \left[r(s,a) + \max_{a'} Q^{*}(s',a') \right] \tag{2.12}$$

The main difference in the Bellman equation value functions, respectively for the on-policy and the optimal, is the presence of $\max$ over actions, which reflects the fact of the agent has to pick whichever action leads to the highest value.

2.2.3 Algorithms

RL algorithms are divided in two categories, *model-based* and *model-free*. In the last type, agents do not need previous knowledge of the environment and interact directly with it to learn the optimal policy. Instead, the other type has a predefined representation or model of the environment with which the agent interprets the inputs. It is possible to see the model as a function that predicts state transitions and rewards. Thus, the model-based approach is more difficult to design due to the target of accurately reflecting the real world, and often a ground-truth model of the environment is not available.

Moreover, a sensitive aspect is what to learn, such as i) policies, ii) action–value functions (Q-functions), iii) value functions or iv) environment models.

However, in a general reinforcement learning task, the goal is to map situations to the actions that are best in those situations, but we have also to take into account the extent of the agent's actions. For example, when we have multiple choices, and we want the agent to learn which is the best thing to do, we are using the *associative search task*, so called because it involves trial-and-error learning to search for the best actions and association of these actions with the situations in which they are best. This kind of approach is intermediate between the k-armed bandit problem and the full reinforcement learning problem because it involves learning a policy, but each action affects only the immediate reward. To have full reinforcement learning we have to allow the actions to affect the next state.

The central idea of the modern RL is the temporal-difference (TD) learning, a combination of Monte Carlo ideas and dynamic programming ideas. This method updates estimates based in part on other learned estimates, without waiting for a final outcome (we say it bootstraps).

In this context, there are several algorithms that could be used [28] and the Q-Learning is a family of algorithms that approximates $Q_0(s, a)$ of the optimal action-value function $Q^{*}(s,a)$. This kind of approximators follow the Bellman principle Section 2.2.2 and they use as objective a function based on the Bellman equation (Equation (2.12)).

The Q-Learning implementation consists of a loop for each step of the problem episodes, where it updates the action value for the current state ($Q(s,s)$) observing the consequences of the action on the environment (that leads to the next state s'). The algorithm procedure is shown in Algorithm 1. The core part is the update step at a certain time t.

Algorithm 1: Q-Learning [30]

```
Initialize $Q(s,a)$, for all $s \in S^{+}, a \in A(s)$, arbitrarily except that
  $Q(terminal, \cdot) = 0$;
foreach episode do until $s$ is terminal
    Initialize all $s \in S$;
    foreach step of episode do
        Choose $a$ from $s$ using policy derived from Q;
        Take action $a$, observe $r$, $s'$;
        Update $Q(s,a)$ with $r$;
        $s \leftarrow s'$;
```

Assuming that s is the state at time t, s' is the state at time $t+1$ and r is the reward taken at time $t+1$, the Q-Learning update function can be resumed in the following equation:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma \max_{a} Q(s_{t+1},a) - Q(s_t,a_t)\right] \tag{2.13}$$

As can be deducted from the Equation (2.13), it makes no decisions to select the best next action, but it simply selects the maximum value from the next state s_{t+1}. This kind of behavior is named off-policy, and it enables the possibility to use data collected at any point during the training, regardless of the agent choices and the environment exploration.

An important part of such an equation (Equation (2.13)) is the r component: the reward. Several algorithms, included Q-Learning, uses a mechanism of delayed rewards that simulates what we can see in nature with the stimuli.

In detail, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards, such as the mechanisms of punishment influence the current and next decisions of an animal that reacts to a stimulus.

The Q-Learning Algorithm 1 is one of the most adopted due to its implementation simplicity: it can be applied online, with a minimal amount of computation, to experience generated from interaction with an environment.

The action selection at the time t can be performed with several algorithms. The most used is the $\epsilon greedy$ mechanism, that is a simple method to balance exploration and exploitation by choosing between them randomly. In details, there is an $\epsilon \in [0, 1]$ value decreasing during the time and the action for the state s_t is selected from Q as follows:

- $maxQ_t(a)$ with probability $1 - \epsilon$
- random action a with probability ϵ

Thus, during the step loop of the Algorithm 1, the agent will have less chance to randomly select an action (exploration), and instead, it will apply what it learned (exploitation).

Moreover, the algorithm is simple to interpret because the action taken by the agent is understandable from the Q values, and it is given by the following equation:

$$a(s) = \arg\max_a Q_\theta(s, a) \tag{2.14}$$

In conclusion, the Q-learning approach tends to be less stable because it indirectly optimizes the agent performance by training a local Q_θ that satisfies the optimal choice. However, this method takes advantage of reuse data more effectively than other optimization techniques due to its off-policy update.

Related works

In this section, the solutions already available in the literature are discussed, addressing the main difference with respect to the presented work.

There will be presented several examples as a reference for similar problems.

As presented in Chapter 1, Data Lake is an environment where data are principally exchanged through the network. Also, there are similarities with the World Wide Web content management, where the caching systems have a role to improve the reliability of the contents. The scenario is also similar because in both, independently of the field of application or the users, there is an ever-growing data size and demand.

As already mentioned, the most used strategies adopted to manage the caching policies are LRU and LFU [14], that works for the majority of the caches. Even they are simple, these strategies cannot deal with the dynamics of content popularity and network topologies. Recent efforts have gradually shifted toward developing learning and optimization-based approaches, that can smartly manage the cached content. The problem is not to be taken lightly due to the increasing storage capacity per decade [31] which consequently allows a significant increase in data production. The amount of data can vary depending on the field and the Data Analysis takes advantage of this huge volume of information to produce new knowledge. In the Big Data field, it is possible to notice the limits of golden standard policies like LRU [15].

Consequently, several Machine Learning techniques have been proposed to improve file caching and, in general, better content management.

In [32] the authors propose to train a Deep Neural Network in advance to manage better the real-time scheduling of cache content into a heterogeneous network. The target was to minimize the network energy consumption and reduce transmission delay via caching placement and content delivery optimizations. The better content management aims to have a direct impact on the infrastructure communication, indirectly improving the user experience with a better popular content serving. Besides the objective is quite common in caching management, the approach is not generic and its application field is very specific: e.g. it uses a cost function related to the network repeaters. Furthermore, it is not set in the Data Lakes field, as a consequence, the study concerns a low number of files that are requested with respect to a real use-case involving Big Data.

With the increase in traffic and transmissions, some network objects like the routers have now the capability to cache objects and serve directly upon user requests. This approach led to infrastructure configuration like the content delivery network (CDN) where the proxy servers are distributed spatially relative to end-users with the capacity of caching popular data for better content delivery. Also, with this configuration, there are trends of objects that are requested once ($\sim 70\%$ in [33] web content analysis). Thus, the most difficult decision remains which object to cache and evict. For this purpose, various solutions have been tried.

In [34] a deep recurrent neural network is applied to predict the cache accesses and make a better caching decision. The purpose was to predict the characteristics of an object ahead of time. In particular, their goal was to predict the popularity of an object to reduce the network avoiding the eviction of future popular objects. As a consequence, they expected to increase the number of cache hits. Howsoever, that project concerns cache and synthetic dataset sizes far from the Data Lake volumes.

Another example of prediction approach is in [35], where predictions are used to optimize the eviction of a cache with a fixed size. However, this cover the aspects to mitigate the the performance when there is a not perfect oracle, that is often the situation in a Data Lake context.

Big Data and Data Lakes are possible thanks to distributed file systems. In [36] there was an attempt to automate the caching management of a distributed data cluster using the Gradient Boosting Tree. But, in that project they have a small cache size with file distributed as chunks in the storage system (Hadoop Distributed File System, HDFS) and still, the number of unique files is not high in comparison of a Big Data target. Moreover, it depends on a technology related to data distribution. Still, that study tried to lower the infrastructure load with a better caching prediction.

It is evident that the environment is a critical aspect to take into account when we talk about caching management and, due to the dynamic set, there is needed of a more flexible and autonomous solution that can adapt itself to new solutions.

Recently, techniques that uses the Reinforcement Learning approach are emerging. For example, in [37] a Deep Reinforcement Learning (DRL) is used to cache the highly popular contents across distributed caching entities in the context of content delivery networks (CDN). However, even if the system allows an online adaptation the experiment uses a small number of files that have to be placed optimally in the hierarchical cache system.

There are also DRL approaches like the Wolpertinger architecture [38] used in [39] that tries to optimize the cache hit rate. But, in that case, the authors assume that all the files in the cache will have the same size and this is not usual in a Data Lake context where the file sizes could differ, and thus, they are important for better data management.

Thus, the problems solved by the cited works are not comparable in size with the ones that are the target of this project. In Data Lakes there are a higher number of files to manage and a huge amount of requests per day (more details in the use case). Moreover, a direct comparison is not trivial because they mostly use synthetic datasets not released online and generated with a specific distribution that matches their use cases, for example, the network traffic or HDFS data requests. Conversely, this thesis is discussed as a generic approach to the problem that respects the Data Lake model. However, the core of caching management lies in when and what to add or evict, and they are aspects of the same importance.

Problem description

Contents

4.1 Definition and targets 36
4.2 Context 39
4.2.1 Data-sets 44
4.3 Simulation environment 50

In this section, the problem of this project will be addressed in detail. Data sources used will be discussed as well as the origin of those data. Moreover, there will be additional information on the real-world use case that is approached with the intent to give an idea of the size and the complexity of the problem. The topic context is focused on the Compact Moun Solenoid (CMS [40]) computing model and its evolution towards Data Lakes.

4.1 Definition and targets

This thesis project aims to improve the cache content management in the context of Data Lakes (Section 1.4). The cache can be seen as an intermediary object between the client and the storage. A client can be anything is requesting a file: a real user, a program, or any piece of software. The important thing is to satisfy the highest possible number of requests. In particular, as shown in the next chapters, the thesis is contextualized in a specific use case that is the High Energy Physics (HEP) Data Analysis and the caching layer could have a high influence on user-experience. Besides the context, this work addresses the problem using agnostic approaches. Because the functionality of the cache is similar to other contexts, better content management should bring unaltered data access and easier infrastructure management. Unaltered data access means that the cache has a minimal and no perceptible impact on the data flow and the easier management concerns the way the caches are built and distributed to mitigate the access to the main storage.

From the client's point of view, the request will be accepted by a Data Lake. The caching layer takes care of the underground work needed. The possible choices by the caching layer are mainly three:

- If the file is already in the cache memory, the cache will serve it;
- If the file is not stored in the cache:
    - The cache can retrieve the file from the storage and then serve it from the memory (Figure 4.1);
    - The file is served with the cache as proxy, without storing the file (Figure 4.2);

Consequently, the Data Lake environment forces the requests to pass through the cache system to stress the caching mechanisms. For this reason, this thesis stresses caching content management as the main target of the project. This work exploits machine learning techniques in order to optimize and automate cache management. The main motivation is to reduce as much as possible the hardware resources needed for an effective cache. In other

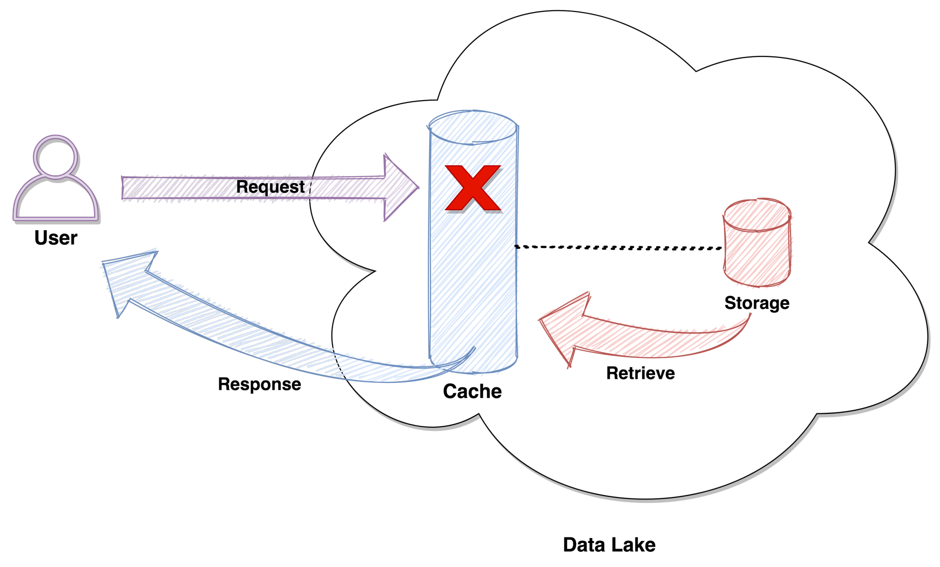


Figure 4.1: Action schema that depicts when the cache retrieves a file on miss

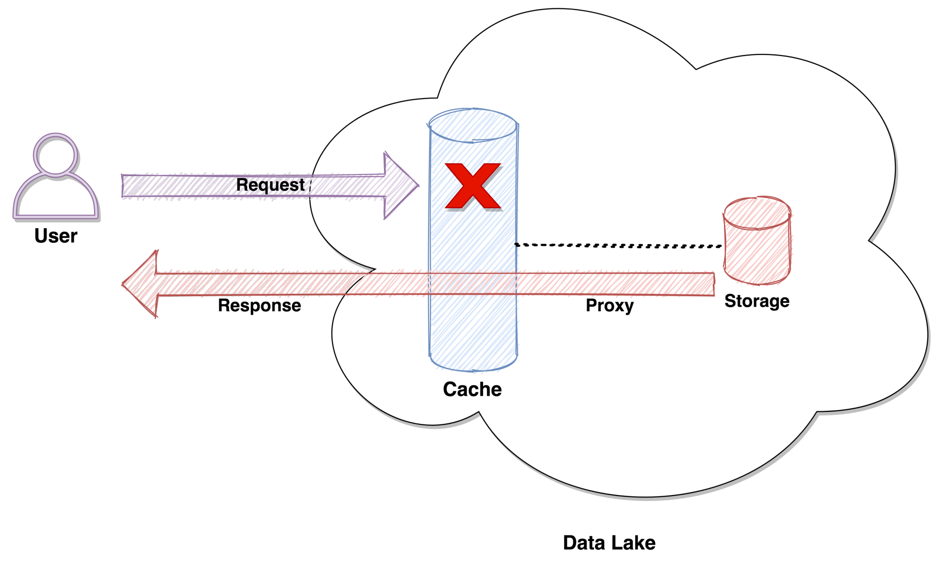


Figure 4.2: Action schema that shows when the cache is used as a proxy to serve a missed file

words, the main goal would be to obtain the very same performances by using a smaller cache thanks to an intelligent, autonomous, and not statically configured caching policy management.

Another important issue addressed by this project is how to evaluate the benefits of an approach. For example, since the caching system is a part of the services and affects the flow of data, a possible metric could be a measurement for the Quality of service (QoS). In detail, the desired behavior is to increase the amount of data served by the cache (Figure 4.3). Also, because services are strictly dependent on the network in the Data Lake model, the traffic prioritization and data resource management is the characteristic of the desirable QoS in this project context.

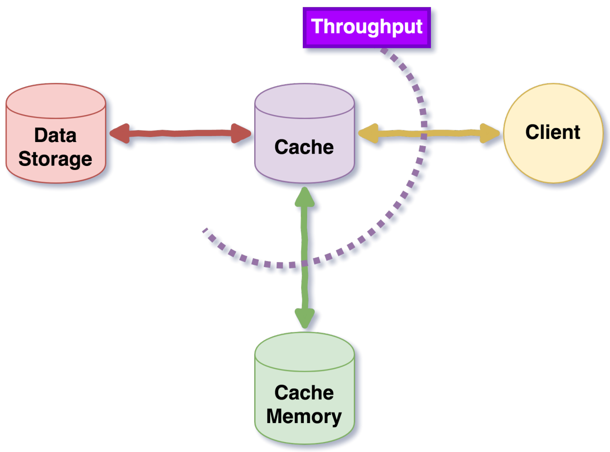


Figure 4.3: Basic environment schema with focus on throughput aspect, a metric to evaluate the caching behavior

As already mentioned, the result of this project aims to be independent of frameworks and technology used. Also, this project is to develop a generic experiment-agnostic model that could be applied to any HEP Experiment adopting the Worldwide LHC Computing Grid (WLCG) Data Lake model (Section 4.2). Thus, it evaluates the different methods in a simulation environment described in the following chapters. For example, the simulation can use also a dataset taken from the real world analysis (described in Section 4.2.1) in the domain of HEP without taking care of the different technologies used to manipulate the data, e.g. the ROOT Analysis Framework [41]. Consequently, the resulting approaches can be deployed in any context despite the technology requirements.

4.2 Context

The Large Hadron Collider (LHC) at CERN (the European Organization for Nuclear Research) is the world's largest and most powerful particle accelerator. The LHC consists of a 27-kilometer ring of superconducting magnets with a number of accelerating structures to boost the energy of the particles along the way. The particle beams inside the LHC are made to collide at four locations around the accelerator ring, corresponding to the positions of four particle detectors: ATLAS[42], CMS [40], ALICE [43], and LHCb [44].

Since the design phase of the experiments, the Worldwide LHC Computing Grid (WLCG) project was established, with the target to develop, deploy, and operate a grid distributed model in order to deal with the huge amount of data to be processed. This global collaboration is composed of around 170 computing centers in more than 40 countries, linking up national and international grid infrastructures. In particular, the referred grid is a heterogeneous and geographically distributed computer resource used to reach a common goal, in this case, the High Energy Physics data analysis.

The WLCG grid system worked nicely since the beginning of the data taking of the four experiments (2009) supporting the continuously increasing demand for computing resources.

Regarding the next critical challenge at LHC, it is expected for 2028, when the next generation of the accelerator, named High-Luminosity Large Hadron Collider (HL-LHC), will be fully operative. The operating conditions of the HL-LHC will increase and thus the projections on data and computing needs will be higher than a factor of ~ 50. Also, just the costs relative to the storage infrastructure are estimated to be around 16 times higher than today (Figure 4.4). With such expectation, it becomes clear that software and computing of the experiment must improve and also the current grid model must be reviewed and improved as well, especially if the system starts to manage Exabytes instead of Petabytes of data.

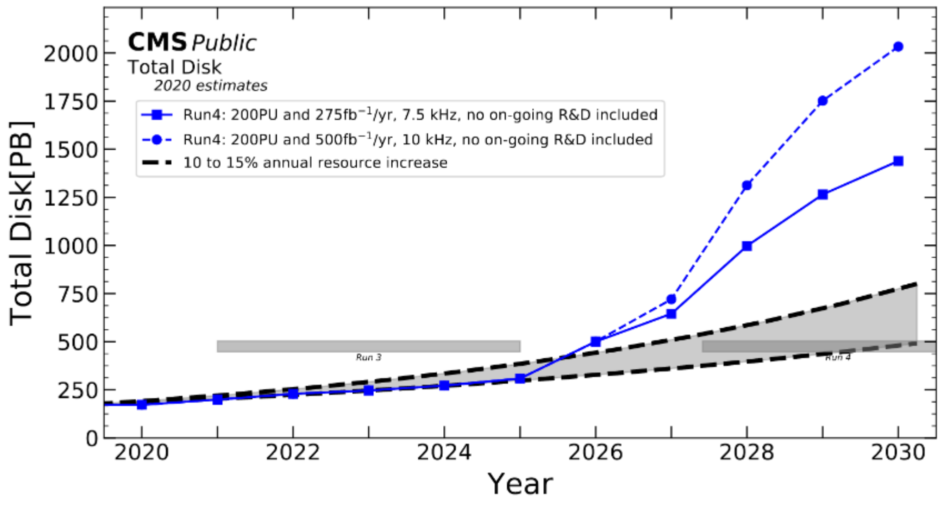


Figure 4.4: Projections of the increase of data storage at CMS

Recently, many architectural, organizational, and technical changes are being investigated to address this challenge and one of the most promising prototypes (especially related to data analysis) is the WLCG data lake model [1, 2], a storage service of geographically distributed data centers connected by a low-latency network. Such architecture is designed with the intent of decreasing the cost and to leverage the economy of scale.

Data Lake at WLCG

The Worldwide LHC Computing Grid (WLCG) collaboration is planning a strategy to face the storage challenge of the next decade still staying with a flat budget for funding. It is clear that the current paradigms of data and computing management cannot sustain future requirements and a breakthrough is needed.

The WLCG community calculated that storage and computing resources necessary in the future with the scheduled upgrades is a factor of ~ 50 and above, despite any current technology evolution. Thus, to bring performance and efficiency without forgetting operation simplification, the community thought and are experimenting with several solutions to address the problem. One of these solutions, especially concerning the storage side, is the Data Lake implementation. In details, this model is meant for i) reduce storage costs,

ii) abstract the data layer and iii) manage better the current facilities. The proposed model is named *"Data Lake straw model"* and it is a declination of the canonical Data Lake that involves not only the information management. A Data Lake in the Straw Model is a group of data and compute centers with no defined borders by construction. The world is divided into several Data Lakes that are associated with network latency. The internal configuration may vary on the scope of the community and experiment needs. Moreover, a Data Lake hosts a distributed analysis working set of data and several caches are used to reduce the impact of latency and reduce network load. The data can be relocated from one Data Lake to another and the most popular datasets may be hosted in more than one Data Lake.

Starting with this brief description, it is possible to analyze the single components that are designed within the Data Lake and their various roles:

- Archive Center (AC): responsible for archive custodial data, the source of information. It should use not performant storage like tape drivers;
- Data and Compute Center (DCC): focused on disk storage faster than AC (mechanical hard disks) but with also computing power, it is used to increase the quality of service (QoS) for analysis tasks;
- Compute Center with Cache (CCC): as the meaning, it is a center focused on calculus without memory to store the data but with a fast cache to serve the analysis jobs;
- Compute center without cache (CCDA): a poorer version of CC with also a lower volume of the cache. It relies especially on the network to access data. It has no disk space and consumes computing jobs taking data from either a CCC or a DCC.

In Figure 4.5 there is an example schema of the Data Lake components. Clearly, the network connections have a significant role in the lake composition and also if we think about the links with other lakes.

Without a doubt, the Data Lake description of Section 1.4 is included in the WLCG model. Even so, this project is stressed about the cache component, and because of this, the schema is further simplified to tackle the specific target of cache data management.

