Journal of Grid Computing (2023) 21:42
https://doi.org/10.1007/s10723-023-09664-z

RESEARCH

Smart Caching in a Data Lake for High Energy Physics Analysis

Tommaso Tedeschi · **Marco Baioletti** · **Diego Ciangottini** · **Valentina Poggioni** · **Daniele Spiga** · **Loriano Storchi** · **Mirco Tracolli**

Received: 23 August 2022 / Accepted: 13 April 2023 / Published online: 12 July 2023

Abstract The continuous growth of data production in almost all scientific areas raises new problems in data access and management, especially in a scenario where the end-users, as well as the resources that they can access, are worldwide distributed. This work is focused on the data caching management in a Data Lake infrastructure in the context of the High Energy Physics field. We are proposing an autonomous method, based on Reinforcement Learning techniques, to improve the user experience and to contain the maintenance costs of the infrastructure.

Keywords Reinforcement learning · Caching strategies · High energy physics · Data lake

T. Tedeschi (✉)
Department of Physics and Geology, University of Perugia, Via A. Pascoli, Perugia 06123, Italy
e-mail: tommaso.tedeschi@pg.infn.it

T. Tedeschi · D. Ciangottini · D. Spiga · L. Storchi · M. Tracolli
Sezione di Perugia, INFN, Via Pascoli, Perugia 06123, Italy
D. Ciangottini
e-mail: diego.ciangottini@pg.infn.it

D. Spiga
e-mail: daniele.spiga@pg.infn.it

L. Storchi
e-mail: loriano@storchi.org
https://www.storchi.org/

M. Tracolli
e-mail: m.tracolli@gmail.com

M. Baioletti · V. Poggioni
Department of Mathematics and IT, University of Perugia, Via A. Pascoli, Perugia 06123, Italy
M. Baioletti
e-mail: marco.baioletti@unipg.it

V. Poggioni
e-mail: valentina.poggioni@unipg.it

L. Storchi
Department of Pharmacy, University "G. D'Annunzio" of Chieti-Pescara, Via dei Vestini, Chieti 60111, Italy

1 Introduction

The Large Hadron Collider (LHC) [1] at CERN (the European Organization for Nuclear Research) is the world's largest and most powerful particle accelerator. 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 [2], CMS [3], ALICE [4], and LHCb [5]. A critical challenge at LHC is the next generation of the accelerator expected for 2029, when the named High-Luminosity Large Hadron Collider (HL-LHC) will be fully operative: the upgraded machine will reach an instantaneous luminosity of at least 5×10^{34} cm^{-2}s^{-1} and a center of mass energy of 14 TeV (with respect to 2×10^{34} cm^{-2}s^{-1} and 13.6 TeV of current data-taking period). As a result, data will be produced at higher rates, with a greater event complexity. Consequently, computing resources and storage requests from LHC experiments will increase: as an example, for the disk storage, the future requirements are estimated to ultimately increase by a factor of around 2.5x by the end of 2029 (Fig. 1). With such expectations, in particular considering that the system will start to manage Exabytes (instead of Petabytes) of data, it becomes clear that software and computing of the experiments, as well as the model adopted, must be reviewed and improved through an intensive R&D activity, which represents the key to lower future requests (moving from the solid lines to the dashed ones in Fig. 1, to make sure they stay inside the gray bands).

Recently, many architectural, organizational, and technical changes have been investigated to address the challenge of introducing a new data management model: one of the most promising prototypes is the Worldwide LHC Computing Grid (WLCG) Data Lake model [7,8], a storage service of geographically distributed data centers connected by a low-latency network. In this model, from the infrastructural perspective, caching systems are used to mitigate latency and to serve better popular data.