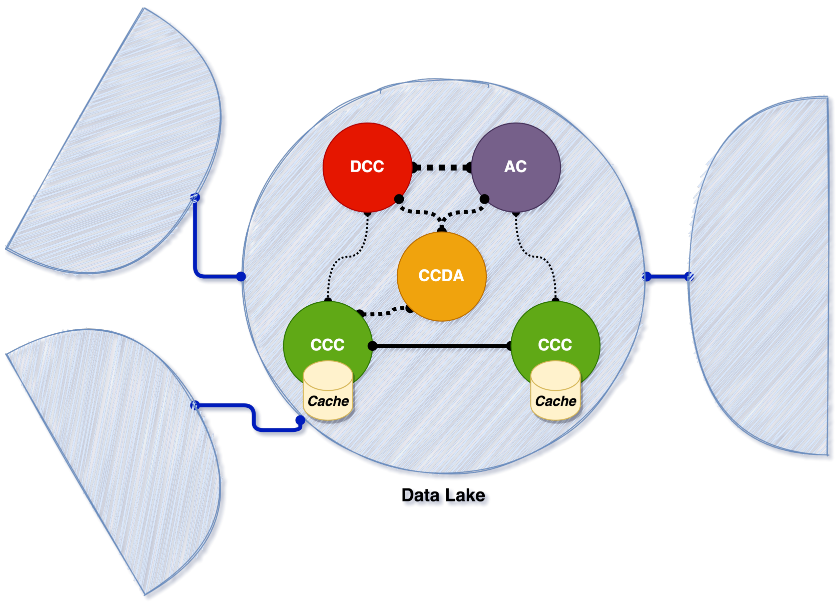


Figure 4.5: WLCG Data Lake schema example with focus on internal components

Data analysis at CMS

In the field of Big Data, there are many sources from business to data science. This work is taken a use case from the latter one, to be specific, in the High Energy Physics field. It is used as a real case experiment in the context of the Compact Muon Solenoid (CMS) data analysis workflow using their request data logs. The CMS experiment is one of the greatest experiments at CERN, and it deploys its data collections, simulation, and analysis activities on a distributed computing infrastructure involving more than 70 sites worldwide.

The current CMS computing environment is a hierarchical distributed infrastructure with a primary Tier-0 center at CERN, supplemented by seven Tier-1, more than 50 Tier-2, and many Tier-3 centers at national laboratories and universities throughout the world. The CMS software running on this high-performance computing (HPC) system executes numerous tasks, including the reconstruction and analysis of the collected data, as well as the generation and detailed detector simulation of Monte Carlo (MC) event samples. Recently, they are experimenting with new models to manage the whole computing infrastructure due to future updates. Thus, it has aroused our attention because they are moving towards the Data Lake model described in Section 4.2 dealing with a huge amount of data (hundreds of PetaBytes each year). Moreover, this case is particularly interesting, and it holds a rich source of information for system tuning and capacity planning: a data log of the past years' analysis collected by the monitoring infrastructure deployed at the Worldwide LHC Computing Grid (WLCG, Section 4.2). Logs are full of data and metadata about the usage of the distributed computing infrastructure by its large community of users. These referred monitoring resources are still growing, and now they are stored into a Hadoop cluster [45] and they occupy nearly 4 PetaBytes.

4.2.1 Data-sets

This work uses information on historical user analysis activities at CMS. It has to be mentioned that also several synthetic datasets are available, but they are out of the scope of this work (more information are in the Appendix A.3).

Regarding the CMS source, the metadata contains details such as the file name, the file size, or the file type. The information contained in the log database refers to the data popularity research made in CMS [46, 47]. The collected information is archived per day and represents all the file requests that should be served.

The activities submitted by the users are composed of different parts schematized in Figure 4.6. Users can launch a task composed of several jobs that require different files.

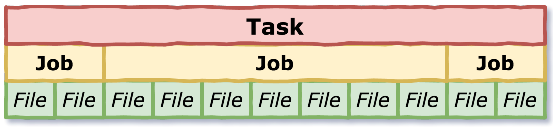


Figure 4.6: Activity composition in a task of a data analysis user

The activities stored in the log files are requests coming from the whole world, depending on where the CMS users are. As a consequence, it was decided to filter those requests by the field *region*, which represents the state on where the request was made. The regions chosen for our experiments are the Italian (IT) region and the United States (US) region. Furthermore, the project focused on the work on the year 2018 requests.

The data analysis per day of the mentioned sources shows that the average number of requests per file is not high, also counting only the files requested more than once. Therefore, the number of files requested per day is comparable to the number of requests per day.

The number of tasks is two orders of magnitude lower than the number of jobs and the number of sites (the origin place of the request which represents a computing center) is much lower than the number of users. This pattern is the same in the two selected regions, the only difference is that in the US we have a greater amount of requests.

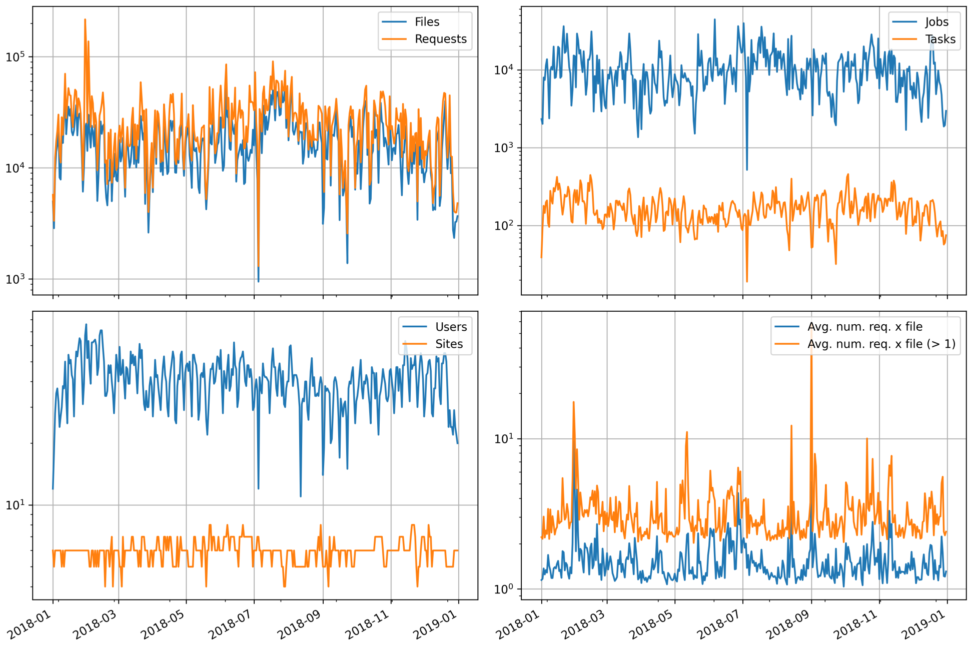


Figure 4.7: Data general statistics of Italian requests in 2018 showing the number of files, requests, jobs, tasks, users, and sites during the year

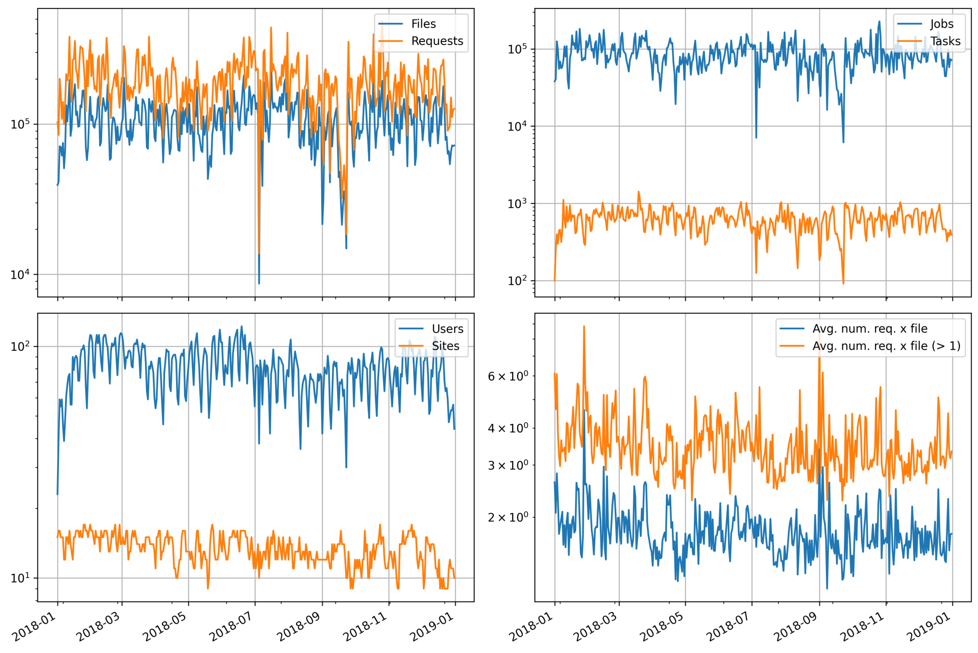


Figure 4.8: Data general statistics of United States requests in 2018 showing the number of files, requests, jobs, tasks, users, and sites during the year

Another sensitive information is the file size and, because of the huge amount of requests, it is very important to correctly manage the data caching. The Figures 4.9 and 4.10 show that we have files from 1GB to 40GB but the most are under 10GB, in particular Between 2 and 4GB.

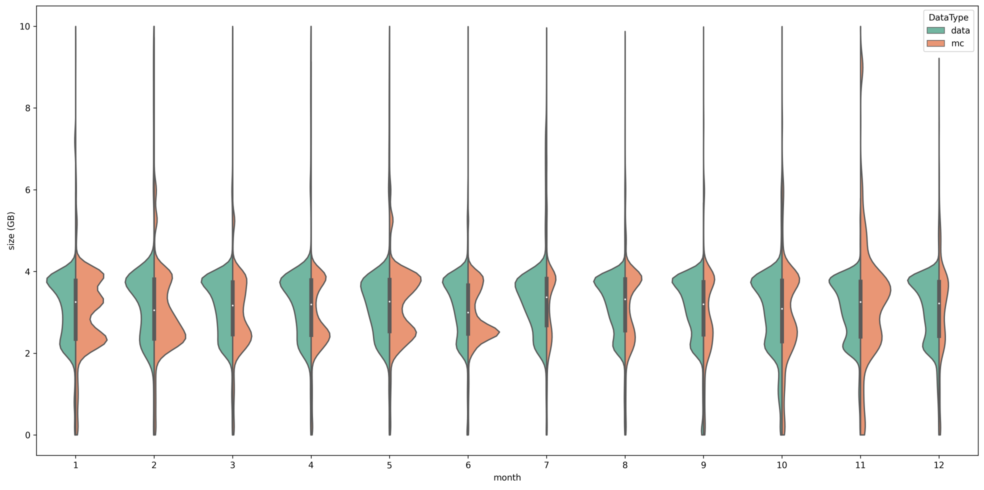


Figure 4.9: File size month distribution of Italian requests in 2018 showing the most common sizes for the file requested in each month of the year

From the analysis of the data types, the requests seem equally distributed between data types *mc* and *data*. The *mc* data are files concerning Monte Carlo simulations and *data* are real data from the experiment. The file types, which represent the typology of the file requested, are distributed based on the tasks, and it is not possible to infer any specific pattern on this distribution, but it has a confirmation of data type distribution if we compare the file type with and without *SIM*, that indicates if the file is a data type Monte Carlo or not.

This preliminary analysis of monitoring data provides useful information about the dimension of the problem. The number of requested files is very high with respect to the number of users and sites where the jobs are run. Nevertheless, the number of requests per file is not too high and the average value is small even if we consider only the files requested more than once.

Moreover, this behavior is confirmed in the other regions and, as a consequence, it is expected to have comparable results regardless of the region. The main difference is with the USA region that has a higher order of magnitude in

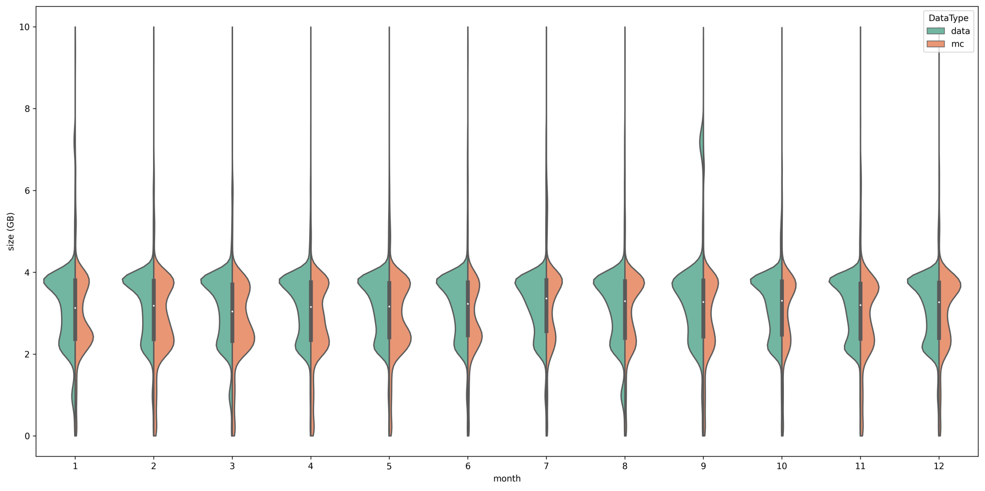


Figure 4.10: File size month distribution of United States requests in 2018 showing the most common sizes for the file requested in each month of the year

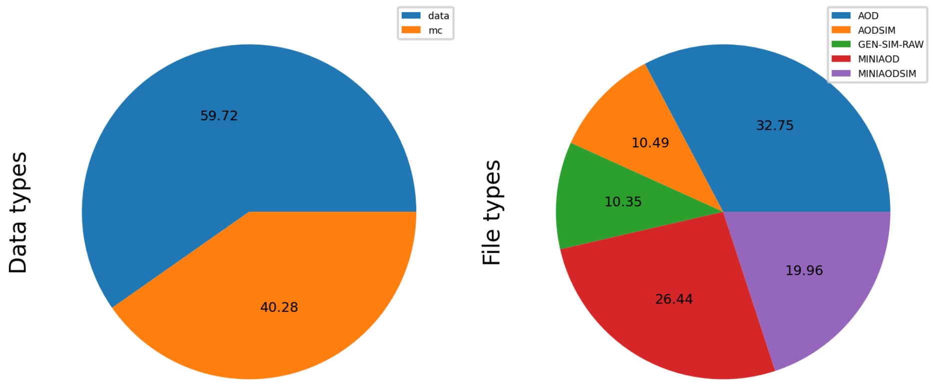


Figure 4.11: Data type composition of Italian requests in 2018 showing insight of domain-specific information about the file type

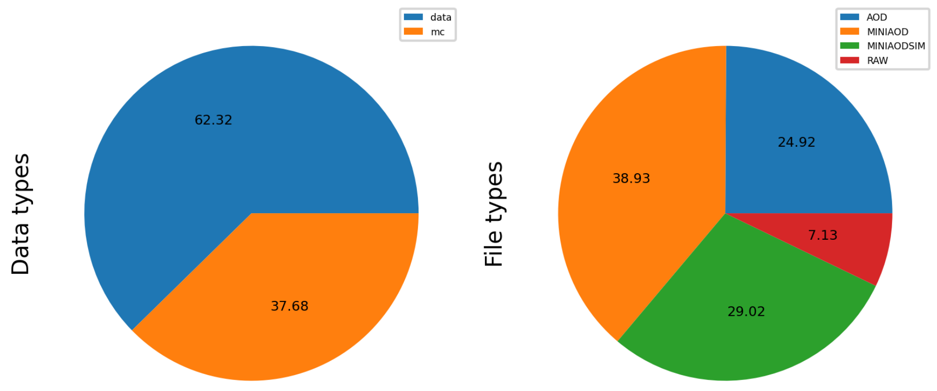


Figure 4.12: Data type composition of United States requests in 2018 showing insight of domain-specific information about the file type

the number of requests although, the average number of requests is comparable and also the relationship between the total number of requests and the number of sites, users, and tasks are the same.

The whole analysis confirms that tackling this problem falls in the field of Big Data due to the file sizes and the number of requests.

4.3 Simulation environment

Enhancing an object in a specific context is not trivial and the Data Lake scenario is complex and has many actors as shown in Section 4.2. Thus, we want to take control of the different techniques used during the experimentation for a better understanding of the consequences and to have a reproducible toolchain for future development. For these reasons, a specific environment was built, as close as possible to reality, according to the WLCG specifications and behaviors (Section 4.2).

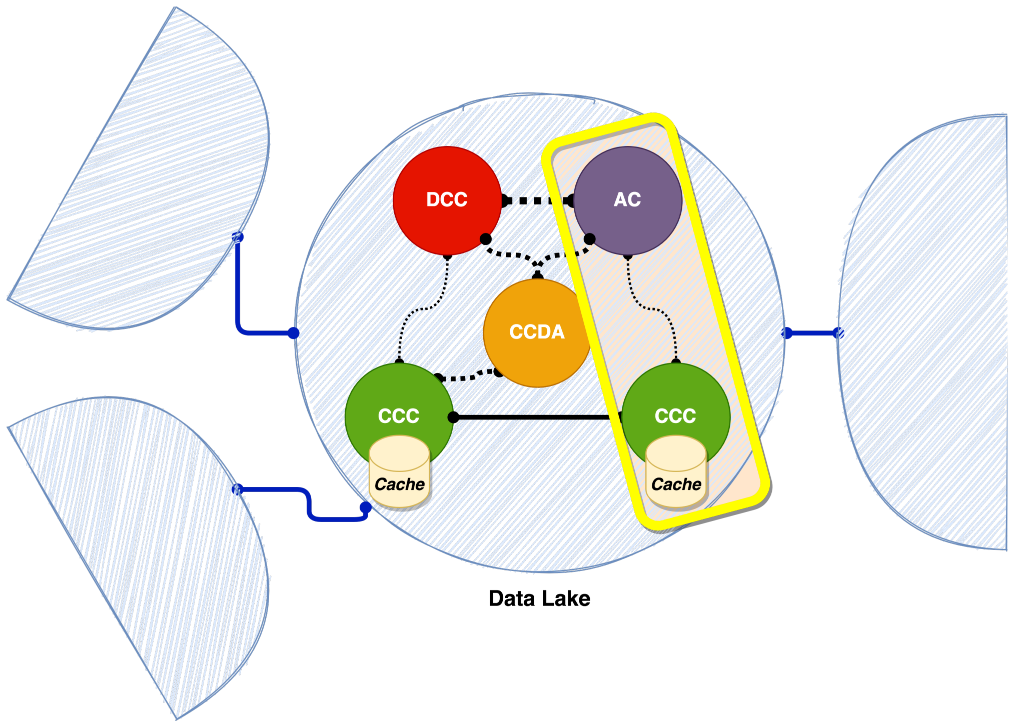


Figure 4.13: Illustration of environment implementation with the focus on the components selected to solve the problem

The environment model uses fewer components compared to the full organization of a Data Lake at WLCG to put in evidence the reaction of the cache system. As it is depicted in Figure 4.13, the project focuses on the interaction between the clients and a Data Lake composed of a computing center with a cache layer and the main storage system. Thus, the principal component is the cache itself, compliant with the real object used in the WLCG Data Lake model.

Moreover, this model can be used for specific datasets of the real-world use case (described in Section 4.2.1) or also synthetic datasets.

The simulation environment allows experimenting with the cache in the Data Lake scenario analyzing the results, with the information contained in the logs. Each approach used to modify the cache behavior can be stressed using several configurable parameters and features used as input to the cache system. The tool created in this project is convenient to use for experimentation before deployment in the real world, where technologies are dependent on the type of data flow and the field of application. However, it is possible to use the same tool to provide a service to external sources.

In fact, this project aims to be as generic as possible to be adapted in any context of caching management, without relating to any framework. Of course, it is still possible to use domain-specific information to enrich the explored method. Hence, the proposed environment aims to experiment with an Artificial Intelligence that auto adapts itself to new situations in the context of Data Lakes (Section 4.2), despite the domain of the data or the current topology of the network.

As mentioned before, the model has three basic components: i) a main storage system (where the files reside); ii) a cache that serves the requests; iii) and a client that asks for data. The simulation consists to resolve all the client's requests and serve the files from the cache.

As a result, the simplified model allows testing different policy mechanisms to control the request flow, deciding when the cache has to behave like a proxy or to not store a file in memory. As a rule, the main actions available in the simulation environment are to store and not store a file in the memory. When the cache decides to not store a file it is served in proxy mode, which means that it will fall back on the network.

The Figure 4.14 shows a schema of the environment and the main statistics collected to evaluate the behaviors. These data allow us to define several metrics explained in further chapters.

Accordingly, to the previous description of the environment and the schema in Figure 4.14, there is an amount of data read from the storage that splits into two parts:

i) read on miss: data served in the proxy mode because files are not stored

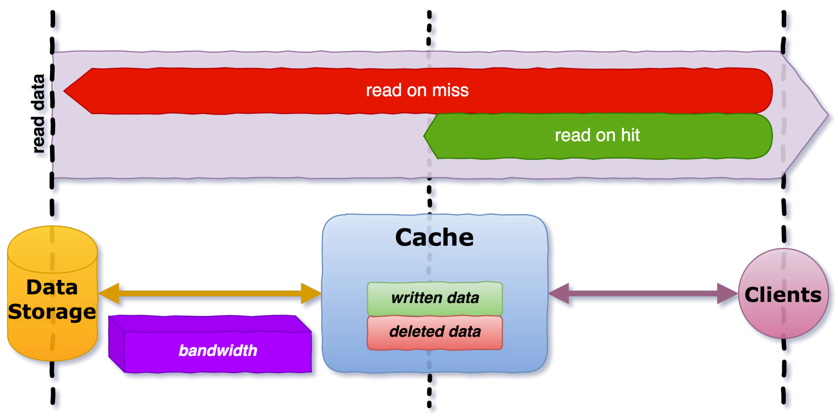


Figure 4.14: Simulation environment schema showing the several aspects taken into account and the units measured

in the cache memory;

ii) read on hit: data served directly from the cache memory.

An ideal cache aims to hold the read on hit higher than the miss one to unload as much as possible the main storage server. However, this is not always possible as anticipated with the analysis in Section 4.2.1 and it very depends on the client requests. For this reason, introducing new metrics is necessary to evaluate and compare different algorithms in the Data Lake scenario.

In addition, since the simulator is used to stress the cache decisions, it uses the simple topology of schema in Figure 4.14 to calculate the hypothetical daily bandwidth without any mechanism of the file transfer. This limit is used as a threshold for the daily requests and, if it is exceeded, the request is not managed by the current cache. This mechanism is used in the real world to *redirect* requests to other caches when the selected one is overloaded. Another approach that could be used is to queue the unresolved requests, but it is not useful in the current simulation due to the caching layer described in Section 4.2, where it is expected to have a topology of caches organized to collaborate.

The bandwidth between the cache and the main storage is an important aspect to monitor, as well as the two main quantity derived from cache operations:

i) written data

ii) deleted data.

As a consequence, the previously described watermarks impact all those quantities. The watermark thresholds are a reference for the cache to unload a bulk of files when a certain condition is satisfied. In this mechanism, there are two watermarks, an *high watermark* and a *low watermark*. When the occupied capacity of the cache reaches the *high watermark*, the cache starts to delete files until the *low watermark* is reached. These two thresholds are parameters that can be easily configured in the simulation and also disabled.