The goal of the present work is to provide an efficient Reinforcement Learning (RL)-based caching system which could be used by the WLCG collaboration, and more specifically for the CMS experiment [3], which is one of the biggest experiments at CERN and deploys its data collections, simulation, and analysis activities on a distributed computing infrastructure involving more than 60 sites worldwide. Here, with respect to our previous work [9], we consider a new cache size (1000 GiB) and new algorithms: the new SCDL2 and the DQN QCache implemented with different eviction frequencies. The work is organized as follows. In Section 2 a brief description of the Data Lake architecture is given. Then, an introduction to the background

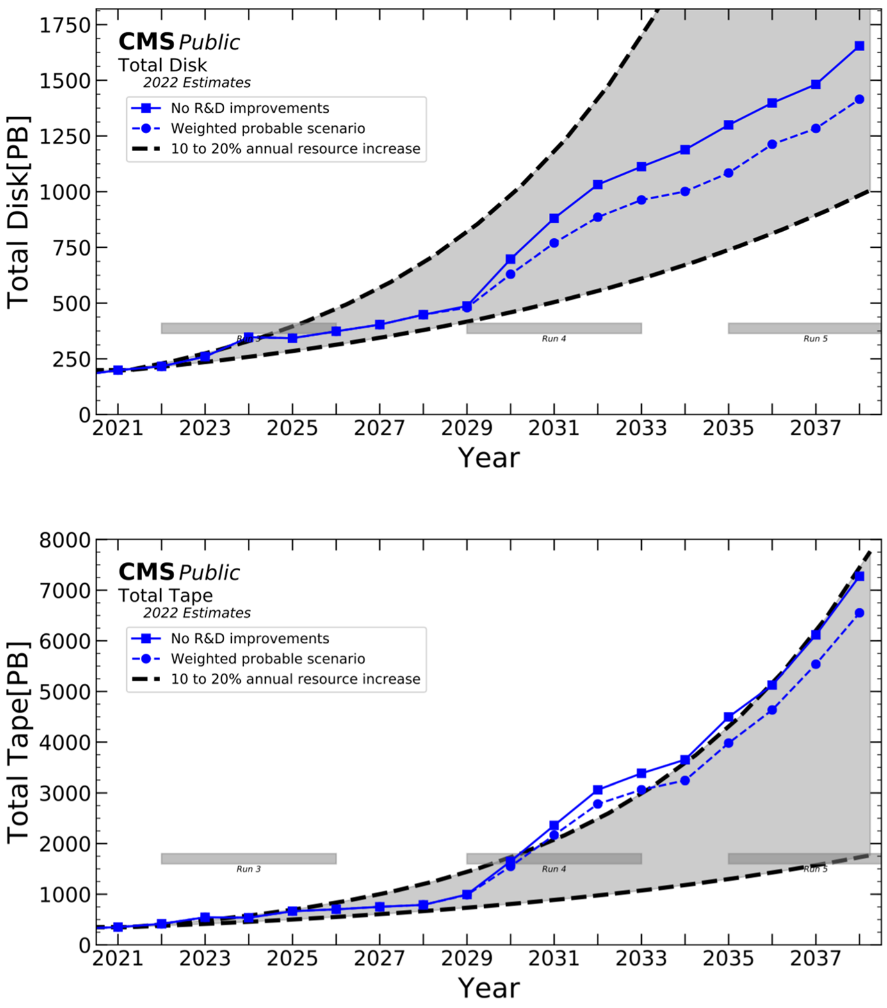

Fig. 1 2022 projections of the increase of data storage at CMS (both for disk and tape), taken from [6]

concepts needed to understand the project is provided in Section 3. In Section 4 the proposed approaches are presented, and comparisons with other solutions are described. Section 5 contains a description of the experimental environment, while the experimental results are described and commented in Section 6. Finally, Section 7 provides some conclusive remarks.

2 Data Lake at WLCG

In the envisioned Data Lake [7,8,10] environment 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.

The environment components included in the Data Lake model are:

- Archive Center (AC): responsible for archive custodial data, the source of information. It should use non-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): it is a center focused on computation with a fast cache to serve the analysis jobs;
- Compute Center with Direct Access (CCDA): a poorer version of a Compute Center having 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.

It is clear that, in this context, the role of the cache becomes the key to effective and efficient data access, while saving the storage needs of the experiment. More specifically, in this work we focus on the CCC component reading from a DCC, evaluating and optimizing the performances of the CCC cache system.