In conclusion, there are several parameters to keep under control for the cache content management improvement. It is not trivial to translate the gain taken with an algorithm or an approach with respect to the user experience that is strictly related to data access. Of course, the better the cached content is managed and the greater will be the end-user impact. However, this would be only a side effect because the main goal of the project is to automate and facilitate the management of the cache layer for system maintainers.

The Go programming language

The core of the project is the simulation environment and its modules that allows deploying the different proposed approaches. Because of the great importance of this component, the choice of the programming language cannot be done lightly. In fact, despite the great number of languages available, the final decision was to use the Go programming language [48, 49]. Go allows us to create a portable environment without losing in performances and, in addition, it is a very popular choice to build cloud services and tools. Also, the language allows deploying static executables in a variety of architectures and this is a very useful feature in a cloud environment that includes devices from the IoT to HPC (High-Performance Computing) servers.

Citing The Go Programming Language book [49]:

> Go is sometimes described as a "C-like language," or as "C for the 21st century."

Go was influenced by several languages, such as *C*, *Alef* and others. In particular, from *C* has kept the way of compiling to efficient machine code and cooperating naturally with the abstractions of current operating systems. Also, it has elements of communicating sequential processes (CSP) and concurrency programming, and it also has a garbage collector.

Moreover, the Go language includes a whole tool chain, thus with the Go project, they are available the tools and the standard libraries. The main strength of this language is its simplicity [49]:

> Simplicity requires more work at the beginning of a project to reduce an idea to its essence and more discipline over the lifetime of a project to distinguish good changes from bad or pernicious ones. [...] convenience. Only through simplicity of design can a system remain stable, secure, and coherent as it grows.

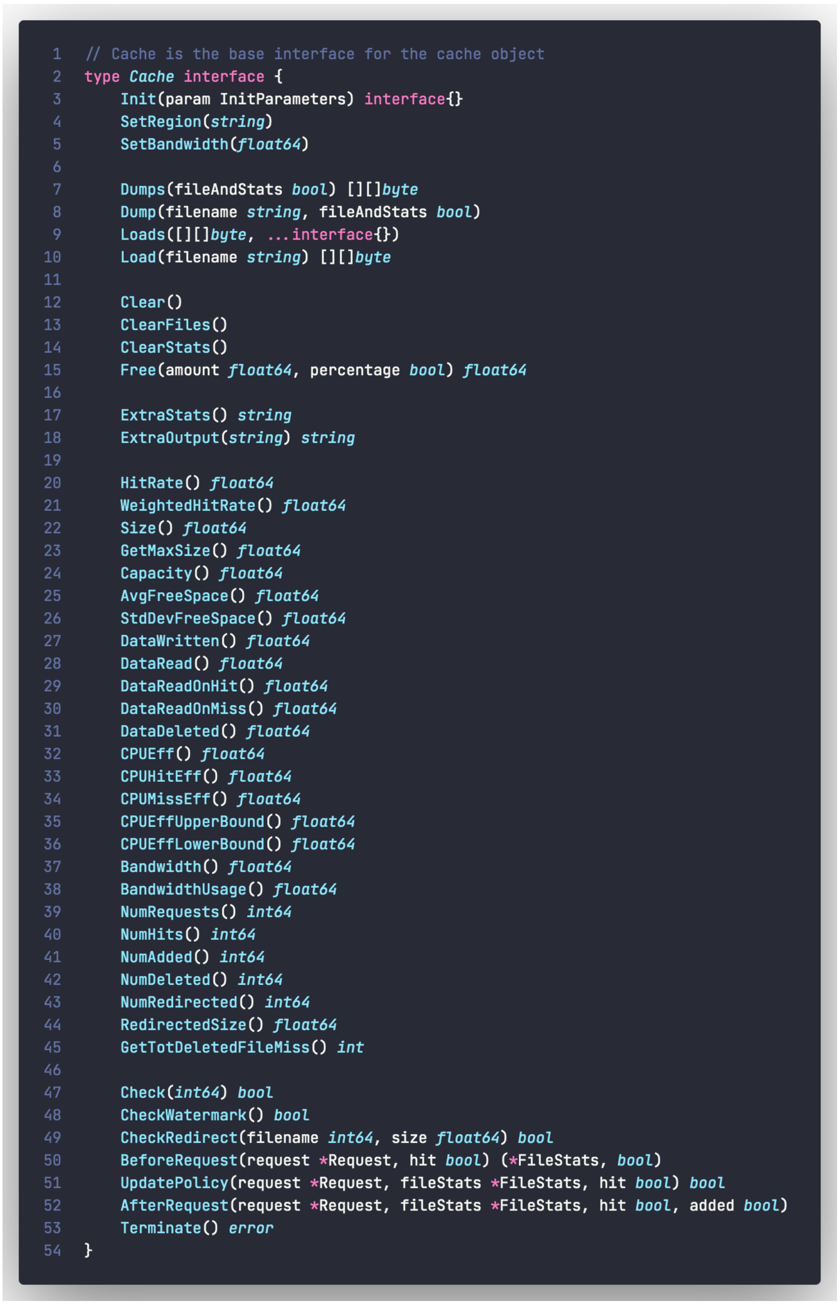
Finally, simplicity does not mean that it is not performant [49]:

> in practice Go gives programmers much of the safety and run-time performance benefits of a relatively strong type system without the burden of a complex one. (...)

In conclusion, Go is a perfect match for a cloud environment. The portability, the performances, and the ability to stay close to the system, without losing modern programming patterns such as the concurrency and async tasks, make Go a good programming language to solve the problems addressed.

Cache extensions

The cache object in the simulator is a struct that matches the interfaces in Figure 4.15. Creating a struct that satisfies those requirements implies the possibility to use the custom object as a cache inside the simulator. Several methods of the struct are necessary to produce the statistics and the information needed to understand the behavior of the cache. Thus, it is advisable to extend the basic cache already available in the simulator, such as LRU. Also, the already implemented approaches could be used as references for new custom methods.



```go
// Cache is the base interface for the cache object
type Cache interface {
    Init(param InitParameters) interface{}
    SetRegion(string)
    SetBandwidth(float64)

    Dumps(fileAndStats bool) [][]byte
    Dump(filename string, fileAndStats bool)
    Loads([][]byte, ...interface{})
    Load(filename string) [][]byte

    Clear()
    ClearFiles()
    ClearStats()
    Free(amount float64, percentage bool) float64

    ExtraStats() string
    ExtraOutput(string) string

    HitRate() float64
    WeightedHitRate() float64
    Size() float64
    GetMaxSize() float64
    Capacity() float64
    AvgFreeSpace() float64
    StdDevFreeSpace() float64
    DataWritten() float64
    DataRead() float64
    DataReadOnHit() float64
    DataReadOnMiss() float64
    DataDeleted() float64
    CPUEff() float64
    CPUHitEff() float64
    CPUMissEff() float64
    CPUEffUpperBound() float64
    CPUEffLowerBound() float64
    Bandwidth() float64
    BandwidthUsage() float64
    NumRequests() int64
    NumHits() int64
    NumAdded() int64
    NumDeleted() int64
    NumRedirected() int64
    RedirectedSize() float64
    GetTotDeletedFileMiss() int

    Check(int64) bool
    CheckWatermark() bool
    CheckRedirect(filename int64, size float64) bool
    BeforeRequest(request *Request, hit bool) (*FileStats, bool)
    UpdatePolicy(request *Request, fileStats *FileStats, hit bool) bool
    AfterRequest(request *Request, fileStats *FileStats, hit bool, added bool)
    Terminate() error
}
```

Figure 4.15: Golang cache interface used in the simulator

Simulation parameters

The simulator has been developed to be highly customizable in each of its aspects. In Table 4.1 are described in detail the main customization parameters available for the simulation. Other parameters depend on the algorithm or technique used.

Parameter	**Info**
data source	specifies the source of the simulation, that can be a file or a folder that contains several files
type of the simulation	indicates if the simulator has to follow a different flow, for example loading a previous cache instance or not
window	length of the simulation. It is possible to indicate the $start$, $stop$ and the $size$ of the window. The unit is the day
region	filter data by a specific region
cache type	the algorithm or the approach to use
cache dimension	the size of the cache memory in two components: the $value$ and the $unit$. E.g. for a cache of $100Terabytes$ is $value = 100$ and $unit = T$
bandwidth	the available bandwidth with $value$ entry in *Gbits*. Also, the *redirect* mechanism can be activated
watermarks	controls if the cache has to use the watermarks or not and also both, the *high watermark* and the *low watermark*

Table 4.1: Simulation parameters

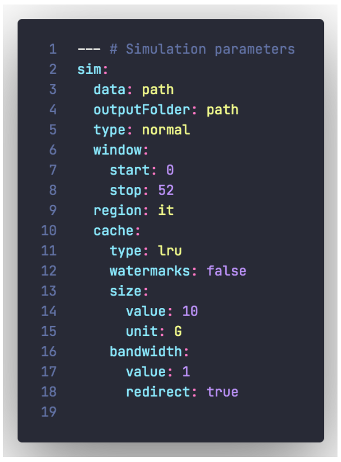
As can be reported from Table 4.1, the *cache type* parameter is used to select a precise algorithm or policy for caching content management. The cache simulator supports several types of replacement policies out of the box that is used for comparison:

- Least Recently Used (LRU): the golden standard policy used in caching systems;

- Least-Frequently Used (LFU): another widely-adopted policy that does not take into account the time of the request but counts how often an item is needed;
- Size Big and Size Small: policies that remove files starting respectively from the biggest and the smallest.

In addition, it is possible to extend source code to implement new policies and select them with the same configuration file (Section 4.3).

The desired configuration can be described as a simple YAML file [50]. YAML stands for "YAML Ain't Markup Language" and it is a human-readable data serialization language. In the Figure 4.16 there is an example of simulation environment configuration:



```yaml
--- # Simulation parameters
sim:
  data: path
  outputFolder: path
  type: normal
  window:
    start: 0
    stop: 52
  region: it
  cache:
    type: lru
    watermarks: false
    size:
      value: 10
      unit: G
    bandwidth:
      value: 1
      redirect: true

```

Figure 4.16: Simulation environment configuration file example in YAML

Simulation analyzer

In addition to the previously described simulator, an analyzer was built to easily navigate the results. The simulation produces several comma-separated values following the CSV standard [51]. The results are often compressed for convenience and the analyzer is written in Python[52] using the framework Plotly [53]. The Python environment is commonly used for Data Analysis purposes and in fact, has a lot of very useful frameworks and libraries.

The analyzer presents the results of the simulation in a practical dashboard that can be accessed through the browser. The dashboard is an independent service, and it is cloud-ready, thus it can be deployed in a container. An example is shown in Figure 4.17. With such a dashboard it is possible to filter, select, and compare the results of the different approaches used.

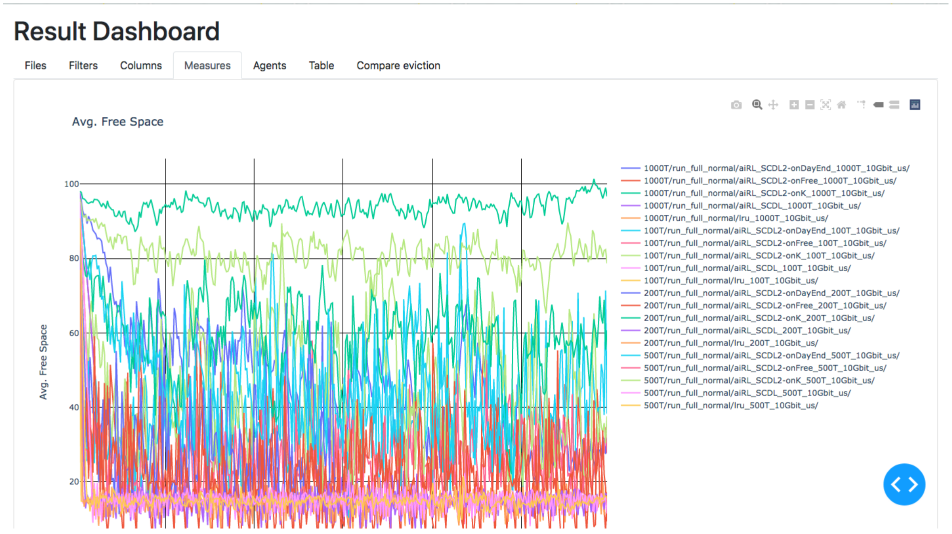

Result Dashboard
Files
Filters
Columns
Measures
Agents
Table
Compare eviction
Avg. Free Space
Avg. Free Space
100
80
60
40
20
1000T/run_full_normal/aiRL_SCDL2-onDayEnd_1000T_10Gbit_us/
1000T/run_full_normal/aiRL_SCDL2-onFree_1000T_10Gbit_us/
1000T/run_full_normal/aiRL_SCDL2-onK_1000T_10Gbit_us/
1000T/run_full_normal/aiRL_SCDL_1000T_10Gbit_us/
1000T/run_full_normal/lru_1000T_10Gbit_us/
100T/run_full_normal/aiRL_SCDL2-onDayEnd_100T_10Gbit_us/
100T/run_full_normal/aiRL_SCDL2-onFree_100T_10Gbit_us/
100T/run_full_normal/aiRL_SCDL2-onK_100T_10Gbit_us/
100T/run_full_normal/aiRL_SCDL_100T_10Gbit_us/
100T/run_full_normal/lru_100T_10Gbit_us/
200T/run_full_normal/aiRL_SCDL2-onDayEnd_200T_10Gbit_us/
200T/run_full_normal/aiRL_SCDL2-onFree_200T_10Gbit_us/
200T/run_full_normal/aiRL_SCDL2-onK_200T_10Gbit_us/
200T/run_full_normal/aiRL_SCDL_200T_10Gbit_us/
200T/run_full_normal/lru_200T_10Gbit_us/
500T/run_full_normal/aiRL_SCDL2-onDayEnd_500T_10Gbit_us/
500T/run_full_normal/aiRL_SCDL2-onFree_500T_10Gbit_us/
500T/run_full_normal/aiRL_SCDL2-onK_500T_10Gbit_us/
500T/run_full_normal/aiRL_SCDL_500T_10Gbit_us/
500T/run_full_normal/lru_500T_10Gbit_us/

Figure 4.17: Result dashboard example screenshot

Smart Caching for Data Lakes

Contents

5.1 Exploring file scoring 60
5.2 An agent for a better addition 62
5.3 Smart caching with double agents 66

This chapter introduces the various approaches that have been exploited in the project using the simulation environment described in Section 4.3. It starts exposing a method to measure the quality of a file from its statistics. This quality is then used to better evaluate the possibility of storing valuable information into the cache memory and to discard not useful files.

Then, using the notions presented in Section 2.2, it depicts the Artificial Intelligent approach: Smart Caching for Data Lakes (SCDL). SCDL is described in its two versions: the first one, which focuses only on filtering the client requests to store the most valuable data, and the second one, an upgrade of the previous version where the agent decisions affect also the eviction of the stored files.

The various algorithms will be described in detail to explain the methods and the ideas used to tackle the problem of cache content management.

5.1 Exploring file scoring

The first study was focused on how to guess the goodness of a file to increment the content of the cache memory [54]. The target of this approach is exploring a method to evaluate the file before making the decision to store it. Such a method uses the file features and collects statistics to improve cache behavior.

This technique named *weight function* should aid the cache to accept only files that matter. As shown in Figure 5.1, the basic idea is to evaluate a file request assigning a precise score and choose if it is worth to be stored.

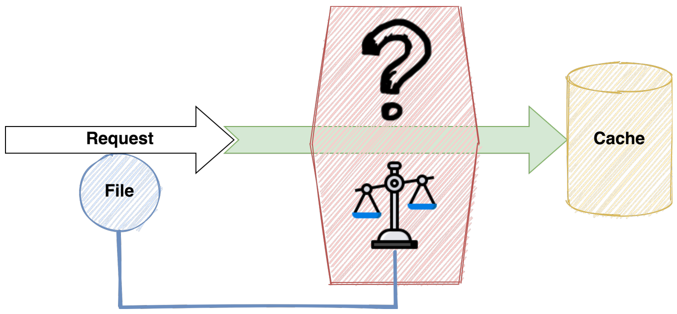


Figure 5.1: Schema of weight function idea to select the file to store

This approach is based on the concept of file weight (a score) which is used to determine if the cache has to accept or not the file. The policy is: a file with a smaller weight is more likely to be inserted into the cache. To accomplish this task, several statistics about the file are collected to compute the score, and the average weight value of all files is used as a threshold for cache addition. Of course, the threshold could be changed but verifying the best threshold is out of the scope of this thesis. Besides, the intent of this static technique was principally to have a better insight from the file to use in a more dynamic and auto-adaptive approach such as Reinforcement Learning.

The algorithm

Given a requested file f at time t, it considers the following features:

- $numRequests(f, t)$: the number of requests (frequency) to f until the current time t
- $size(f)$: the file size
- $requestDelta(f, t)$: the average time difference between the previous file requests and the last one (a relative recency)

The average time difference is calculated on the last $k = 32$ requests (or on all the last requests, if the file has been requested $k < 32$ times), according to the following formula:

$$requestDelta(f, t) \quad = \quad \frac{\sum_{i=1}^{k-1} time(f, k) - time(f, i)}{k} \tag{5.1}$$

where $time(f, i)$ is the time of the i-th request to f ($time(f, k)$ is the time of the last request).

The statistics to compute the score (or weight) of a file are kept in a time window of 14 days. The threshold to accept a file is calculated on the median value of all the collected file weights until time t.

With this approach were investigated 3 functional families of the weight function which aggregates in different ways the following features: *numRequests*, *size* and *requestDelta*.

Each of the 3 families have α, β, γ as parameters

- Additive family

$$\begin{aligned} weight_A(f, t) \quad = \quad & \alpha \cdot numRequests(f, t) + \beta \cdot size(f) + \\ & \gamma \cdot requestDelta(f, t) \end{aligned} \tag{5.2}$$

- Additive-Exponential family

$$\begin{aligned} weight_E(f, t) \quad = \quad & numRequests(f, t)^{\alpha} + size(f)^{\beta} + \\ & requestDelta(f, t)^{\gamma} \end{aligned} \tag{5.3}$$

- Multiplicative family

$$\begin{aligned} weight_M(f,t) &= numRequests(f,t)^{\alpha} \cdot size(f)^{\beta} \cdot \\ & \quad requestDelta(f,t)^{\gamma} \end{aligned} \tag{5.4}$$

In the experimentation phase, several combinations of function parameters α, β, γ will be tested to evaluate the cache reaction for each function. The expected result should be a combination of the parameters that enhance the importance of the file feature to improve the cached content during the time.

5.2 An agent for a better addition

As introduced in Chapter 4, the objective is to apply the method to a computing environment where data can vary a lot by type and size. Moreover, the client requests may vary depending on the trend of the analysis or on a special event incoming. Thus, searching for a score using statistics to make a better content of cache memory, as explained in Section 5.1, could be not enough for overall optimization of a cache layer, including the user experience, because it is not a self-adapting solution to the cache problem but only a better request filtering for a hotter cache. In fact, a cache system should do a more fine job adapting itself to the requests and choosing autonomously what to accept or not, relying only on the client's necessities. For this reason, following the same idea of the *Weight Function*, the Smart Caching approach is conceived as a smarter solution to filter the file accepted by the cache. Due to the autonomous requirement, the choice to use a Reinforcement Learning technique was preferred to the creation of a model or to other Machine Learning techniques.

Workflow

The proposed approach is based on Reinforcement Learning (RL), a Machine Learning paradigm that learns how to dynamically operate in any environment. The main technique used is the associative search task, known also as *contextual bandits* in literature [30].

It uses the trial-and-error technique to search for the best actions in a given situation. As a control algorithm, it uses the off-policy Q-Learning to update the action values. Furthermore, delayed reward mechanisms are integrated to improve the action decisions due to the cache environment changes. The environment is described in Figure 5.2 and, in detail, it is composed of a cache with a specific size of S and a network bandwidth B with the main storage.

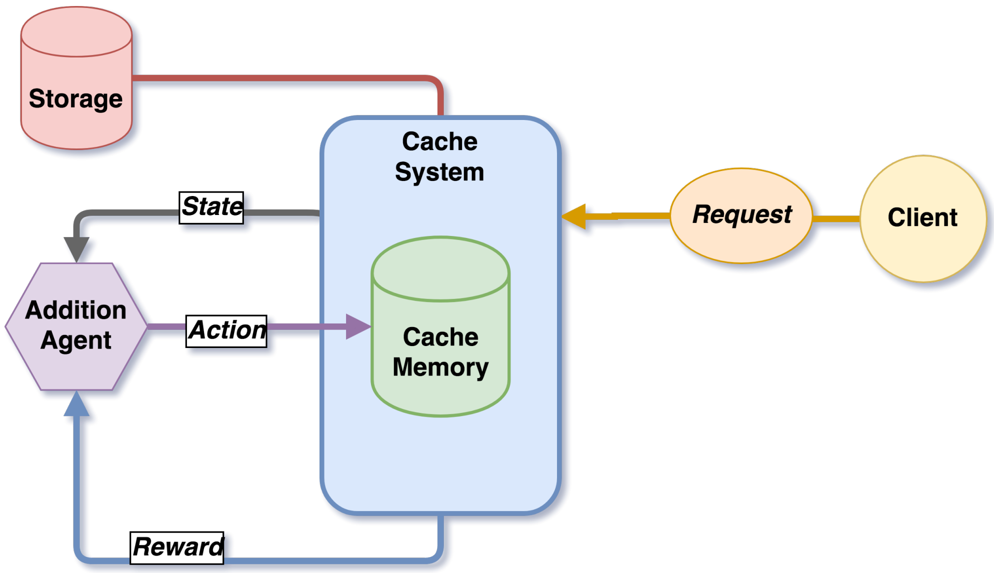


Figure 5.2: Schema of the environment of single-agent approach that uses the information of client request to accept or not a file into the cache

State Space

The state s is composed using the basic information (features) taken from the file f statistics collected during the environment lifetime. Thus, to decide whether a file f should be stored in the cache it selects the following features of f: i) the size (s_f), ii) the number of requests (n_f), and iii) the delta time (t_f), i.e. the elapsed time between the current time and the time of the last request.

The statistic history traces 7 days of the file's requests, and it is deleted if the file is no longer requested and it is not present in the cache memory. The environment uses a discrete internal time to calculate the recency of a file.

Moreover, the state is enriched with the cache occupancy percentage oc and cache hit rate hr.

Since the number of states must be finite, the file features are discretized in a finite number of classes (using a simple binning technique with ranges) and hence the state S is represented as a quintuple $(\sigma, \phi, \delta, o, \eta)$, where σ, ϕ, δ are the labels of the class to which the size, the frequency, the delta time of f belongs, respectively; o, η are the labels of the class to which the occupancy percentage and the hit rate belongs.