In terms of caching data management our main goal is to solve a problem that has many affinities with a Content Delivery Network, and with the web content caching (especially when video file streaming is considered [11]). However, this project specifically targets the High Energy Physics research community, that needs to optimize the data access in the Data Lake environment while making the system more autonomous to avoid the human intervention as much as possible. For these reasons we chose a RL approach [12], that learns to interact directly with the environment, self-adapting to new situations in the context of Data Lakes, regardless of the data location or the current topology of the network.

To summarize, the model we are considering is made of three basic components: the main storage system (i.e., where the files reside), a cache that serves the requests, and a client that requires the data. The main goal of the caching system is clearly to resolve all the client's requests and serve the files from the cache. This simplified model allows testing different policies to control the request flow.

3 Background on Reinforcement Learning

As previously stated, the approach used in this work is based on RL, which is one of the most important methods in Machine Learning (ML), and aims at training an agent to interact with a particular environment.

RL differs from the other types of ML 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 optimal 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.

One challenging aspect of RL [12–14] algorithm is the trade-off between exploration (i.e., trying new actions) and exploitation (i.e., applying what was learned). The balance between them remains an unresolved problem and one of the most delicate parameters to set.

3.1 Environment Description

The agent trained in RL [15] continuously interacts with the environment as shown in a schematic way in Fig. 2. At each time step, the agent observes the environment, obtaining a state s, and chooses a certain action a to execute, according to a given policy π. As a consequence, it receives a reward (which can be neg-

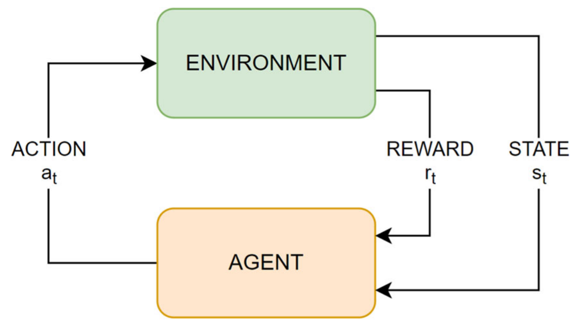

Fig. 2 Reinforcement Learning schema

ative, i.e. a punishment) r from the environment. The ultimate goal is to maximize its cumulative reward, the so-called *return*, hence finding the optimal policy π^*, which maximizes the expected *return* when the agent acts correctly.

The Optimal Action-Value Function $Q^*(s, a)$ is the function that computes the expected *return* if, starting from s, the action a is executed applying the policy π^*. Hence, the optimal action is selected as:

$$a^*(s) = \arg\max_{a} Q^*(s, a) \tag{1}$$

Moreover, the Optimal Action-Value Function $Q^*(s, a)$ obeys to the self-consistency Bellman equation:

$$Q^*(s, a) = \underset{s' \sim P(\cdot \| s,a)}{\mathbb{E}} [r(s, a) + \gamma \max_{a'} Q^*(s', a')] \tag{2}$$

where s' identifies the next state (sampled from the distribution $P(\cdot \| s, a)$ governing all environmental transitions) and $\gamma \in [0, 1]$ is the so-called *discount factor*.

3.2 Q-learning and Deep Q-learning

Q-Learning [16] is one of the best-known RL methods: in its simplest form, the agent tries to learn the $Q^*(s, a)$ function by acting ϵ-greedily, i.e. by selecting a random action a with probability ϵ (that decays over time), otherwise by selecting an action a according to (3).

$$a(s) = \arg\max_{a'} Q(s, a') \tag{3}$$

The first behavior allows the exploration of all possible actions, whereas the second one allows the exploitation of the knowledge gained by the agent.

Learned Q-values are stored in a tabular form for each pair (s, a). The particular Q-value is updated at each step according to (4)

$$Q'(s, a) \longleftarrow Q(s, a) + \alpha(r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)). \tag{4}$$

The memory and computation required for the Q-value algorithm would be too high for real-world scale problems thus, in several applications, Deep Neural Networks (DNNs), which approximate the Q-Learning functions, are used (Deep RL).