Hence, the states for the addition agent is defined as:

$$S = (\sigma, \phi, \delta, o, \eta) = b(s_f, n_f, \delta_{t_f}, oc, hr) \tag{5.5}$$

where b is the function mapping s_f, n_fm δ_{t_f}, oc and hr.

The system also takes into account the two cache watermarks, a higher W_{high} watermark and a lower W_{low} watermark. When the size of the files stored in the cache reaches W_{high}, the least recently used files are removed until W_{low} is reached. This mechanism prevents the cache memory from becoming too full. The parameters to W_{high} and W_{low} are set according to the amount of available space. Also, the underlying cache uses the Least Recently Used (LRU) criterion to decide which files have to be removed.

Action Space

Each time the agent requests a file f, the state s is computed in terms of the statistics of f and the cache. For each possible state, there are two actions: *Store* and *NotStore*. It considers as next state s' (Equation (5.6)) the same input state s (Contextual Bandit). As a consequence, the agent has to learn which action is the best for the current state through the delayed rewards used as stimuli of past decision traces with the intent to allow only the valuable files to be stored. To assign the delayed rewards it memorizes the last action a taken for the input state s and it rewards or penalizes that action based on cache constraints when the same state s occurs in the next request.

The agent updates the action values of the state using the Q-Learning method, where the quality Q (goodness) of the state is a combination of the state and the action value:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (r + \gamma \cdot Q(s', best) - Q(s,a)) \tag{5.6}$$

where s is the state, s' is the next state (equal to s for this approach) and a the action. The α parameter is the learning rate and γ is the discount factor. In the experiments those variables are fixed: $\alpha = 0.9$ and $\gamma = 0.5$.

Rewards

The reward r is computed as follows: the cache environment will assign a negative or positive reward on the basis of several constraints that involve cache statistics.

In this method, those actions that store files that do not increment the value of Read on Hit are penalized, in particular, the desired target is that the amount of hits is higher than the miss files.

The positive reward r is the size of f in $Gigabytes$, while the negative reward is $-f$. These rewards are assigned to the last actions decided for the state where the actions were taken. For each file are stored a maximum of 32 decisions.

The Algorithm

In the Algorithm 2 a simplified description of the SCDL process is described. The approach uses the $\epsilon - greedy$ technique to balance the exploration and exploitation of actions. The ϵ parameter is always initialized to 1.0 and it is decreased with an exponential decay rate set from the environment configuration with a lower limit of 0.1.

To summarize the algorithm, when a file is requested, its past action history is used to apply the delayed rewards, as explained before. Also, if the requested file is not in the cache, the agent choice is stored in the memory.

Algorithm 2: Algorithm SCDL

```
Data: file requests
Initialize cache
Initialize historical statistic table stats ; // to collect file statistics
Initialize Q-Table additionTable ; // all action values to 0
Initialize map doneActions ; // history of taken actions
for each requested file req do
    $stats.update(req.filename)$
    $hit \leftarrow cache.Check(req.filename)$
    $currentState \leftarrow cache.State(req)$
    if $map.check(req.filename) = true$ then
        for $pastState, pastAction \leftarrow map.get(req.filename)$ do
            $reward \leftarrow makeReward(pastState, pastAction, req, cache.Statistics)$
            $additionTable.Update(pastState, passtAction, reward)$
    if $hit = false$ then
        if $randomNumber() < \epsilon$ then
            $currentAction \leftarrow additionTable.peekRandomAction(currentState)$
        else
            $currentAction \leftarrow additionTable.peekBestAction(currentState)$
        $map.insert(req.filename, curState, curAction)$
        if $currentAction = Store$ then
            $cache.AddFile(req.filename, req.size)$
    else
        $cache.Serve(req.filename)$
```

5.3 Smart caching with double agents

The proposed approach, named SCDL2 (*Smart Cache for Data Lakes*), is based on a previous Smart Caching system for Data Lakes (SCDL) introduced in the previous Section 5.2. As a consequence, it is pursued in the RL way but with agents that have more control over the cache memory.

In this upgraded version, SCDL2 uses two different agents to solve the caching problem: one decides whether a requested file has been stored (addition agent) and the other chooses how to free the cache memory (eviction agent) deleting a specific file category. Thus, an agent bases its decision on the state of the request and the other on the state of the cache memory.

The RL technique employed is the Q-Learning method, which uses the information taken from the file statistics and from the cache status. The main difference between SCDL2 and SCDL is that the latter has only the addition agent, it uses the LRU as a policy for freeing the memory and the associative search for the best action of a given state. Also, SCDL2 is a full RL and thus it takes into account the state transitions.

Consequently, for the addition agent, there is a full RL with the next state given from the request choice history. Furthermore, for the eviction part, there is no link with the watermark mechanism but instead, there are several ways with which the eviction agent is triggered.

The sequential decision process with which the agents learn the best behavior through interaction with the environment is described in Figure 5.3.

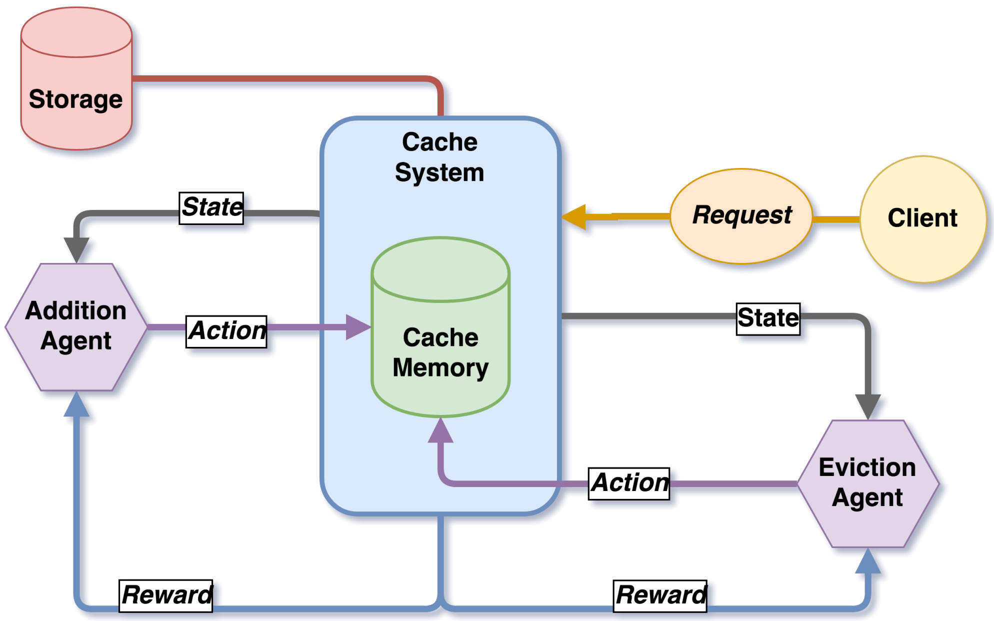


Figure 5.3: Reinforcement Learning schema of the double agent approach, where the AI choose both, the addition and eviction of a file into the cache memory

Workflow

As shown in Figure 5.3, the goal is to modify how the cache accepts the files into the memory and how the files are evicted when new space is necessary. In particular, only the valuable files should be stored in the memory. The environment is defined by the cache memory with a specific size S and bandwidth B. In such an environment, both the agents use the trial-and-error technique to search for the best actions in a given situation, using the Q-Learning to update the action values. The delayed reward mechanisms remain integrated to improve the action decisions due to the cache environment changes. Moreover, the environment uses a discrete internal time. This time is incremented at each request, and it is used to calculate the recency of a file.

When a file f is requested:

- the *Addition Agent* is called in order to decide whether storing or not the file f;
- the *Eviction Agent* is called if a particular event is satisfied and chooses which files to remove. The possible events are:
    - it is necessary to free Space
    - it is the end of the day
    - k iterations have passed

After a request is managed, both agents receive delayed rewards if necessary (depending on file choice history).

State Space

The two agents work with different state spaces. The addition agent has the same state space as SCDL agent Equation (5.5).

Instead, to decide which file to delete, the eviction agent uses statistics about the current state of the cache. In particular, the files stored in the cache memory are divided into categories with which the cache chooses the files to delete. Similarly to the addition agent file features, the eviction takes the following information from the stored files: i) the size (s_f), ii) the number of requests (n_f), and iii) the delta time (t_f), i.e. the elapsed time between the current time and the time of the last request.. These are used to associate the file to a specific category c with: i) all file of size (s_c), ii) a number of requests (n_c), and iii) a delta time (t_c). For each category it is calculated the amount of space occupied by the category itself (oc_c).

The statistic history traces 7 days as for the previous approach and the environment uses a discrete internal time to calculate the recency of a file.

Because the number of states must be finite in the Q-Learning, these features are discretized in a finite number of buckets.

Hence, the states for the addition agent are defined as in Equation (5.5):

$$S_a = (\sigma, \phi, \delta, o, \eta) = b_a(s_f, n_f, \delta_{t_f}, oc, hr) \tag{5.7}$$

where b_a is the function mapping s_f, n_fm δ_{t_f}, oc (current cache occupancy) and hr (current cache hit rate).

Instead, the eviction agent has the following state:

$$S_e = (\sigma_c, \phi_c, \delta_c, o_c, o, \eta) = b_e(s_c, n_c, \delta_{t_c}, oc_c, oc, hr) \tag{5.8}$$

where b_e is the function mapping s_c, n_c, δ_{t_c}, oc_c, oc and hr to the corresponding buckets.

Action Space

The results of agents' decisions are stored into two different Q-tables, where there are evaluations of actions for each possible state: *additionTable* and *evictionTable*.

The action space for the *addition agent* is composed of two actions: *Store* and *NotStore*.

Instead, the action space for the *eviction agent* contains four actions representing the four available eviction methods: *NotDelete*, *DeleteAll*, *DeleteHalf*, *DeleteQuarter* and *DeleteOne*. Those methods identify how a selected category has to be managed. The choice to consider a finite number of actions for a specific category instead of having a different delete action for each file stored in the cache (i.e. all the files that are possible to delete in a time) allows us to limit the agent search space.

The agents update the action values of the state using the Q-Learning method, where the quality Q (goodness) of the state is computed as a combination of the state and the action value:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (r + \gamma \cdot Q(s, best) - Q(s,a)) \tag{5.9}$$

where s is the state and a the action. The α parameter is the learning rate and γ is the discount factor. As in Section 5.2, those variable are fixed during the experiments: $\alpha = 0.9$ and $\gamma = 0.5$.

Rewards

Since the decision of storing a file f affects the cache composition and its actual contribution cannot be determined immediately, the approach uses delayed rewards.

Therefore, after each file request, it stores the action that the agent chose for the given state (situation). Then, later in time, it will evaluate that decision with a positive or a negative reward depending on specific rules.

In short, it chooses to penalize those actions which do the cache more work more, such as write new files and delete files to free space, trying to avoid unuseful operations, and limit the cache actions. The reward r has a unitary value that increments with several constraints. For the addition agents it is checked if the chain of choices made for the file f complies with the following rules:

- if current file hit: for each previous choice, $r = 1$. The action takes and extra $+1$ if the situation passed from a miss to an hit;
- if current file miss: for each previous choice, $r = -1$. An additive malus of -1 is given if the file passed from hit to miss in the last action in history.

For the eviction agent there is a similar reward policy but focused on the file category:

- if current file hit: for each previous choice of that category, $r = 1$. The action takes and extra $+1$ if the action is *NotDelete* and also if the capacity is not higher;
- if current file miss: for each previous choice of that category, $r = -1$. An additive malus of -1 is given if the action is *NotDelete* or if the capacity is increased.

As stated in Section 5.3, the categories depend on several file features, and because these can change during the file life into the cache, the category of a file change also with them during the time.

Last but not least, there is a special situation for the eviction agent where it is rewarded despite the file category. When the agent is forced to delete files because the cache has to insert a new one, but the operation does not finish successful, all the decisions of the eviction agent that are *NotStore* take a predefined malus $r = -P$ and all the actions that delete files receives $r = P$. Usually, $P = 1.0$ and is defined a priori.

The algorithm

In Algorithm 3 a high level description of the process flow is described, while in Algorithm 4 and Algorithm 5 there are the details of the two different agent steps.

Algorithm 3: Main flow of SCDL2

```
Data: file requests
Initialize an empty cache
Initialize Q-Table additionTable
Initialize Q-Table evictionTable
Initialize map $history_{addition}$
Initialize map $history_{eviction}$
for each requested file req do
    $stats.update(req.filename)$
    $hit \leftarrow cache.check(req.filename)$
    $additionPhase(hit, req)$
    if $trigger\ eviction\ agent$ then
        $evictionPhase()$
    $delayedRewardsAddition(req)$
    $delayedRewardsEviction(req)$
```

Both agents use the $\epsilon - greedy$ technique to make choices and as in the previous approach, the ϵ parameter is initialized to 1.0 and it is decreased with an exponential decay rate set from the environment configuration with a lower limit of 0.1.

The choices made are collected in a map for each agent and this information is used to provide late rewards. The delayed rewards function scans all the choices made for the file f requested and applies the rules explained in the previous section to reward the agents relying on the history.

Algorithm 4: Addition phase in SCDL2

```
additionPhase($hit, req$)
$state \leftarrow cache.state(req)$
if $hit = false$ then
    if $randomNumber() < \epsilon_{addition}$ then
        $currentAction \leftarrow additionTable.peekRandomAction(state)$
    else
        $currentAction \leftarrow additionTable.peekBestAction(state)$
    $history_{addition} \leftarrow (state, currentAction)$
    if $currentAction = Store$ then
        $cache.add(req)$
else
    $serve(req.filename)$
```

Algorithm 5: Eviction phase in SCDL2

```
evictionPhase()
$state \leftarrow cache.state(req)$
foreach file $f$ in cache do
    $updateFileCategory(f)$
foreach category $c$ do
    if $randomNumber() < \epsilon_{eviction}$ then
        $currentAction \leftarrow evictionTable.peekRandomAction(state)$
    else
        $currentAction \leftarrow evictionTable.peekBestAction(state)$
    $history_{eviction} \leftarrow (state, currentAction)$
    $apply(curAction)$
```

Experiments

Contents

6.1 Evaluation metrics **75**

6.2 Tests and results **81**

6.2.1 Selection by file score 83

6.2.2 Cache composition by SCDL 87

6.2.3 Cache management by SCDL2 90

6.2.4 Extreme use cases 93

6.3 Evaluation **95**

6.3.1 Metrics correlation 95

6.3.2 Notes on agent learning 95

6.3.3 Notes on selected features 97

6.3.4 Result considerations 98

This chapter covers the experiments done using the approaches described in the previous sections. The experiments will be presented as a constructive flow that respects the logical sequence of the thesis investigation.

Before the tests, a presentation of the metrics and how the experiments were made are illustrated. The results are first presented and then discussed in the appropriate section, where used methods are also compared to explain the obtained results.

6.1 Evaluation metrics

One reason to define several metrics to measure the proposed environment of Section 4.3 is to have a better understanding of the consequences of the cache's choices. A general approach does not relate simply to a cache evaluation metric such as the hit rate, because the dimension of the problem and the involved actors are really different and the final target could be diverse: for example, we want to monitor the choices' impact on the network bandwidth because they can have a side effect on user experience in the real world. Therefore, referring to the environment described in Figure 4.14, the metrics in Figure 6.1 are proposed to monitor the effect of a cache policy and to compare different approaches.

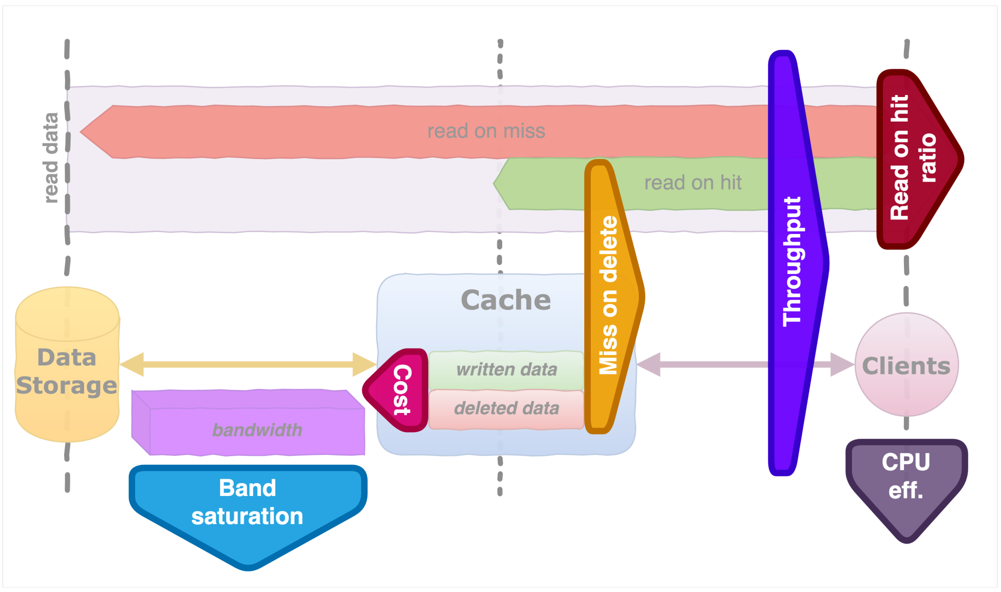


Figure 6.1: Projection of the metrics on the simulation environment that shows the several aspects controlled to study the cache behavior

The metrics have different targets and their purpose will be described in details:

- **Throughput**: to evaluate the quality of a cache in fulfilling the client requests;

- **Cost**: to evaluate the operational cost of the cache;
- **Band saturation**: to monitor the load on the network when serves remote contents;
- **Read on hit ratio**: to measure how much content is served from the cache memory;
- **Miss on deleted files**: to understand if the deleted files are important or not;
- **CPU efficiency**: to estimate the user experience effects.

Since there is no particular order with which the metrics are collected, it is necessary to specify that the first two, the **Throughput** and the **Cost**, are the most valuable for the purpose of this project, and they are used for the main comparisons. All the metrics' measures will be collected during the simulation days.

The first metric, the *Throughput*, is defined as follows:

$$Throughput = \frac{readOnHitData - writtenData}{C} \tag{6.1}$$

where C is the cache size and the *readOnHitData* is calculated as follows:

$$readOnHitData = \sum_{hit\, f} f_{size} \tag{6.2}$$

where f is a file requested from the cache memory (cache hit). With the same logic, the *writtenData* is the sum of the size of all files written in the cache memory after a request:

$$writtenData = \sum_{written\, f} f_{size} \tag{6.3}$$

then, the *readData* represents the whole volume of the file requested to the cache:

$$readData = \sum_{requested\, f} f_{size} \tag{6.4}$$

with the *Throughput* metric, it could be possible to evaluate the goodness of the memory content during the time. The borderline cases of this measurement are the following:

- the cache writes nothing: if the written data is 0 the read from the memory cache is impossible and then the cache is useless;
- the cache writes but there is no hit: this happens when requested files are always not present in memory.

It follows that the Throughput can not frame the whole cache reaction. Thus, to get a better picture of the effects of a novel method, it could be measured by the algorithm's effectiveness also with respect to the other metrics we mentioned before in this section. The next one, the measure *Cost*, aims at quantifying how much the cache is working in terms of pure cache operations, i.e. the size of the deleted or written files, with respect to the cache size C:

$$Cost = \frac{writtenData + deletedData}{C} \tag{6.5}$$

the *deletedData* is defined as the sum of the sizes deleted by the cache memory:

$$deletedData = \sum_{deleted\, f} f_{size} \tag{6.6}$$

With the *Cost* metric, it could be possible to have an idea of the pressure on the cache, similar to the load factor of a system. The borderline case when the cache writes nothing at all has to be avoided and usually, there should be a good balance between data addition and eviction: this is the reason why it is better to check also the written and deleted data separately. Similarly to the *Throughput*, the measure is normalized with respect to the size C to have a better comprehension of this metric when caches of different sizes are compared.

Then, it is obvious that a measure to quantify the usage of the network could be useful, especially the amount of data exchanged between the cache and the main storage. Therefore, the *BandSaturation* just measures this aspect and, of course, mainly depends on the files which were not in the cache at the moment they were requested (cache miss, Figure 4.14) and the available bandwidth of the cache B:

$$BandSaturation = \frac{readOnMissData}{B} \tag{6.7}$$

where the *readOnMissData* is the sum of the file sizes that have a miss in the cache memory:

$$readOnMissData = \sum_{missed\, f} f_{size} \tag{6.8}$$

with the same idea but with a different perspective, the *ReadOnHit ratio* aims to measure the amount of data served from the cache compared to the whole data requested by the clients:

$$ReadOnHit\ ratio = \frac{readOnHitData}{readData} \tag{6.9}$$

where *readOnHitData* is defined as before in Equation (6.2).

The *ReadOnHit ratio* value is strictly related to the cached content and manifests the goodness of the algorithm's previous choices.

The $avg\ \#miss$ metric computes the average number of misses after file evictions during a day. In particular, this metric is useful to monitor the errors made by the eviction agent because it considers how many times a deleted file has been requested in the future:

$$avg\ \#miss = \frac{\sum_{missed\ f} \#miss_f}{\#days} \tag{6.10}$$

Until now, the proposed metrics include only cache related information but, since the goal is also to improve the end-user experience, and it is possible to access historical information about data analysis workflows, a specific metric was included in the environment. Such a metric is the CPU efficiency of the analysis jobs, by considering the CPU time and Wall time of each request.

Regarding the CPU efficiency, it is important to highlight that the adopted strategy for the evaluation is based on a dedicated study which analysis first measured the loss of CPU efficiency while reading data from a remote storage element using monitoring data from the CERN MONIT project (Section 4.2). It resulted in costs on average about 15% of CPU time with respect to local data reading. Then, thanks to a testbed built over the Italian CMS Tier 2 topology proved that the CPU efficiency on a cache miss can be treated like remote access to that file, instead, a hit from the cache is equivalent to local file access. With this concept, it is possible to estimate in the various simulations the impact of CPU efficiency depending on the decisions made by the cache.

Hence it is possible to compute the average CPU efficiency as follows:

$$CPUEfficiency = \frac{\sum CPUTime}{\sum WallTime} \tag{6.11}$$

Moreover, we can have also an idea of how the proxied requests managed by the cache can affect this efficiency [55].

In detail, it is possible to estimate the decrease of performances of the proxied requests treating them as remote requests, which lose performance due to the delay in retrieving the file (Figure 6.2). As a result, we have two cases to calculate CPU efficiency:

- file served from cache memory (hit): same CPU efficiency of local files in the historical data;
- file served in proxy mode (mis): CPU efficiency decreased by δ that is defining as the following difference:

$$\delta = CPUEfficiency_{localfiles} - CPUEfficiency_{remotefiles} \tag{6.12}$$

In conclusion, there are several metrics to monitor and evaluate the cache system. Analyzing the changes of each metric we can deduce the goodness of behavior compared to another one.

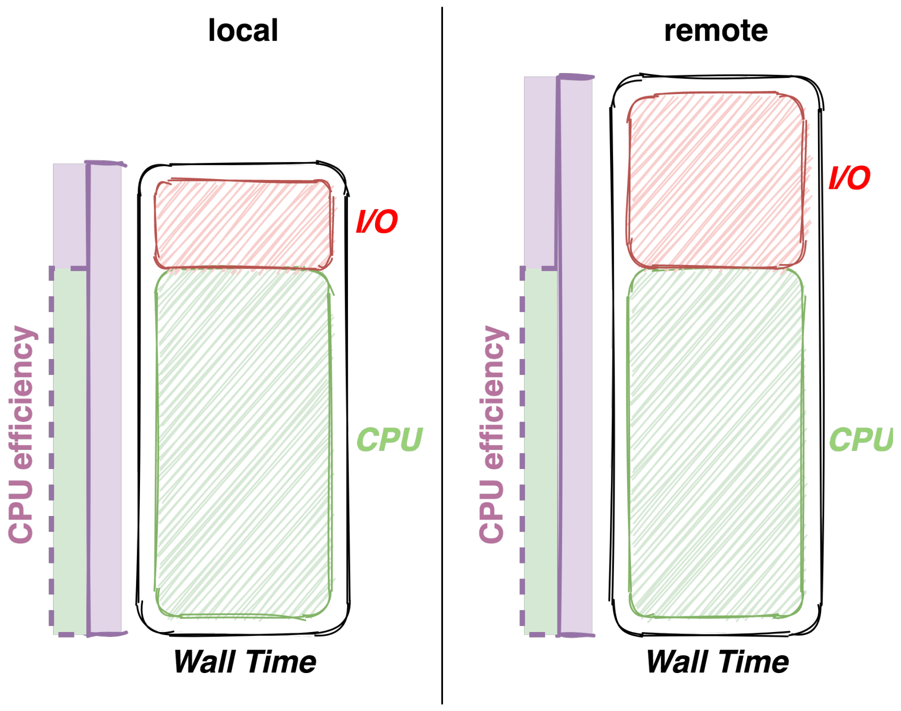


Figure 6.2: CPU efficiency difference between a local and remote served file

6.2 Tests and results

The problem has been addressed with different approaches that have specific targets to test. Using the simulation environment of Section 4.3, the approaches described in Chapter 5 were simulated using as source data the requests of dataset described in Section 4.2.1. As mentioned before, the dataset covers the data analysis requests for the entire year 2018. Thus, the simulation starts to inject requests into the environment for the time specified in the configuration file (Section 4.3). During the environment simulation, the program collects several statistics and stores them in log files that can be viewed and compared with the analyzer tool (Section 4.3).

In the Figure 6.3, it is in evidence the thesis targets upon which a number of experiments have been made:

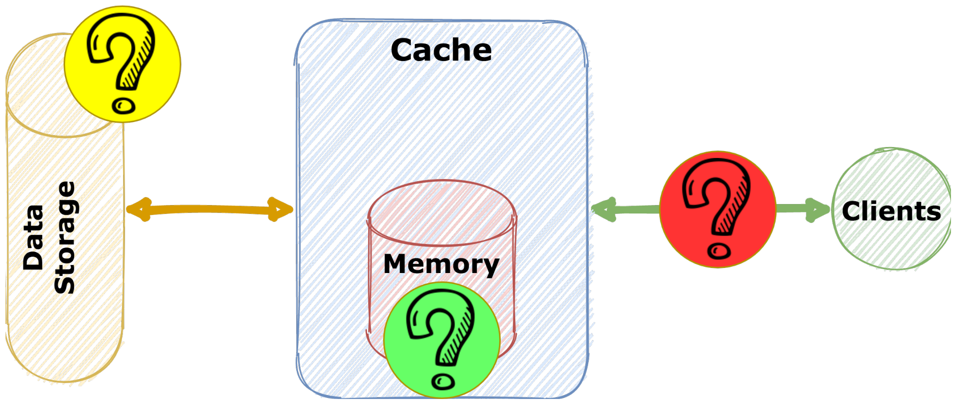


Figure 6.3: Explanation of the experiments' targets compared to the cache model used

The targets are covered by the approaches of Chapter 5, thus there are principally three results from the tests that are characterized as follows:

- **Selection by file score**: this experiment wants to evaluate if it is possible assigning a precise score for a requested file to make a decision upon its addition to the cache memory. The target was to find a way to assign a weight to a file, with the aim to use such information for a better cache composition. It involves the weight function approach of Section 5.1 and it is represented in the Figure 6.3 with the yellow question mark;

- **Cache composition by SCDL**: the experiment aims to find an autonomous agent that decides the files that have to be added into the cache using the client request information. In this case, the question mark on Figure 6.3 is the red one, and it refers to the SCDL approach described in Section 5.2. As a result, it is expected a self-adapting artificial intelligence to filter better the clients' requests;
- **Cache management by SCDL2**: this last experiment attempts to find a smarter way to evict and compose the cached content. The idea is to find two autonomous agents that deal with the two main aspects of the cache memory, the addition, and the eviction. It refers to the SCDL2 approach described in Section 5.3 and involves the question marks green and red in Figure 6.3.

All the experiments aim to find an independent solution to the problem of caching content management, trying to tackle the problem from different points of view. Consequently, each experiment is carried out with a different approach that holds the strategy adopted to solve the specific task. The obtained results will be analyzed using the metrics described in Section 6.1.

6.2.1 Selection by file score

This experiment is performed to investigate how to evaluate a file when it is requested. The cache system collects statistics about the file, such as the number of requests, the size, and the time when the request occurs, to assign a score. This score is treated as a weight, thus the files with a smaller score are preferred with respect to those having a larger score. That means the target function has to respect this constraint.

During this class of experiment, the cache uses the watermark mechanism with the thresholds set for the higher and the lower watermarks respectively to 95% and 75%. All the simulations were made on the Italian dataset (Section 4.2.1) with a realistic but minimal cache size of $100 Terabytes$. All the simulated caches use LRU as a policy to manage the file queue (especially important for the eviction phase).

The statistics of a file remain in the system for 14 days and the mechanism to clear the statistics makes the file checks every 7 days.

Algorithm 6: Algorithm used to test the scoring function.

```
Data: file requests
Result: a cache simulation
Initialization of variables
Inizialization statistic $table$
for each requested file $f$ do
    $t \leftarrow$ time of the request
    $hit \leftarrow checkFileInCache(f)$
    $table.updade(f)$
    $table.updateWeight(f)$
    if hit then
        $updateLRUQueue(f)$
    else
        if $weight(f, t) \leq table.avgWeight()$ then
            $insertWithLRU(f)$
    $checkWatermark()$
```

The Algorithm 6 is used to perform the tests. To resume the approach (Section 5.1), with the high level description of algorithm (Algorithm 6), it inserts a file into the cache only if the file weight is less or equal the average weight of all files of the statistic table. It uses this algorithm with each scoring function $weight_A$, $weigth_E$, $weight_M$ by considering the following values for the parameters α, β and γ: $\{0, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, 1, 2, 4, 8, 16, 32, 64\}$.

Table 6.1 shows the three measures *Throughput* (Equation (6.1)), *Cost* (Equation (6.5)), and *ReadOnHit ratio* (Equation (6.9)) for the LRU cache strategy (used as a baseline in the comparisons) and the best 10 combinations of parameters in each of the 3 functions (Equations (5.2) to (5.4)). Results are sorted by Throughput, Cost, and read on hit ratio. The main measure that is taken into account is the Throughput because the target is to optimize the cache work.

It is possible to see that all the proposed weight functions have a smaller Cost with respect to LRU. Also, they have better Throughput. Thus, they write fewer data into the cache and they still make the clients read a considerable amount of data from their memory.

There is no emergent functional family from the experiment results and all the solutions have a higher load on the network, as it is possible to read from the bandwidth metric.

Observing the parameter values, it is evident that the file size has the most important role (β parameter), whereas the number of requests plays a secondary role (α parameter). Instead, the delta time (γ parameter) does not seem to affect the best results of the functions.

The fact that the frequency is not too relevant has two possible explanations: either the range of the number of requests of a file is not comparable with the other two variables (the size of the file is in Megabytes and the delta time in minutes) or the α parameter should take a much higher value to have a significant impact on the score. The same applies to the γ parameter that seems to be useful just in the multiplicative family (Equation (5.4)) because of its construction.

Furthermore, despite it seems plausible to have the file size as the center of the weight, the delta time could be completely overshadowed by the implicit queue management of the files in the cache. In fact, all the weight functions still use the LRU mechanism to evict files and make space. Hence, all the score techniques try to find a good cache composition through file weights giving LRU a threshold to accept a new one and still they free the memory from oldest and unused files in the same way as LRU. This is the reason why the γ parameter has a low impact on the score function.

In the end, despite poor results of the bandwidth, the CPU efficiency seems not to be compromised and still remaining high. It is $\sim 58\%$ in all the solutions.

Table 6.1: Test results grouped by function family.

cache				**Throughput**	**Cost**	**ReadOnHit ratio**	**Bandwidth**	**CPU eff.**
LRU				-0.1405	0.9799	38.8620	47.68	60.81
function family	α	β	γ	**Throughput**	**Cost**	**ReadOnHit ratio**	**Bandwidth**	**CPU eff.**
Additive	0	$\frac{1}{3}$	0	0.0278	0.3387	23.6848	57.59	59.00
	0	$\frac{1}{2}$	0	0.0278	0.3387	23.6848	57.59	59.00
	0	$\frac{3}{4}$	0	0.0278	0.3387	23.6848	57.59	59.00
	0	1	0	0.0278	0.3387	23.6848	57.59	59.00
	0	16	0	0.0278	0.3387	23.6848	57.59	59.00
	0	2	0	0.0278	0.3387	23.6848	57.59	59.00
	0	32	0	0.0278	0.3387	23.6848	57.59	59.00
	0	4	0	0.0278	0.3387	23.6848	57.59	59.00
	0	64	0	0.0278	0.3387	23.6848	57.59	59.00
	0	8	0	0.0278	0.3387	23.6848	57.59	59.00
AdditiveExp	0	$\frac{1}{3}$	0	0.0414	0.2280	18.9804	60.39	58.26
	0	$\frac{1}{2}$	0	0.0388	0.2512	20.0446	59.73	58.44
	0	$\frac{3}{4}$	0	0.0342	0.2756	20.9988	59.18	58.59
	$\frac{1}{3}$	$\frac{1}{2}$	0	0.0335	0.2537	19.6043	60.02	58.36
	$\frac{1}{3}$	$\frac{3}{4}$	0	0.0331	0.2758	20.8898	59.25	58.57
	$\frac{1}{2}$	$\frac{3}{4}$	0	0.0307	0.2769	20.6754	59.38	58.53
	$\frac{3}{4}$	$\frac{3}{4}$	0	0.0292	0.2778	20.5328	59.48	58.51
	0	1	0	0.0278	0.3387	23.6848	57.59	59.00
	$\frac{1}{3}$	1	0	0.0277	0.3387	23.6747	57.60	59.00
	$\frac{1}{2}$	1	0	0.0271	0.3387	23.5685	57.66	58.98
Multiplicative	0	$\frac{1}{3}$	0	0.0414	0.2280	18.9804	60.39	58.26
	0	$\frac{1}{2}$	0	0.0388	0.2512	20.0446	59.73	58.44
	0	$\frac{3}{4}$	0	0.0342	0.2756	20.9988	59.18	58.59
	32	32	16	0.0339	0.2835	18.2971	59.69	57.14
	32	16	16	0.0291	0.2879	18.1009	59.73	57.08
	32	16	1	0.0279	0.3088	19.2294	59.07	57.30
	0	1	0	0.0278	0.3387	23.6848	57.59	59.00
	32	$\frac{1}{3}$	8	0.0226	0.3934	23.8249	56.54	58.07
	32	$\frac{1}{3}$	$\frac{1}{3}$	0.0210	0.4115	25.1731	55.88	58.33
	32	8	16	0.0180	0.3849	22.3991	57.42	57.76

This partially confirms that the proposed metrics are useful to have an idea of the different cache behaviors with also an approximate side effect evaluation of the end-user experience.

6.2.2 Cache composition by SCDL

The goal of this experiment was to test autonomous intelligence that adapts itself to the client's requests, choosing not with a fixed function which files to write into the cache but continuously evolving during the time.

Within the simulation, the artificial intelligence approach was compared with the golden standard LRU and other commonly used policies (Section 4.3), such as LFU, Size Big, and Size Small. All algorithms have been implemented and tested over a simulation produced with the data described in Section 4.2.1. The experiments were performed using different cache sizes. The test runs in order to demonstrate the ability of SCDL to improve the measures described in Section 6.1 with respect to other caching policies. Because LRU is the main algorithm adopted in a cache layer it is considered as the baseline for these results.

All the tests uses the Algorithm 2 with an ϵ decay rate parameter depending on the dataset used: $\epsilon_{decayrate} = 10^{-5}$ for the Italian dataset and $\epsilon_{decayrate} = 10^{-6}$ for the US dataset (due to its higher amount of requests per day). Moreover, since the simulation covers an entire year of requests, there is a mechanism to unleash the ϵ when the agent is not performing well. In detail, if the $Q-function$ is decreasing for 8 days in a row, the ϵ is reset to 1.0 and the past action history of the files are cleaned.

First, this approach was tested on the Italian dataset with a fixed cache size of $100Terabytes$ and comparing the other standard policies. The results are shown in Table 6.2 attest that the Throughput and the Cost of the SCDL approach are higher compared to the others, also to the most used LRU. The measure results are shown in percentage respect to the denominator defined in Section 6.1.

Table 6.2: Test results of IT region with cache size of $100T$ and $10Gbit$ bandwidth.

Cache Type	Throughput	Cost	ReadOnHit ratio	Bandwidth	Avg. #miss after del.	CPU eff.
SCDL	-0.0032	0.6406	35.37	49.90	462	60.21
LRU	-0.1405	0.9799	38.86	47.68	471	60.81
LFU	-0.2252	1.0538	33.25	51.27	969	59.83
Size Big	-0.3044	1.1230	28.46	54.62	1155	59.30
Size Small	-0.3193	1.1366	27.31	55.29	1181	58.64

It scored the best result for the Throughput measurement. Moreover, the Cost of the Reinforcement Learning approach is half compared to the other policies. A check by looking at the CPU efficiency shows that it is not penalized so much because it scored just $\sim 1\%$ less. Hence, the decision to filter the requests is working quite well, without a heavy impact on the user. Even though the CPU efficiency is only measured to verify the side effects in this particular use case because it uses historical information of a different environment to approximate the cache effect measurement.

Due to the lower performances of the other algorithms, the further tests focus on several cache sizes with a direct comparison with the golden standard policy LRU. The new battery of experiments aims to verify the trend of the filtering agent with respect to the memory available.

Table 6.3: Test results of IT region with different cache sizes and $10Gbit$ bandwidth.

Cache Type	Size	Throughput	Cost	ReadOnHit ratio	Bandwidth	Avg. #miss after del.	CPU eff.
SCDL	500T	0.0474	0.1035	55.84	35.99	766	64.15
SCDL	200T	0.0408	0.3012	42.65	45.04	667	61.61
SCDL	1000T	0.0358	0.0429	65.40	28.76	633	65.91
LRU	500T	0.0328	0.1400	57.50	34.54	828	64.43
LRU	1000T	0.0324	0.0532	67.35	27.10	655	66.25
SCDL	100T	-0.0032	0.6406	35.37	49.90	462	60.21
LRU	200T	-0.0154	0.4409	45.78	43.04	616	62.17
SCDL	50T	-0.1298	1.3210	29.28	53.86	389	59.06
LRU	100T	-0.1405	0.9799	38.86	47.68	471	60.81
LRU	50T	-0.4491	2.1111	33.54	51.29	327	59.83
SCDL	10T	-1.4140	6.8304	22.05	58.29	168	57.70
LRU	10T	-3.3340	11.4636	26.64	55.66	183	58.56

As shown in Table 6.3 this approach still work well also with different cache sizes. Moreover, the most interesting aspect is that there are better results with smaller cache sizes, also compared to the LRU solutions. Those results are very interesting because, from the end-user experience, they seem to maintain a sustainable CPU efficiency with smaller cache requirements. In fact, it is possible to see that even if the *ReadOnHit ratio* is lower in the proposed approach, also the average number of a miss after a file delete is lower. That could be very precious information to sustain a better user experience, this could be useful for infrastructure maintainers.

However, the proposed approach has a slightly higher network load, that is deductible from the bandwidth column (Equation (6.7)).

Despite that, the target to have less work (from the cache perspective) is fulfilled. This RL approach makes the cache low active and does only the minimum operations. This can be seen in a lower amount of written and deleted data but there are several missed files not stored that affect the network because the cache will serve those files in proxy mode (Section 4.1).

To verify the behavior of the agent, another test battery was made on a different dataset, the US one. The results are shown in Table 6.4. This confirms the good behavior of the agent also with a different environment.

Table 6.4: Test results of US region with different cache sizes and $10Gbit$ bandwidth.

Cache Type	**Size**	**Throughput**	**Cost**	**ReadOnHit ratio**	**Bandwidth**	**Avg. #miss after del.**	**CPU eff.**
SCDL	500T	0.1022	0.2229	50.67	89.85	2547	74.09
SCDL	200T	0.0911	0.5246	40.33	91.48	1716	73.08
SCDL	1000T	0.0878	0.1138	59.28	86.06	2985	74.92
LRU	1000T	0.0684	0.1711	61.52	84.31	3215	75.14
LRU	500T	0.0545	0.3636	53.67	88.79	2610	74.38
SCDL	100T	0.0212	1.0142	33.56	91.96	1121	72.43
LRU	200T	-0.0523	0.9335	44.36	90.87	1888	73.47
LRU	100T	-0.3095	1.8825	37.78	91.50	1770	72.83

6.2.3 Cache management by SCDL2

In this latest test bench, two agents are involved. Contrary to the previous experiment, there are several results for this algorithm based on the eviction trigger used. In particular, there are different techniques that characterize the timing with which the eviction agent is called:

- on free (when it is necessary to delete a file)
- at the end of each day
- every k requests (a fixed parameter of the simulation)

Moreover, it is also tested in the case when the eviction agent is deactivated and thus it would be similar to the previous approach.

SCDL2 algorithm has been implemented and tested with the data described in Section 4.2.1. The test ran in order to demonstrate the ability of SCDL2 to improve the measures described in Section 6.1 with respect to other caching policies described in Section 4.3.

The results are shown in Table 6.5 attest that the overall performances of the SCDL2 approach are higher compared to the other policies.

Table 6.5: Test results of IT region with cache sizes of $100T$ and $10Gbit$.

Cache Type	Throughput	Cost	ReadOnHit ratio	Bandwidth	Avg. #miss after del.	CPU eff.
SCDL2 no eviction	-0.02623	0.6789	34.77	50.40	481	60.11
SCDL2 on free	-0.06135	0.6986	31.95	52.42	556	59.55
SCDL2 on day end	-0.07501	0.7116	30.98	52.91	638	59.39
SCDL2 on k	-0.11881	0.7420	28.44	54.73	731	58.87
LRU	-0.14059	0.9799	38.86	47.68	471	60.81
LFU	-0.22528	1.0538	33.25	51.27	969	59.83
Size Big	-0.30448	1.1230	28.46	54.62	1155	59.30
Size Small	-0.31938	1.1366	27.31	55.29	1181	58.64

SCDL2 has the best result for the *Throughput* measurement. However, the *ReadOnHitRatio* for SCDL2 is not the best due to our request filtering, as shown in the previous approach. Also, the eviction agent affects a lot the number of misses and, as a consequence, the overall performance.

Within the simulation, it is also checked the CPU efficiency, and it is verified that it has not substantially changed.

However, the objective of having a lower load from the cache perspective is fulfilled. Moreover, from the results, it is possible to see that LRU is the only option that performs well and thus it will be used as a comparison in the next tests.

In Table 6.6 there are several results with different cache sizes. From these results, it is evident that the eviction agent makes several errors and a higher number of misses after delete that lower down the performances. Despite this, the SCDL2 overall performances are good, and it is possible to conclude that the best trigger solution is the one that frees memory only on demand (*on free*).

To verify the behavior of the agents it was also tested on the bigger US dataset. In Table 6.7 it is evident that the trend of the approach is the same. Moreover, considering the CPU efficiency, it is possible to see that there is a low variance despite the different cache sizes. Therefore, though the eviction agent makes more errors, of course, the user request pattern affects a lot of the cache decisions and the available space is a fundamental parameter for better content management.

To summarize, this RL approach makes the cache less active by doing the minimum number of operations to maintain a good file composition. This results in a lower amount of written and deleted data, but the eviction agent needs a better grasp on which files to delete, to avoid further miss after their deletion. The presence of missed files still affects the network that has a higher load, principally caused by the request filtering of the addition agent. The result tables give a snapshot of the whole simulation, but it has to be mentioned that the results can be visualized also with the analyzer tool described in Section 4.3 (further information can be found in Appendix A.4).

In conclusion, the two agents affect the cache environment in different aspects. The addition agent is the main responsible for writing fewer data and selecting files in a more rigorous way. Hence, the available network bandwidth is more used and the *Throughput* is increased. The eviction agent changes how the files remain in the cache. Hence, its main effects are to increase the *Throughput* and to decrease the *Cost*, maintaining a higher level of Read on

Table 6.6: Test results of IT region with different cache sizes and $10Gbit$ bandwidth.