In the present work, following the approach proposed by Mnih et al. [17], the Q-value function $Q(s, a)$ is approximated by a DNN, while the objective function is still based on the Bellman Equation in (2). Moreover, an experience replay buffer, as well as a target network, are used to guarantee a stable training. This learning algorithm is called Deep Q-Network (DQN).

4 Algorithms

4.1 Related works

The most used strategy to manage caches is a "Write Everything" approach associated to a Least Recently Used (LRU) or to a Least Frequently Used (LFU) eviction policy [18–20]. It can be effective in most of the cases, but it cannot deal with content popularity and complex network topologies. Hence, recent efforts have gradually shifted toward developing learning and optimization-based approaches, and several ML techniques have been proposed to improve file caching and, in general, better content management.

L. Lei et al. [21] propose to train a DNN in advance to better manage the real-time scheduling of cache content into a heterogeneous network. In [22] a Deep Recurrent Neural Network is applied to predict the cache accesses and to make a better caching decision, but this work has been applied just to cache and synthetic dataset whose sizes are far from the Data Lake volumes. Another example of a prediction approach is presented in [23], where predictions are used to optimize the eviction of a cache with a fixed size. While, in [24], an attempt to automate the caching management of a distributed data cluster using the Gradient Boosting Tree is presented. It is evident that the environment is a critical aspect that has to be taken into account when we talk about caching management and, due to its variability, a more flexible and autonomous solution that can adapt itself is needed. To meet this need, techniques based on RL approach have been recently proposed. In [25] a Deep RL approach is used to cache the highly popular contents across distributed caching entities in the context of CDN. However, even if the system allows an online adaptation, the experiment uses a few files that have to be placed optimally in a hierarchical caching system. There are also Deep RL approaches, like the Wolpertinger architecture [26] used by C. Zhong et al. [27], that try to optimize the cache hit rate. But, in that case, the authors assume that all the files in the cache have the same size, and this is not always the case in High Energy Physics context.

Thus, the problems solved by the cited works are not fully comparable in size and needs with respect to the ones that we are targeting in our project, where there are a much larger number of files to manage and a huge amount of requests per day to satisfy. Moreover, the field of application is different and very specific. The High Energy Physics context has a data access pattern that cannot be always directly compared to other use cases as we are dealing with a heterogeneous community of users chaotically producing files of different size and structure. Furthermore, there is a real necessity to meet the future requirements with the current budget constraints, otherwise the user experience will be drastically compromised.

For these motivations, we consider three different RL-based algorithms (SCDL, the new SCDL2 and DQN QCache) to tackle the cache decisions in terms of file eviction and addition, similarly to what is done in [28] and depicted in Fig. 3. In the first two algorithms a similar mechanism to the one used by the caching system accordingly the WLCG Model (Section 2) is used to prevent the cache memory to become too full or too empty. The mechanism is based on a high (i.e. W_{high}) and a low (i.e. W_{low}) watermark. When the W_{high} memory occupation is reached a file deletion process starts and continues until W_{low} memory occupation is reached. The two watermarks are set to 95% and 75% of the cache size, respectively for W_{high} and W_{low}. The last algorithm, i.e. DQN QCache, only uses the W_{high} watermark.

4.2 SCDL, SDCL2 and DQN Qcache caching algorithms

The Smart Cache for Data Lake (SCDL) algorithm has been the first approach we proposed [29]. It is based on the Q-Learning method and implements only the addition agent. Here we are reporting only the related pseudocode [30] (see SI for extra details).