Cache Type	Size	Throughput	Cost	ReadOnHit ratio	Bandwidth	Avg. #miss after del.	CPU eff.
SCDL2 no eviction	500T	0.0480	0.09	54.10	37.16	641	63.76
SCDL2 no eviction	200T	0.0382	0.29	41.75	45.79	649	61.44
SCDL2 no eviction	1000T	0.0367	0.03	62.87	30.72	499	65.40
LRU	500T	0.0328	0.14	57.50	34.54	828	64.43
LRU	1000T	0.0324	0.05	67.35	27.10	655	66.25
SCDL2 on free	500T	0.0298	0.11	47.35	42.07	951	62.49
SCDL2 on free	1000T	0.0272	0.04	56.02	35.69	1003	64.12
SCDL2 on free	200T	0.0053	0.32	37.43	48.98	661	60.59
SCDL2 on day end	1000T	0.0052	0.06	39.37	47.08	808	61.00
SCDL2 on day end	500T	0.0046	0.13	37.93	48.30	957	60.69
SCDL2 on day end	200T	-0.0046	0.33	35.27	49.94	815	60.17
SCDL2 on k	1000T	-0.0107	0.07	29.30	54.18	621	59.04
LRU	200T	-0.0154	0.44	45.78	43.04	616	62.17
SCDL2 on k	500T	-0.0212	0.14	29.50	54.05	628	59.03
SCDL2 no eviction	100T	-0.0262	0.67	34.77	50.40	481	60.11
SCDL2 on k	200T	-0.0552	0.37	29.30	53.98	740	59.04
SCDL2 on free	100T	-0.0613	0.69	31.95	52.42	556	59.55
SCDL2 on day end	100T	-0.0750	0.71	30.98	52.91	638	59.39
SCDL2 on k	100T	-0.1188	0.74	28.44	54.73	731	58.87
LRU	100T	-0.1405	0.97	38.86	47.68	471	60.81
SCDL2 no eviction	50T	-0.2037	1.48	29.68	53.75	403	59.11
SCDL2 on free	50T	-0.2529	1.49	27.37	55.20	406	58.71
SCDL2 on day end	50T	-0.2583	1.49	26.99	55.45	434	58.63
SCDL2 on k	50T	-0.3038	1.53	26.45	55.73	660	58.52
LRU	50T	-0.4491	2.11	33.54	51.29	327	59.83
SCDL2 no eviction	10T	-2.1534	8.50	23.08	57.78	168	57.89
SCDL2 on free	10T	-2.2246	8.22	21.24	58.91	273	57.54
SCDL2 on day end	10T	-2.2403	8.16	21.08	58.96	280	57.52
SCDL2 on k	10T	-2.3275	8.29	20.92	59.06	450	57.50
LRU	10T	-3.3340	11.46	26.64	55.66	183	58.56

Table 6.7: Test results of US region with different cache sizes with $10Gbit$ bandwidth.

Cache Type	Size	Throughput	Cost	ReadOnHit ratio	Bandwidth	Avg. #miss after del.	CPU eff.
SCDL2 no eviction	500T	0.0968	0.24	51.29	89.79	2565	74.14
SCDL2 no eviction	1000T	0.0880	0.11	59.27	87.18	2944	74.77
LRU	1000T	0.0684	0.17	61.52	84.31	3215	75.14
SCDL2 on free	1000T	0.0644	0.11	54.14	88.87	4347	74.42
SCDL2 no eviction	200T	0.0640	0.62	42.03	91.38	1698	73.24
SCDL2 on free	500T	0.0554	0.23	45.67	90.95	3279	73.60
LRU	500T	0.0545	0.36	53.67	88.79	2610	74.38
SCDL2 on day end	500T	0.0450	0.24	44.44	91.25	2966	73.48
SCDL2 on day end	1000T	0.0322	0.12	46.77	90.88	2755	73.71
SCDL2 on day end	200T	0.0057	0.63	38.14	91.79	2270	72.86
SCDL2 on free	200T	-0.0126	0.62	36.68	91.83	2050	72.73
SCDL2 on k	1000T	-0.0169	0.12	31.37	92.18	1982	72.21
SCDL2 on k	500T	-0.0356	0.25	31.06	92.19	1978	72.18
LRU	200T	-0.0523	0.93	44.36	90.87	1888	73.47
SCDL2 no eviction	100T	-0.0526	1.28	36.05	91.82	1148	72.67
SCDL2 on k	200T	-0.0854	0.62	30.32	93.67	1883	71.98
SCDL2 on free	100T	-0.1284	1.20	31.38	92.18	1674	72.22
SCDL2 on day end	100T	-0.1396	1.21	31.00	92.19	1565	72.18
SCDL2 on k	100T	-0.1822	1.24	29.96	92.22	1984	72.07
LRU	100T	-0.3095	1.88	37.78	91.50	1770	72.83

hit.

6.2.4 Extreme use cases

While developing the simulator, two datasets have been generated, with the Size Focused Dataset Generator Appendix A.3, to test specific extreme use cases. In particular, the data sources generated are:

- Extreme small size dataset: where the 90% of the files are very small (between $8MB$ and $64MB$) and a relatively small set of files have a bigger size (in the range of $1GB$ to $4GB$);
- Extreme big size dataset: where the 90% of the files are big (between $1GB$ $4GB$) and remains set of files have a smaller size (in the range of $8MB$ to $64MB$).

The results made using a cache of 100GB and 10Gbit of bandwidth are reported in Tables 6.8 and 6.9. Because the tables are a summary of the full results, only the SCDL and SCDL2 with *on free* method are shown to attest that both the methods have good behavior with these kinds of extreme situations compared to the standards.

However, the main purpose of a synthetic dataset is to make a further check on the algorithms when proper source data is not available. Hence, the dataset generator represents another important tool for future experiments within the Data Lake environment used in this project, especially to test several stressful situations for the cache system.

Table 6.8: Results of extreme big size synthetic dataset with a cache of 100GB and 10Gbit of bandwidht.

Cache Type	**Throughput**	**Cost**	**ReadOnHit ratio**	**Bandwidth**
SCDL2 on free	3.71	17.19	51.98	1.09
SCDL	-1.14	21.93	41.64	1.33
LRU	-6.99	30.85	35.85	1.46
Size Small	-20.05	43.91	7.93	2.08
Size Big	-22.50	46.37	2.87	2.19
LFU	-23.44	47.30	0.89	2.24

Table 6.9: Results of extreme small size synthetic dataset with a cache of 100GB and 10Gbit of bandwidht.

Cache Type	**Throughput**	**Cost**	**ReadOnHit ratio**	**Bandwidth**
SCDL2 on free	1.74	20.63	47.96	1.3
SCDL	-3.12	24.61	36.91	1.5
LRU	-10.43	36.30	31.36	1.72
Size Small	-24.06	49.93	3.57	2.36
Size Big	-24.26	50.13	3.11	2.37
LFU	-25.45	51.31	0.83	2.43

6.3 Evaluation

The various experiments show that it is possible to improve the proposed metrics. Overall, in all the results there is almost no bad impact from the user's experience point of view. However, the improvements in the Throughput and the Cost metrics are significant and very interesting. In the following sections, there are several insights into the experiment results.

6.3.1 Metrics correlation

The metrics correlation is shown in Figure 6.4. In particular, there is an inverse correlation between the throughput and the cost because, if the cache memory is populated with good files, the throughput will grow and the cost will not be increased. Moreover, the throughput and the read on hit are lightly related to the CPU efficiency, which is strictly related to reading files from the cache. The bandwidth is in slight relation with the number of miss after deletion because, if the cache makes more miss, they will affect the network with the serving in proxy mode (remote files).

Thus, the proposed metrics seem to take a snapshot from different points of view of the simulation and, as described in Section 6.1, it has been intentionally done. However, they could be used to define a new composed evaluation metric that takes into account all these separated aspects that should have taken under control, maybe creating a unique score that can state the goodness of the cache's behavior.

6.3.2 Notes on agent learning

The ϵ decay rate is a fundamental aspect to take under control in RL approaches and usually greatly depends on the environment. In those experiments, the chosen value for the decay allows the agents to have a time span of adaptation (exploration) that is about two weeks. Because of the source used, the chosen decay allows the agents to learn and make better decisions compared to LRU in that timeframe. Thus, it was a good fit for the dataset, and several tests on the ϵ parameter shown the two following situations:

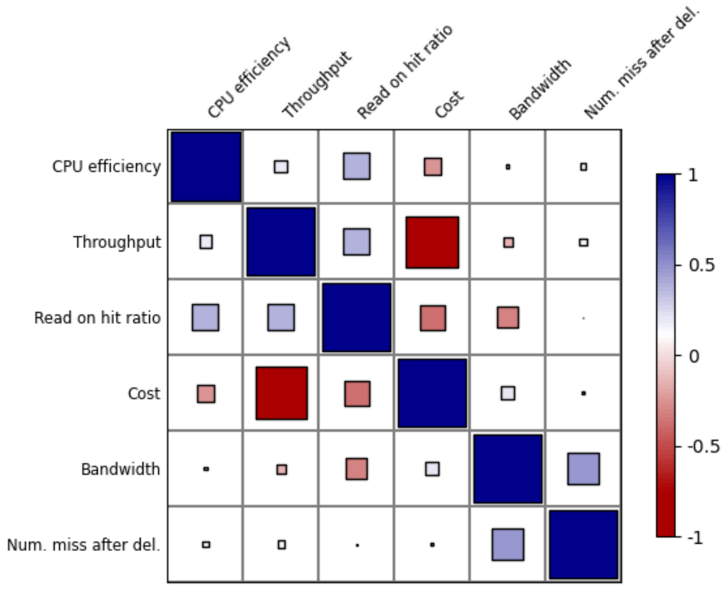


Figure 6.4: Correlation matrix of the various metric used in the experiments

- a fast decay that allows the $epsilon$ to reach the minimum value during a single day will bring a higher read on hit but with a higher cost. The results in a slightly higher throughput compared to the experiment solutions;

- a slow decay that allows the $epsilon$ to reach the minimum value in a range of 3 or 4 months (that is plausible for the context of the problem) allows exploring more before the exploitation also, the results are a lower cost but with lower throughput.

The mentioned scenarios could change because of the variability of the problem, but in Table 6.10 it is depicted how the good value of the ϵ chosen is better than a faster and slower approach.

As a result, the time window chosen for the agent's training seems to be a good balance among all the metric values, especially for the metric Cost, but we must emphasize that this is a parameter strictly related to the source data and that needs more fine-tuning.

Table 6.10: Results of the experiments with a different ϵ decay rate. The table shows the average gain in performance for each metric compared to the ϵ value chosen in the thesis experiments

Metric	Faster ϵ	Slower ϵ
Throughput	-3.95%	-17.92%
Cost	4.22%	-3.92%
Read on hit ratio	1.21%	-3.94%
Bandwidth	-0.36%	1.93%

6.3.3 Notes on selected features

Since the file feature selection is currently the most independent of the problem domain, a further investigation was made to explore the weight of the main file feature for each RL approach used. In those tests, the proposed algorithms are compared using the Italian dataset and a cache of 100Terabytes. In particular, it was selected the SCDL algorithm and SCDL2 with all its variants for the eviction agents. The tests concern about the use or not of the file feature size.

To summarize, the algorithms that did not use the file size won only on 2 cases of 5. The two winners were the methods using only the addition agent. However, a further investigation of the difference between the algorithms that are using the feature size or not shows that globally, it is not convenient to exclude such a feature. In fact, on average, the values of the metrics are not improved and there is a deterioration of the performances. The main loss is in the quantity of the data read directly from the cache memory, as it is possible to deduce from the Table 6.11.

Table 6.11: Results of the experiments without the feature size that show the gain in performance for each metric

Metric	No size
Throughput	-30.13%
Cost	0.4%
Read on hit ratio	-30.14%
Bandwidth	0.92%

In conclusion, it is evident from Table 6.11 that the file size is useful, in some way, for the agents. The fact that the winner algorithms were only SCDL and SCDL2 without the eviction agents also attest that the size is far more important when the cache deletes the files.

6.3.4 Result considerations

As shown in the results (Section 6.2), the principal improvements come from the filtering part (addition agent), which has to evaluate the client file requests, as it is possible to see in Table 6.12. In particular, the SCDL2 approach seems to cost less than the previous SCDL (Table 6.14) when comparing only the addition agent with a cache of the same size. Probably it takes advantage of the full Reinforcement Learning approach but has the same vulnerabilities because it manages the same inputs.

Table 6.12: Comparison of SCDL and SCDL2 in IT region with cache size of $100T$ and $10Gbit$ bandwidth.

Approach	**Throughput**	**Cost**	**ReadOnHit ratio**	**Bandwidth**	**Avg. #miss after del.**	**CPU eff.**
SCDL	-0.0032	0.64	35.37	49.90	462	60.21
SCDL2 no eviction	-0.0262	0.67	34.77	50.40	481	60.11
LRU	-0.1405	0.97	38.86	47.68	471	60.81

Moreover, the weight function experiments show that it is not possible a static solution to evaluate file requests, but it is needed a more auto-adaptive approach such as SCDL. The gain in Cost and Throughput do not justify the loss in performance of the *ReadOnHit ratio*and the greater load of the network (Table 6.13).

However, the eviction aspect is also important, and it could be the key for better management of the whole cache memory that leads also to a better quality of the service (QoS). As shown in Table 6.14, an important aspect for a better cache composition is to monitor the average number of a miss after delete. The eviction agent, depending on the trigger, seems to lack a grasp on this aspect, and this has a bad consequence on future decisions. In fact, also if the two agents do not interact with each other, they still are influenced by the decisions of the other.

Table 6.13: Comparison of weight functions in IT region with cache size of $100T$ and $10Gbit$ bandwidth.

Approach	Throughput	Cost	ReadOnHit ratio	Bandwidth	CPU eff.
Additive	0.0278	0.33	23.68	57.59	59.00
AdditiveExp	0.0414	0.22	18.98	60.39	58.26
Multiplicative	0.0414	0.22	18.98	60.39	58.26
LRU	-0.1405	0.97	38.86	47.68	60.81

Table 6.14: Comparison of SCDL and SCDL2 in IT region with different cache sizes and $10Gbit$ bandwidth.

Cache Type	Size	Throughput	Cost	ReadOnHit ratio	Bandwidth	Avg. #miss after del.	CPU eff.
SCDL2 no eviction	500T	0.0480	0.09	54.10	37.16	641	63.76
SCDL	500T	0.0474	0.10	55.84	35.99	766	64.15
SCDL	200T	0.0408	0.30	42.65	45.04	667	61.61
SCDL2 no eviction	200T	0.0382	0.29	41.75	45.79	649	61.44
SCDL2 no eviction	1000T	0.0367	0.03	62.87	30.72	499	65.40
SCDL	1000T	0.0358	0.04	65.40	28.76	633	65.91
SCDL2 on free	500T	0.0298	0.11	47.35	42.07	951	62.49
SCDL2 on free	1000T	0.0272	0.04	56.02	35.69	1003	64.12
SCDL2 on free	200T	0.0053	0.32	37.43	48.98	661	60.59
SCDL2 on day end	1000T	0.0052	0.06	39.37	47.08	808	61.00
SCDL2 on day end	500T	0.0046	0.13	37.93	48.30	957	60.69
SCDL	100T	-0.0032	0.64	35.37	49.90	462	60.21
SCDL2 on day end	200T	-0.0046	0.33	35.27	49.94	815	60.17
SCDL2 on k	1000T	-0.0107	0.07	29.30	54.18	621	59.04
SCDL2 on k	500T	-0.0212	0.14	29.50	54.05	628	59.03
SCDL2 no eviction	100T	-0.0262	0.67	34.77	50.40	481	60.11
SCDL2 on k	200T	-0.0552	0.37	29.30	53.98	740	59.04
SCDL2 on free	100T	-0.0613	0.69	31.95	52.42	556	59.55
SCDL2 on day end	100T	-0.0750	0.71	30.98	52.91	638	59.39
SCDL2 on k	100T	-0.1188	0.74	28.44	54.73	731	58.87
SCDL	50T	-0.1298	1.32	29.28	53.86	389	59.06
SCDL2 no eviction	50T	-0.2037	1.48	29.68	53.75	403	59.11
SCDL2 on free	50T	-0.2529	1.49	27.37	55.20	406	58.71
SCDL2 on day end	50T	-0.2583	1.49	26.99	55.45	434	58.63
SCDL2 on k	50T	-0.3038	1.53	26.45	55.73	660	58.52
SCDL	10T	-1.4140	6.83	22.05	58.29	168	57.70
SCDL2 no eviction	10T	-2.1534	8.50	23.08	57.78	168	57.89
SCDL2 on free	10T	-2.2246	8.22	21.24	58.91	273	57.54
SCDL2 on day end	10T	-2.2403	8.16	21.08	58.96	280	57.52
SCDL2 on k	10T	-2.3275	8.29	20.92	59.06	450	57.50

Further studies were made to investigate the motivation and the possible bottleneck where the approaches can improve. In particular, as shown in Figure 6.5, it is possible to infer that there is great variability of user requests during the days.

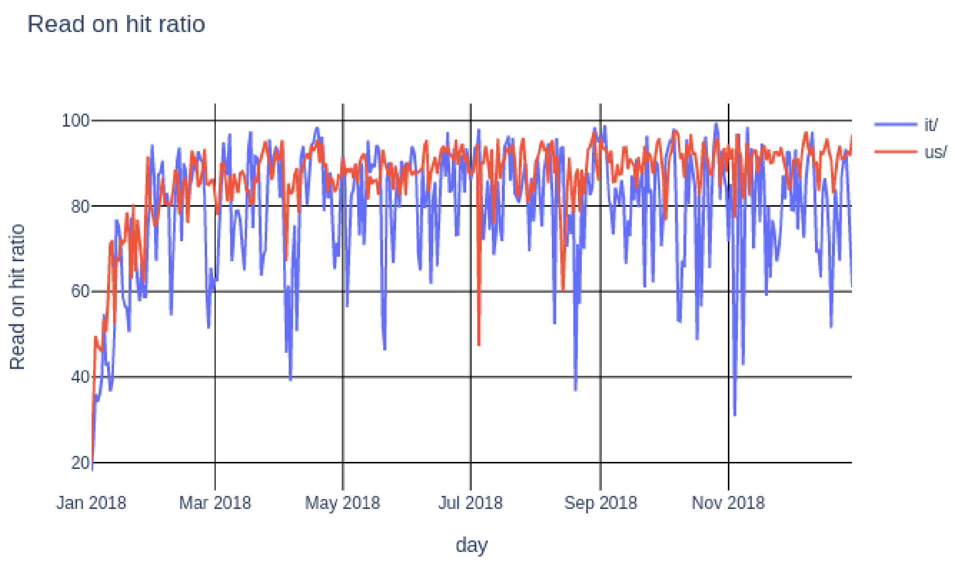


Figure 6.5: Result investigation of *Read on hit ratio* for both datasets with a cache that has infinite memory

In Figure 6.5 there is a focus of the read on hit with respect to the total amount of file requested from the user using a cache with infinite size. Despite the dataset used, there are days with a low percentage of files read from the cache, also 30%. As a consequence, the file requested only once influenced a lot of the caching behaviors and consequently the user experience. Due to the lower space available in the cache and the enormous volume of the source data (in the range of hundreds of Petabytes), it is not trivial to manage the cached content. In fact, the ending cache size for the Italian and US dataset at the end of the simulation was respectively $\sim 12 Petabytes$ and $\sim 16 Petabytes$. Thus, if the portion of data used in a year respects the source is so different it is necessary to use domain-specific information to better understand and guess the clients' patterns.

All the approaches prove that it is possible to enhance the caching content management in a Data Lake architecture. Also, they performed well compared to the golden standard LRU policy that is commonly used in the problem context. From the direct application perspective, the most concrete result of the given experiments is visible in the following Table 6.15:

Table 6.15: Result comparison of IT region with different cache sizes and $10Gbit$ bandwidth.

Cache Type	Size	Throughput	Cost	ReadOnHit ratio	Bandwidth	Avg. #miss after del.	CPU eff.
SCDL2 no eviction	500T	0.0480	0.09	54.10	37.16	641	63.76
SCDL2 no eviction	200T	0.0382	0.29	41.75	45.79	649	61.44
LRU	500T	0.0328	0.14	57.50	34.54	828	64.43
LRU	1000T	0.0324	0.05	67.35	27.10	655	66.25

Because of the cost of the storage and the high dynamism of the environment, the fact that the cache with less space performed better than the standard LRU cache with higher memory is a valuable result that can be used in practice to save storage costs. In Figures 6.6 and 6.7 is visible the constant work of the SCDL2 with no eviction to improve the throughput compared to golden standard LRU with the same size and also against its double (Table 6.15). As depicted from the box plot, the SCDL2 is always better than its sibling LRU cache with 500T, but it does a good job against its double, of course with a higher variance due to the difference in the size of the two caches. In fact, there are more spikes on the comparison with the 1000T LRU cache, especially negative, because of the possibility by the bigger cache to ingest more files.

In the WLCG scenario, one of the main focuses is resource optimization, with the ultimate aim of improving performance and efficiency, as well as simplifying and reducing operation costs. Thus, with such an approach it is possible to enhance the XCache middleware, the current environment used in CMS for the caching layer, and implemented with XRootD [56, 57], to take advantage of the double agent configuration and improve the throughput, without changing the technology (Figures 6.6 and 6.7). In conclusion, the great value of the work is not just an autonomous solution to improve the current caching layer performance in the WLCG Data Lake context, but also a method to contain the costs to face the future requirements.

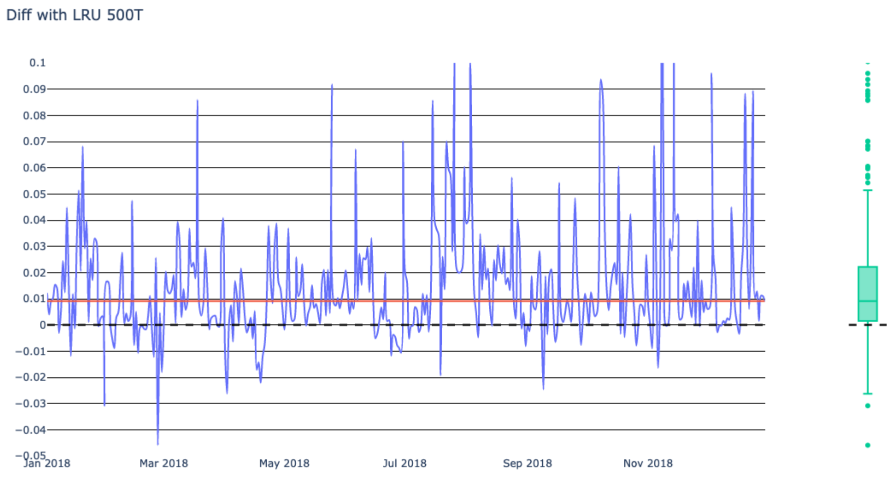


Figure 6.6: This plot shows the difference between SCDL2 with no eviction and LRU, using a cache of size 500T. The red line is the average gain of SCDL2

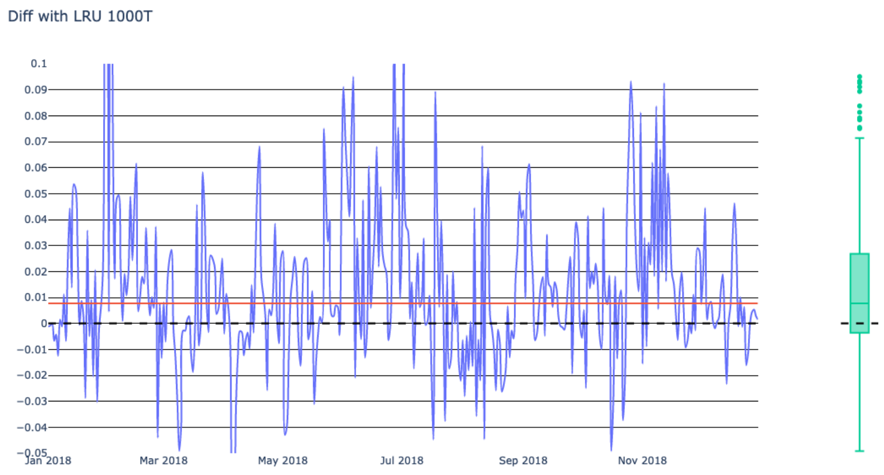


Figure 6.7: This plot shows the difference between SCDL2 with no eviction and a cache size of 500T versus a cache of size 1000T that uses LRU. The red line is the average gain of SCDL2

Future steps

Contents

7.1 Q-Learning improvements 104
7.2 Deep Q-Learning 105
7.3 Integration 106

In this chapter, the future of the work and the possible improvements will be discussed. An idea of the weakness of the approach is depicted and there will be addressed several aspects to enhance the agents.

Since the approach of this thesis work is as general as possible, the techniques proposed could be used also in other contexts, for example, cloud cache for file-sharing or video hosting services. Moreover, the environment could be enriched not only with domain information regarding the data but also with additional system information: e.g. network topology, the status of other caches, latency information, etc. However, even if the project is agnostic, it does not mean that the adaptation is simple and above all, it requires appropriate datasets for simulations and a further study on the useful features to use.

Regarding the portability of the approach, because the project's target was to create an independent object, it is possible to plug in other environments without any effort. During the development were tested communications with other services using the HTTP protocol [58] and also gRPC [59]. The Go language has many frameworks to interact with other ecosystems, thus the developed solution could be deployed as a service in any environment, from the IoT to HPC. In conclusion, due to its cloud-native origins, this project can be deployed in any cloud environment using also the containerization and also systems like Kubernetes [60].

7.1 Q-Learning improvements

As shown in the results, the agents' behavior can be improved. In particular, the number of a miss after delete indicates that there is room for better performances comparing the LRU policy. The inevitable amount of miss for files requested once could be mitigated with a better decision both, on addition and on the eviction. The aim is still the same, we have to use better the cache space.

To investigate better, it was made an analysis of smaller periods of the year (3 months each period), where the cache behavior was deeply monitored. In Figure 6.5 is depicted that often the agents make fewer mistakes compared to the standard cache policy and this is a confirmation of the lower average number of a miss on delete encountered in the experiments.

However, the analysis of the type of miss shows that most of the cases are caused by the addition agent, because there is a miss after the choice

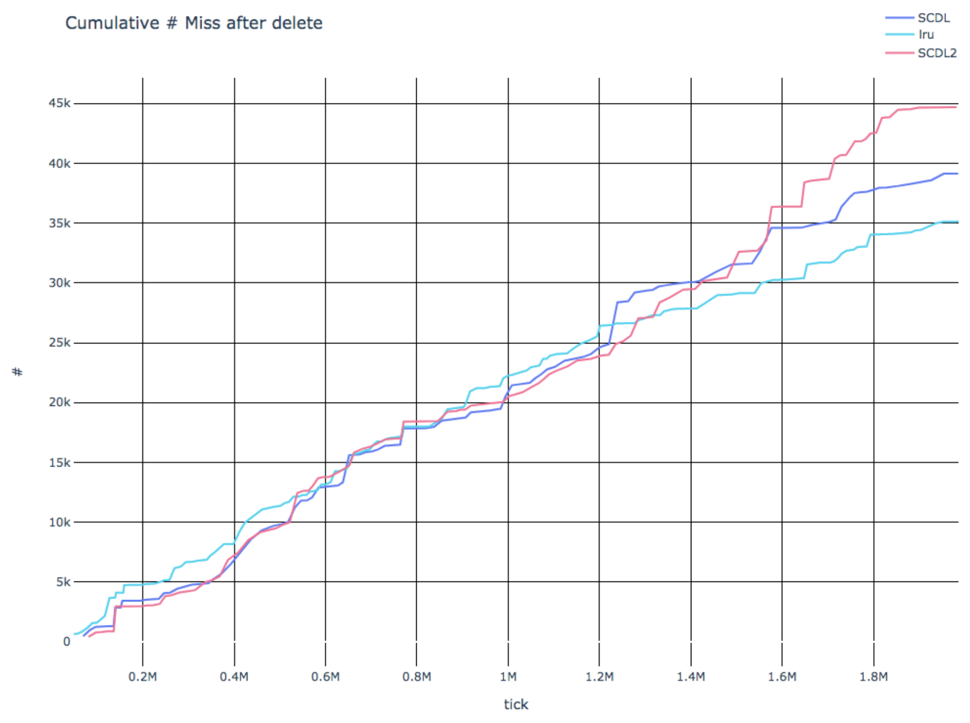


Figure 7.1: Cumulative number of miss on delete for period B of the year (Apr-Jul) using the Italian dataset

to not store a file. In addition, the eviction agent still removes some useful files because there are several misses after the deletion of the file (Figures 7.2 and 7.3).

Altogether, both the agents need an improvement: the addition agent should make fewer mistakes, especially for the files requested two or more times; the eviction agent has to be more precise on deletion, hopefully with the ability to choose and evaluate the deletion of single files.

7.2 Deep Q-Learning

As mentioned in the Section 6.3 there is great variability of the requests during the days. Thus, there is a need for a predicting approach to tackle client requests. A possible solution could be the Deep Q-Learning technique, together with the use of domain information. With a deeper comprehension of

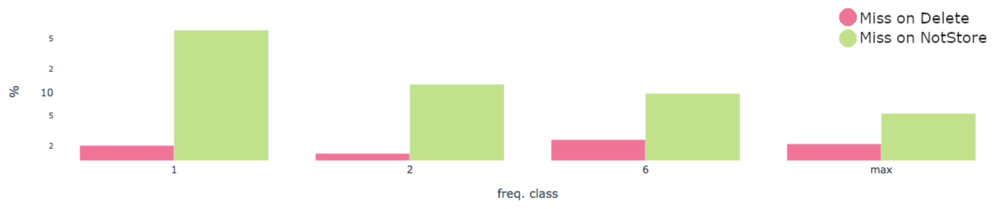


Figure 7.2: Frequency distribution of miss on delete an miss on skip for SCDL in period B (Apr-Jul 2018) for the Italian dataset

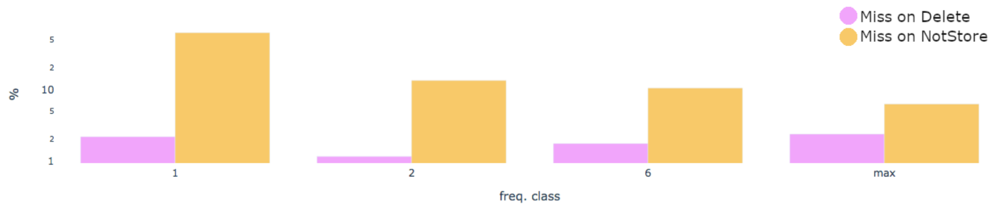


Figure 7.3: Frequency distribution of miss on delete an miss on skip for SCDL2 with no eviction in period B (Apr-Jul) for the Italian dataset

the request, the agent could infer better if a file will be requested in the period after, for example using the data type or the number of different users that have requested that information.

Maybe this RL technique could take advantage of an environment model that gives the agents more precise and selective information. However, this model is certainly more complex to implement online.

7.3 Integration

The WLCG Data Lake model (Section 4.2) is a concept introduced to optimize operational and hardware costs, improving efficiency at HL-LHC. In such a model, storage is consolidated in fewer logical endpoints, and compute capacity is not necessarily co-located with storage. On the contrary, some facilities could offer large compute capacity and no persistent storage at all. Stateless caches would allow data to be buffered close to the compute resources for further re-use, while no persistent storage would be operated at that site. Latency hiding technologies would allow data to be processed as if it was local, given that enough bandwidth is available to fetch it.

For this purpose, the experiments have initiated a major R&D program to develop new ideas to maximize the positive experience from physicists with expected manpower and funding at the sites.

This program was formed into the Data Organisation, Management and Access (DOMA) project, embedded within the WLCG framework. The objective of the DOMA project has been to share, discuss, and debate the merits of the ideas and results emerging from this R&D program. The project has collaborated with the developers of relevant new and emerging technologies, with the objective of finding tools and solutions suitable beyond the LHC community, in an attempt to achieve more standardization.

From the data centers point of view, the first step to earn from this technology is the application of the caches to optimize the use of the main storage also within its redundant copies. The importance of efficient utilization of a heterogeneous infrastructure is already the first goal to face the future in data storage.

In this context, a valuable aspect is to try to estimate the impact of the discussed approach, also from a different perspective: the financial one. In order to do that, it is possible to use the current statistics and budget information for the Italian region within the CMS Experiment as a reference scenario for LHC:

- in Italy there are $10PB$ of disk space deployed in order to serve the computing system of the CMS Experiment;
- the current target is the analysis datasets of CMS (namely MiniAOD and NanoAOD) which are a fraction of the whole source data;
- the current estimated network cost is ~ 140€ per TB.

Therefore, defining a possible scenario where approximately $\frac{1}{3}$ of the current total amount of disk space which is devolved to a cache-based mechanism it is assumed that $7PB$ of the total amount of disk space is managed as usual while 3 can be used as a cache.

Regarding the cache, it is assumed to not have any responsibility for data loss. Consequently, the default policy of CMS in terms of the replica is not applied to the disk space. Conservatively, it should be estimated a gain of a factor of $\times 2$ that is obtained in terms of disk capacity. On top of that, the proposed AI strategy can be applied and, considering the measured results of the presented work, a factor 2 with regard to the standard LRU behavior.

Overall this means that a $2.25TB$ of disk space out of the 3 originally considered can be saved without any performance loss. Aiming at translating this into budget-saving, one could say that a smart cache-based approach can save up to 22.5% of the total budget spent on disk in Italy to support the CMS experiment. Based on the assumption, could be calculated the net cost of the disk, that is means about 300000€ per year. Out of which, 100000€ would come from the AI algorithm presented in this thesis.

Of course, with the same assumptions, the gain estimation would be proportional to the amount of the disk managed as a cache. Moreover, it should be noted that the estimated gain discussed before does not include several important factors:

- Ease of operations such as the bulk of disk space is JBOD, and losses are handled automatically upstream;
- Overall, spend a larger fraction of total funding on CPU and GPU than today;
- Easier management of corrupted file related issues.

Despite the above example made with today's information, the caching improvements proposed can be applied as a tool to improve the infrastructure and belong to the DevOps field. In fact, it is a collaboration between Dev (development) and Ops (IT operations) sectors in a process that makes a union of people and technology to continually provide value to customers. Consequently, the expected result of the project would be deployed in the real-life environment mentioned before.

To summarize, one of the expected results from the integration and experimentation phase of a testbed is to make an on-field evaluation of the values obtained within the thesis work.

Machine Learning Infrastructure

The developed tool could have a separate environment where all the calculus is made and where the AI takes place. However, managing the resources to apply machine learning techniques is not trivial, especially when we have to deal with Big Data or opportunistic sites. For this reason, there are solutions like DODAS [61, 56] (Dynamic On Demand Analysis Service) to deploy a typical Machine Learning pipeline (Figure 7.4) on-demand, and with custom requirements. DODAS is an open-source project for creating analysis container-based clusters on-demand on any cloud infrastructure.

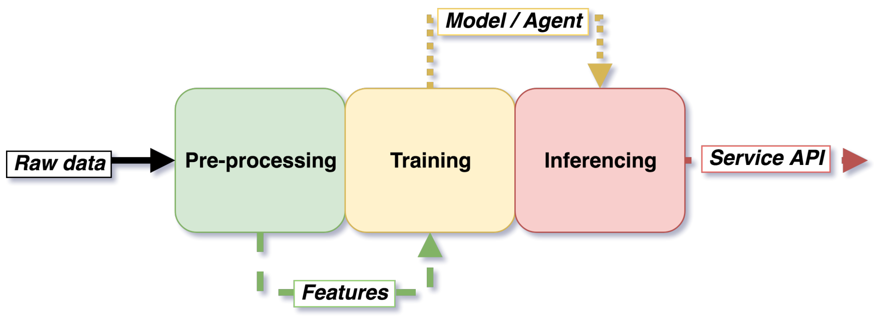


Figure 7.4: Schema of a machine learning typical pipeline

The ML pipeline is composed of independent modules and each module provides a specific service for the ML tool-chain. DODAS lets get to have a highly generic implementation of these building blocks and also to create a platform with those blocks on "any" cloud provider with a minimal effort, enabling self-healing and scale-up capabilities. This is a perfect combination of features that can help to deploy the tool solution with continuous integration with the environment.

As summarized in Figure 7.5, DODAS provides container-based solutions to instantiate several solutions. In particular, in the context of Big Data, it provides the creation of Hadoop clusters, Spark clusters, and generic ML frameworks for training and inference. In order to implement our service, we used these available blocks to automatically compose the tool-chain for our experiments.

In the context of this project, it would be used as enabling technology for the proposed approaches [56]. In particular, a custom container with the required

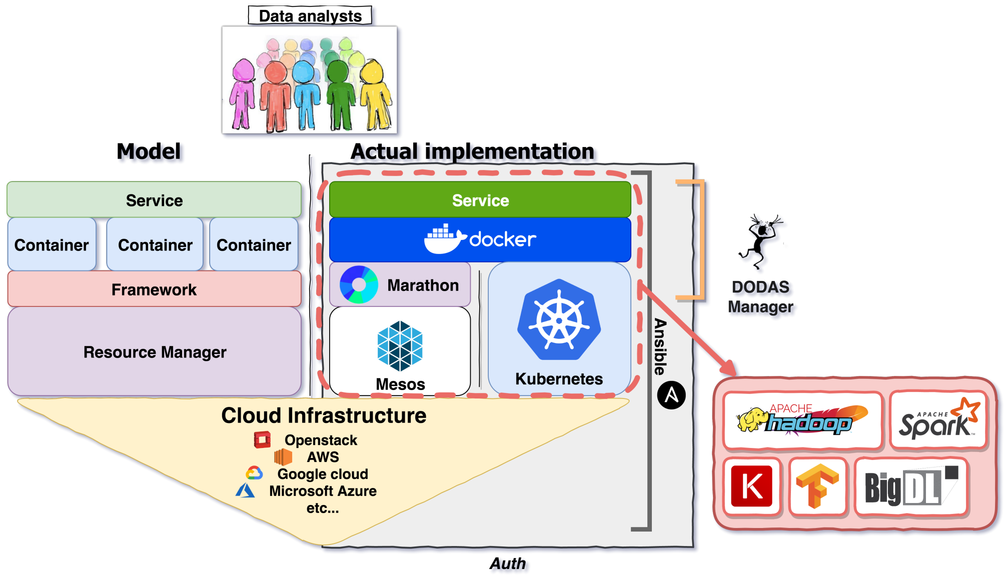


Figure 7.5: DODAS architecture schema showing how the stack is implemented

dependencies for each approach can be injected and executed as a service on-demand within the DODAS infrastructure.

XcachePlugin

One of the technologies used for the caching layer in the CMS context (Section 4.2) but, generally in the High Energy Physics, is the Root cache and proxy framework named XrootD [62, 57]. Other caching technologies were considered, but they were discarded because they require more integration work with the HEP workflow. Consequently, it is possible to deploy one of the solutions proposed to work directly in the real-world scenario simply by implementing the plugin interface for such a framework.

Thus, to insert a new technique to control the XCache content management it is possible to use the plugin interface itself, as shown in Figure 7.6 with a typical external Machine Learning pipeline mentioned in the previous section.

However, since the project is open source, if the approach needs deep control over the system it is possible to create several external services and modify the source code to match the requirements. Moreover, there could be more

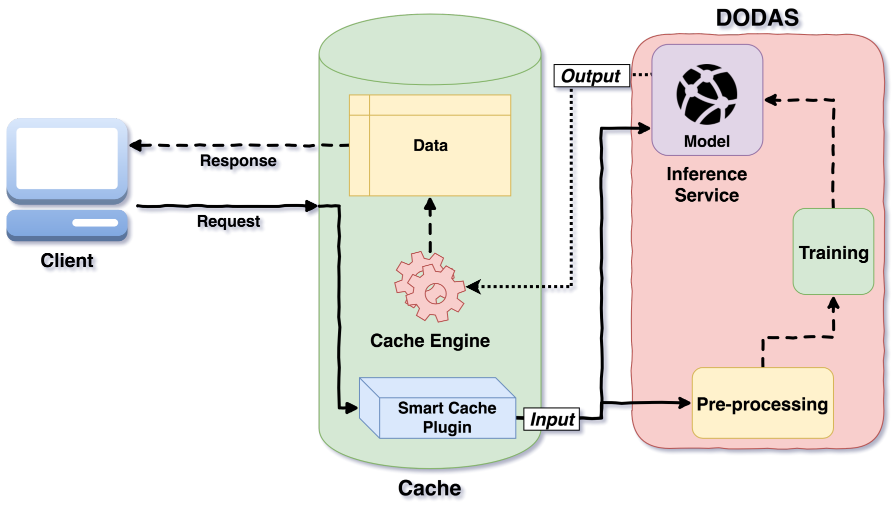


Figure 7.6: XCache plugin schema for AI integration which shows where it is possible to intervene and extend the framework

sophisticated interaction with the cache infrastructure depending on the protocol used while maintaining a configuration with independent microservices between them.

Conclusions

Data Lake is an emergent concept born to deal with the huge amounts of data and also to sustain the incessant increase in customer requests. In a world where Big Data changed everything, from the paradigm used to access them to the management of the information, Data Lake seems the possible answer to face future challenges.

Within this context, there are several models and interpretations of the Data Lake infrastructure. The one that is forming in the field of High Energy Physics, also thanks to the WLCG vision, is a storage service of geographically distributed data centers connected by a low-latency network.

As described in this thesis, the key to such a Data Lake infrastructure is better and autonomous caching data management. The caching layer is an important part of the process of data access and if it works well it should increase the user task throughput.

This Ph.D. program was devoted to define and design an autonomous caching system able to enhance the caching content management, improving the data throughput towards the clients, and lower the operational costs. Moreover, the proposed solution has to adapt itself to new situations without human intervention.

This thesis addresses the problem from different points of view, also by creating a simulation environment to test different techniques and proposing several metrics to monitor the effects of the algorithms.

The work produced three different approaches: the first one is a selection file method based on a *Weight Function* able to evaluate the goodness of a file guessing its future desirability. The target of this approach is to evaluate each file request in order to decide either to store the data or not.

The other two methods instead use a Reinforcement Learning approach. The first one, called SCDL (Smart Caching for Data Lake), has only one agent that decides whether to store or not a file; the agent uses a technique similar to the multiarmed bandit and its decisions are based only on client requests.

The second one, called SCDL2, implements a full Reinforcement Learning approach based on the Q-Learning technique that involves two agents. This time, the agents are enabled to make decisions on both file addition and eviction not only using the information taken from the file request but also exploiting the information taken from the status of the cache memory.

The results show that the approaches gain in performances compared to other standard algorithms used in cache management, in particular, compared to LRU. In detail, all the solutions proposed have a lower cost reaching a profit starting from $\sim 25\%$ up to $\sim 50\%$, considering the same cache size. Furthermore, the throughput of the cache is improved by $\sim 15\%$ up to $\sim 30\%$.

Despite the high dimensionality of the problem and the complexity of the context, this thesis set the first milestone for autonomous caching content management that outperforms the standard algorithms already used. This is a usable solution in Data Lake environments and in particular in the HEP Data Analysis workflows.

The used approaches contribute to better memory management, giving value to the cached content and thus valuing the cache memory. The results obtained in the case of lower cache sizes are promising and this particularly contributes to the objective of cost containment. From this point of view, this work can be considered ready to be used in a real context adding a significant value to the Data Lake model.

The obtained results have room for improvements, especially if the algorithms start to use more domain-specific information. In the future, this technique could be also enhanced with more fine control over file decisions, with a more sophisticated approach like Deep Reinforcement Learning, or also with the support of external models of the used environment. In particular, all the aspects related to the quality of service could be assigned to artificial intelligence that auto adapts itself for better infrastructure maintenance, giving information about the real impact of the caching layer and the improvement of user experience. Moreover, if the future of data storage within Data Lakes will be side by side with a caching system to enhance the content flow, this work provides a first perspective on what could be a future profit thanks to the better autonomous cache management.

In conclusion, the aim to create an autonomous object to enhance the caching layer is achieved. This original contribution that can adapt itself to the user requests and improve the flow of the data from the cache memory shows considerable performance. Also, this work brought several metrics that could be useful for further comparisons and developments in a field of continuous evolution such as the Data Lakes.

Bibliography

[1] I. Bird, S. Campana, M. Girone, X. Espinal, G. McCance, and J. Schovancová. Architecture and prototype of a wlcg data lake for hl-lhc. In *EPJ Web of Conferences*, volume 214, page 04024. EDP Sciences, 2019.

[2] I. Kadochnikov, I. Bird, G. McCance, J. Schovancova, M. Girone, S. Campana, and X. E. Currul. Wlcg data lake prototype for hl-lhc. *Advisory committee*:127, 2018.

[3] C. Xu, K. Wang, P. Li, R. Xia, S. Guo, and M. Guo. Renewable energy-aware big data analytics in geo-distributed data centers with reinforcement learning. *IEEE Transactions on Network Science and Engineering*, 2018.

[4] Y. He, F. R. Yu, N. Zhao, V. C. Leung, and H. Yin. Software-defined networks with mobile edge computing and caching for smart cities: a big data deep reinforcement learning approach. *IEEE Communications Magazine*, 55(12):31–37, 2017.

[5] P. Mell, T. Grance, et al. The nist definition of cloud computing, 2011.

[6] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra, and B. Hu. Everything as a service (xaas) on the cloud: origins, current and future trends. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 621–628. IEEE, 2015.

[7] A. Gorelik. *The enterprise big data lake: Delivering the promise of big data and data science*. O'Reilly Media, 2019.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] K. Shvachko, H. Kuang, S. Radia, and R. Chansler.
The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[10] D. Baum. *Cloud Data Lakes For Dummies.*
John Wiley & Sons, Inc., 111 River St. Hoboken, NJ 07030-5774, 2020.

[11] J. Wang. A survey of web caching schemes for the internet.
ACM SIGCOMM Computer Communication Review, 29(5):36–46, 1999.

[12] N. G. Smith. The uk national web cache—the state of the art.
Computer Networks and ISDN Systems, 28(7-11):1407–1414, 1996.

[13] B. D. Davison. A web caching primer.
IEEE internet computing, 5(4):38–45, 2001.

[14] S. Podlipnig and L. Böszörmenyi.
A survey of web cache replacement strategies.
ACM Computing Surveys (CSUR), 35(4):374–398, 2003.

[15] W. Ali, S. M. Shamsuddin, A. S. Ismail, et al.
A survey of web caching and prefetching.
Int. J. Advance. Soft Comput. Appl, 3(1):18–44, 2011.

[16] G. Tian and M. Liebelt.
An effectiveness-based adaptive cache replacement policy.
Microprocessors and Microsystems, 38(1):98–111, 2014.

[17] T. Koskela, J. Heikkonen, and K. Kaski.
Web cache optimization with nonlinear model using object features.
Computer Networks, 43(6):805–817, 2003.

[18] T. Chen. Obtaining the optimal cache document replacement policy for the caching system of an ec website.
European Journal of Operational Research, 181(2):828–841, 2007.

[19] J. Dixon. Pentaho, hadoop and data lakes.
https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/, 2010. Last check April 9, 2020.

[20] T. King. The emergence of data lake: pros and cons.
https://solutionsreview.com/data-integration/the-emergence-of-data-lake-pros-and-cons/, 2016.
Last check April 9, 2020.

[21] P. P. Khine and Z. S. Wang. Data lake: a new ideology in big data era.
In *ITM Web of Conferences*, volume 17, page 03025. EDP Sciences, 2018.

[22] Q. Yang, M. Ge, and M. Helfert.
Analysis of data warehouse architectures: modeling and classification, 2019.

[23] V. K. Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z.-L. Zhang.
Unreeling netflix: understanding and improving multi-cdn movie delivery.
In *2012 Proceedings IEEE INFOCOM*, pages 1620–1628. IEEE, 2012.

[24] A. Burkov. *The hundred-page machine learning book*, volume 1.
Andriy Burkov Quebec City, Can., 2019.

[25] P. Harrington. *Machine learning in action*.
Manning Publications Co., 2012.

[26] A. Burkov. Machine learning engineering, 2019.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning.
arXiv preprint arXiv:1312.5602, 2013.

[28] J. Achiam. Spinning up in deep rl, 2018.
URL https://spinningup. openai. com.

[29] S. Dreyfus. Richard bellman on the birth of dynamic programming.
Operations Research, 50(1):48–51, 2002.

[30] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*.
MIT press, 2018.

[31] J. Gray and P. Shenoy. Rules of thumb in data engineering.
In *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*, pages 3–10. IEEE, 2000.

[32] L. Lei, L. You, G. Dai, T. X. Vu, D. Yuan, and S. Chatzinotas. A deep learning approach for optimizing content delivering in cache-enabled hetnet. In *2017 international symposium on wireless communication systems (ISWCS)*, pages 449–453. IEEE, 2017.

[33] S. Basu, A. Sundarrajan, J. Ghaderi, S. Shakkottai, and R. Sitaraman. Adaptive ttl-based caching for content delivery. *IEEE/ACM transactions on networking*, 26(3):1063–1077, 2018.

[34] A. Narayanan, S. Verma, E. Ramadan, P. Babaie, and Z.-L. Zhang. Deepcache: a deep learning based framework for content caching. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, pages 48–53, 2018.

[35] T. Lykouris and S. Vassilvitskii. Competitive caching with machine learned advice. *arXiv preprint arXiv:1802.05399*, 2018.

[36] H. Herodotou. Autocache: employing machine learning to automate caching in distributed file systems. *International Conference on Data Engineering Workshops (ICDEW)*:133–139, 2019.

[37] A. Sadeghi, G. Wang, and G. B. Giannakis. Deep reinforcement learning for adaptive caching in hierarchical content delivery networks. *IEEE Transactions on Cognitive Communications and Networking*, 5(4):1024–1033, 2019.

[38] G. Dulac-Arnold, R. Evans, H. van Hasselt, P. Sunehag, T. Lillicrap, J. Hunt, T. Mann, T. Weber, T. Degris, and B. Coppin. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*, 2015.

[39] C. Zhong, M. C. Gursoy, and S. Velipasalar. A deep reinforcement learning-based framework for content caching. In *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*, pages 1–6. IEEE, 2018.

[40] C. Collaboration, S. Chatrchyan, G. Hmayakyan, V. Khachatryan, A. Sirunyan, W. Adam, T. Bauer, T. Bergauer, H. Bergauer, M. Dragicevic, et al. The cms experiment at the cern lhc, 2008.

[41] R. Brun and F. Rademakers.
Root—an object oriented data analysis framework.
Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 389(1-2):81–86, 1997.

[42] G. Aad, J. Butterworth, J. Thion, U. Bratzler, P. Ratoff, R. Nickerson, J. Seixas, I. Grabowska-Bold, F. Meisel, S. Lokwitz, et al.
The atlas experiment at the cern large hadron collider.
Jinst, 3:S08003, 2008.

[43] K. Aamodt, A. A. Quintana, R. Achenbach, S. Acounis, D. Adamová, C. Adler, M. Aggarwal, F. Agnese, G. A. Rinella, Z. Ahammed, et al.
The alice experiment at the cern lhc.
Journal of Instrumentation, 3(08):S08002, 2008.

[44] A. A. Alves Jr, L. Andrade Filho, A. Barbosa, I. Bediaga, G. Cernicchiaro, G. Guerrer, H. Lima Jr, A. Machado, J. Magnin, F. Marujo, et al.
The lhcb detector at the lhc.
Journal of instrumentation, 3(08):S08005, 2008.

[45] A. Georgiou. Storing Data Flow Monitoring in Hadoop. Technical report, 2013.

[46] V. Kuznetsov, T. Li, L. Giommi, D. Bonacorsi, and T. Wildish.
Predicting dataset popularity for the cms experiment.
arXiv preprint arXiv:1602.07226, 2016.

[47] M. Meoni, R. Perego, and N. Tonellotto.
Dataset popularity prediction for caching of cms big data.
Journal of Grid Computing, 16(2):211–228, 2018.

[48] J. Meyerson. The go programming language.
IEEE software, 31(5):104–104, 2014.

[49] A. A. Donovan and B. W. Kernighan. *The Go programming language.* Addison-Wesley Professional, 2015.

[50] O. Ben-Kiki, C. Evans, and B. Ingerson.
Yaml ain't markup language (yaml™) version 1.1.
Working Draft 2008-05, 11, 2009.

[51] Y. Shafranovich. Common format and mime type for csv files. *Internet Eng. Task Force draft, Mar*, 2005.

[52] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN: 1441412697.

[53] P. T. Inc. Collaborative data science. 2015. URL: https://plot.ly.

[54] M. Tracolli, M. Baioletti, D. Ciangottini, V. Poggioni, and D. Spiga. An intelligent cache management for data analysis at cms. In *International Conference on Computational Science and Its Applications*, pages 320–332. Springer, 2020.

[55] D. Ciangottini. Integrazione di una smart cache italiana federata per cms, 2019. URL: %5Curl%7Bhttps://web.infn.it/CCR/index.php/it/la-ccr/home-1%7D. talk at CCR, Italy.

[56] M. Tracolli, M. Antonacci, T. Boccali, D. Bonacorsi, D. Ciangottini, G. Donvito, C. Duma, L. Gaido, D. Salomoni, D. Spiga, and V. Kuznetsov. Using DODAS as deployment manager for smart caching of CMS data management system. *Journal of Physics: Conference Series*, 1525:012057, Apr. 2020. DOI: 10.1088/1742-6596/1525/1/012057. URL: https://doi.org/10.1088%2F1742-6596%2F1525%2F1%2F012057.

[57] L. Bauerdick, K. Bloom, B. Bockelman, D. Bradley, S. Dasu, J. Dost, I. Sfiligoi, A. Tadel, M. Tadel, F. Wuerthwein, et al. Xrootd, disk-based, caching proxy for optimization of data access, data placement and data replication. *Journal of Physics: Conference Series*, 513(4):042044, 2014.

[58] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol–http/1.1, 1999.

[59] M. Marculescu. Introducing grpc, a new open source http/2 rpc framework. *Google Open Source Blog*, 2015.

[60] D. Bernstein. Containers and cloud: from lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[61] D. Spiga, M. Antonacci, T. Boccali, D. Ciangottini, A. Costantini, G. Donvito, C. Duma, M. Duranti, V. Formato, L. Gaido, et al. Dodas: how to effectively exploit heterogeneous clouds for scientific computations. *PoS (ISGC 2018 & FCDD)*, 24, 2018.

[62] XRootD. Xrootd homepage, 2020. URL: `%5Curl%7Bhttps://xrootd.slac.stanford.edu/index.html%7D`. Last accessed March 4, 2020.

[63] D. Crockford. Ecma-404, the json data interchange standard, 2017.

Appendix

A.1 Cloud services

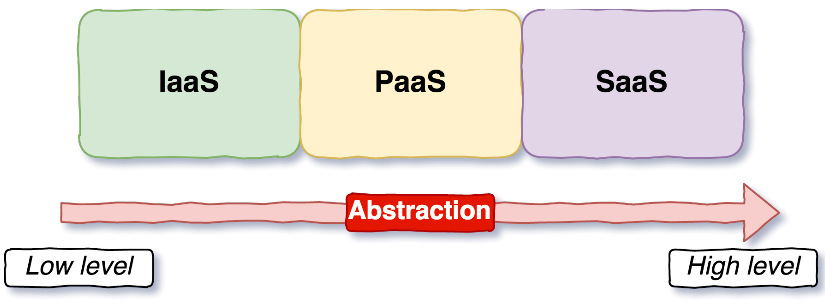


Figure A.1: Cloud service models and abstraction level

In Figure 1.1 there is a detail about the different kinds of management controls according to the type of service provided. Clearly, when the user owns the hardware and there are no services because he has control over the bare metal, the whole stack is completely in the hand of the user. Instead, starting from the IaaS (Infrastructure as a Service), the user loses control over the hardware and become responsible for fewer part of the stack. In this first model, the end-user will interact directly with the machines of the infrastructure (Guest OS) and all that he can put on them.

After, there is the PaaS (Platform as a Service) model, known also as the application category because the providers give as a service a platform allowing customers to develop, run, and manage applications without the complexity of building and maintaining the underlying infrastructure. This will define the runtime environment in which the user has full powers and can develop what he wants.

Last but not least, in the SaaS (Software as a Service) the provider manages all the layers that compose the service and the end-user has to think only to use the software. In this case, the provider gives to the users the direct access to an already deployed application and therefore the actions of the users are strictly linked to the type of application requested.

A.2 Data Analysis pipeline

The typical data analysis pipeline is showed in Figure A.2, where are visible the two main components that precede the analysis: processing and access. In the processing phase, data could be cleaned or can be generated new data useful for the analysis. In the access phase, these data are taken to be processed in an analysis task that produces a result.

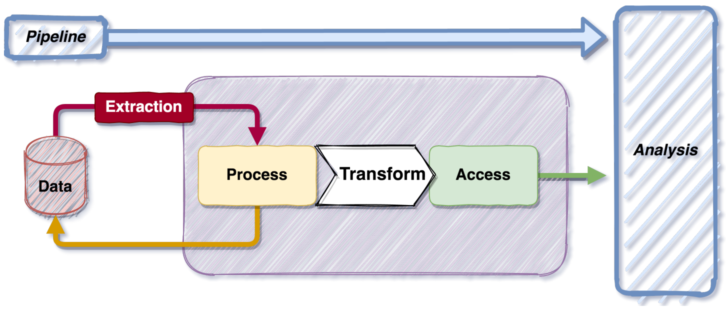
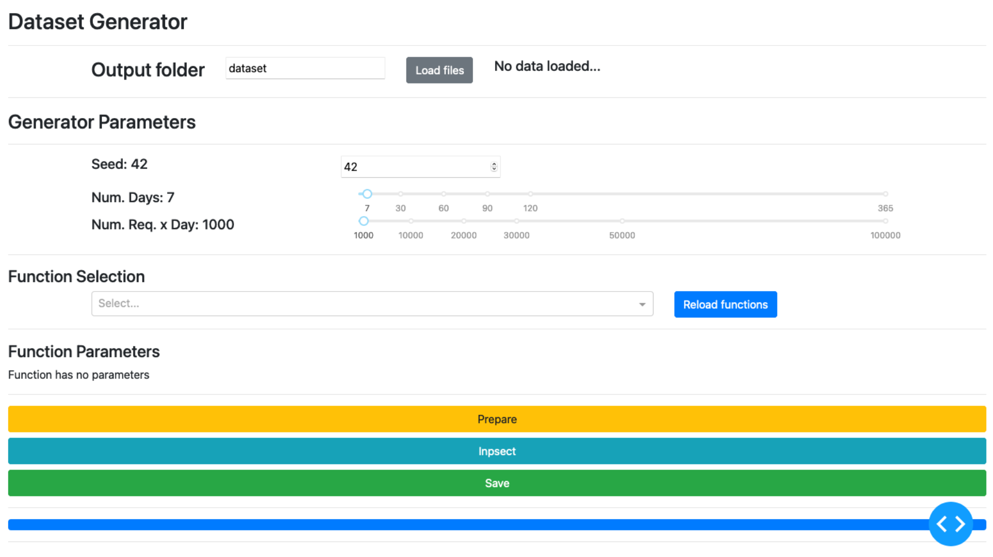


Figure A.2: Typical data analysis pipeline

A.3 Synthetic datasets

Testing and evaluate different methods is not trivial and the data used are very important in the process of the approach validation. For this purpose, a synthetic dataset generator (Figure A.3) was build to make the source for a particular situation. The available recipes to generate source data are meant for stressing a specific pattern and also to check the simulator outputs.


Dataset Generator

Output folder: dataset

No data loaded...

Generator Parameters

Seed: 42

Num. Days: 7

Num. Req. x Day: 1000

Function Selection

Function Parameters

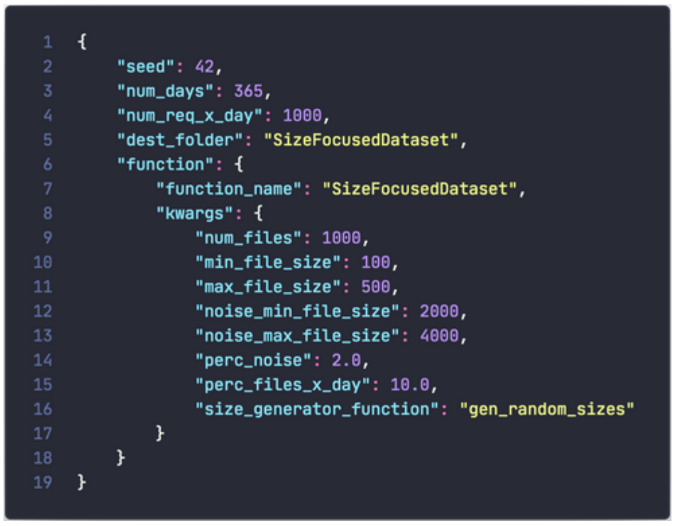
Function has no parameters

Figure A.3: The dataset generator dashboard

The generator tool is written in Python [52] using Plotly [53], and it can be used also from the command line with a JSON [63] configuration file to produce a dataset. An example of the configuration is available in Figure A.4.

Currently, the synthetic dataset generator supports the creation of the following dataset, each one designed to emphasize a situation:

- High-Frequency Dataset: where the requests of a set of files are distributed to make a part of them with a high frequency. It should be the best fit for the LFU policy;
- Recency Focused Dataset: where the requests of a set of files are organized to focus on recency. It should be the best fit for the golden standard LRU;



```json
{
    "seed": 42,
    "num_days": 365,
    "num_req_x_day": 1000,
    "dest_folder": "SizeFocusedDataset",
    "function": {
        "function_name": "SizeFocusedDataset",
        "kwargs": {
            "num_files": 1000,
            "min_file_size": 100,
            "max_file_size": 500,
            "noise_min_file_size": 2000,
            "noise_max_file_size": 4000,
            "perc_noise": 2.0,
            "perc_files_x_day": 10.0,
            "size_generator_function": "gen_random_sizes"
        }
    }
}
```

Figure A.4: Synthetic dataset configuration example in JSON

- Size Focused Dataset: where the most requested files of a set have a specific size. It should be better for policies such as Size Big and Size Small.

As mentioned in the description of each generator, the idea of a stressed situation was something that fits well an already known policy. Despite the generator tool has several views to inspect the created dataset, a further check can be done with the simulator using the policy suggested as the best fit.

A.4 Result view through the simulation analyzer

The simulation analyzer mentioned in Section 4.3 allows to inspect the results with several interactive graphs about the metrics and also the statistics collected during the simulation. Also, it enables to filter the results selecting a specific method or algorithms, and compare them during the period simulated.

In Figure A.5 there is a sample of the comparison between *SCDL*, *SCDL2* and *LRU* regarding the Throughput metric (Equation (6.1)). As stated in Chapter 6, it is notable how the two proposed approaches have a higher throughput compared to the golden standard LRU.

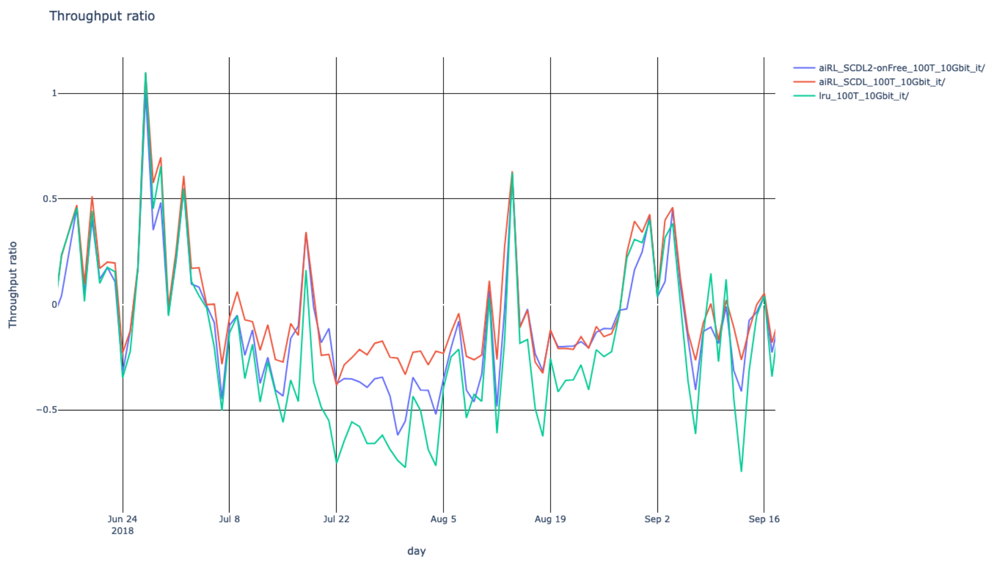


Figure A.5: Throughput comparison using the simulation analyzer

Instead, in Figure A.6 there is another sample but regarding the Cost metric (Equation (6.5)). The results of the experiments (Chapter 6) which attest to the lower cost of the proposed solutions are easily visible in the graph during the evolution of the simulation.

Since there is more information available through the analyzer, it is possible to inspect the different agent behavior as in Figure A.5, where there is a comparison between the *SCDL* and *SCDL2* on their ϵ value during the time. From

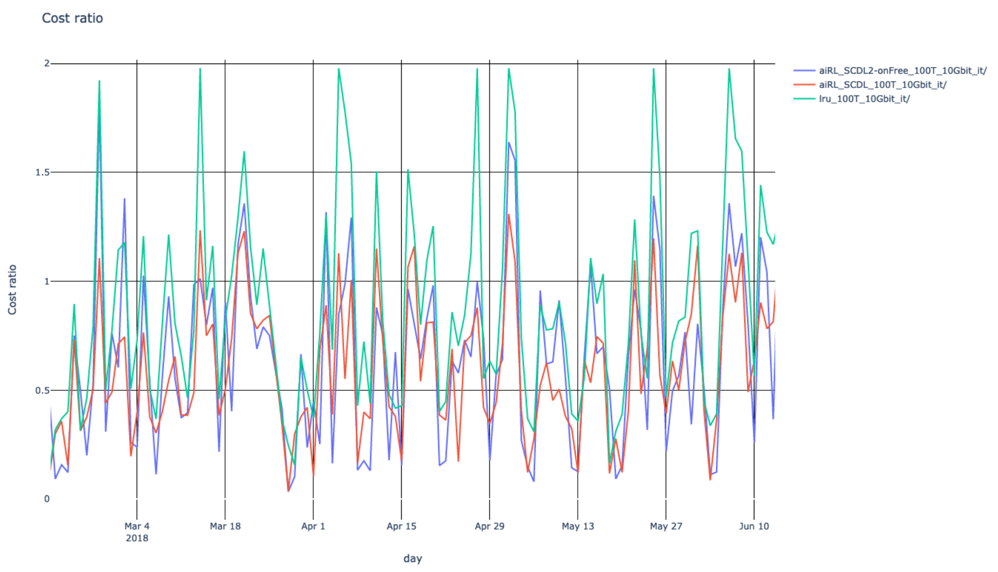


Figure A.6: Cost comparison using the simulation analyzer

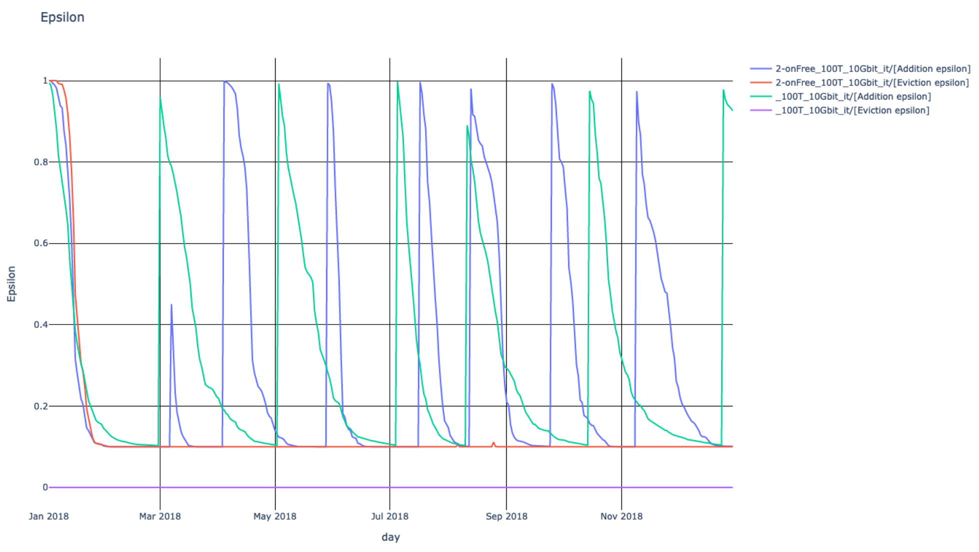


Figure A.7: ϵ decay comparison between SCDL and SCDL2

the graph, it can be inferred that the decay factor chosen allows the agents to learn (or explore) in a period of approximately two weeks. Also, it is visible how the *SCDL* does not have an eviction agent, in fact, its ϵ value is always 0. Moreover, the peaks of the epsilon represent the situation when the agent is decreasing in performance and it requires exploring more than exploiting.