Algorithm 1 Smart Cache for Data Lake (SCDL) algorithm pseudocode.

```
function SCDL(request)
    file ← request.filename
    update the statistics with request
    hit ← cache_search(file)
    if not hit then
        if random< $\epsilon$ then
            action ←random_action(state)
        else
            action ←best_action(state)
        end if
        if action is Store then
            cache_add(file)
        end if
    end if
    delayed_reward(state)
end function
```

The SCDL2 (Smart Cache for Data Lake 2) [30] we are presenting in the present work is an alternative to the previous approach that uses two different agents to solve the caching problem: the Addition Agent decides whether a requested file has to be stored, while the Eviction Agent chooses how to free the cache memory removing all the files belonging to a specific file category. While the Addition Agent focuses its decision on the state of the request, the eviction agent decisions depend more on the state of the cache memory. As shown in Fig. 3, the goal is to modify the policies used by the cache to decide whether to store a file and what to evict. A pseudocode of this approach is available in Algorithm 2.

When a file f is requested, the Addition Agent is called in order to decide whether to store or not the file f. The Eviction Agent is called only in particular situations: when it is necessary to free space in the cache, at the end of each day, after k iterations (corresponding to the number of requests made to the cache). In those situations, it chooses which files to remove. The two agents work with different state spaces: the Addition Agent's state s is quite similar to the SCDL

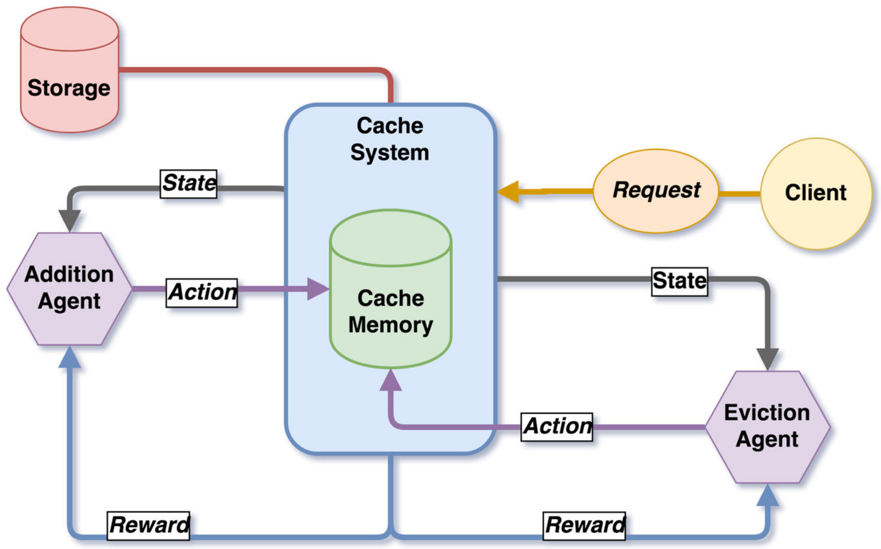

Fig. 3 Reinforcement Learning schema of the double agent approach, where the AI chooses both the addition and eviction of a file into the cache memory

algorithm. In this version, the state is enriched with the cache occupancy percentage oc and cache hit rate hr. As a consequence, states for the Addition Agent are defined as:

$$S_a = b\left(s_f, n_f, \Delta t_f, oc, hr\right) \tag{5}$$

where b is the binning function.

On the other hand, the Eviction Agent composes its state based on the cache memory content. Specifically, the files stored in the cache memory are split into categories subsequently used to choose the set of files to remove. Similarly to the Addition Agent, the Eviction Agent uses $s_f, n_f, \Delta t_f$ to associate the file to a specific category c, that contains all the files of a specific size s_c, that have been requested n_c times and for which Δt_c time has passed since the last request. Moreover, for each category, also the amount of space occupied by the category itself, named oc_c, is considered.

Those features, together with the features characterizing the state of the cache oc and hr, are then discretized in a finite number of buckets, and they result in the following state definition:

$$S_e = b_e(s_c, n_c, \Delta t_c, oc_c, oc, hr) \tag{6}$$

where b_e is the function mapping s_c, n_c, Δt_c, oc_c, oc and hr to the corresponding classes.

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

The action space for the Addition Agent is composed of two possibilities: *Store* and *NotStore*, whereas the action space for the Eviction Agent contains five possibilities: *NotDelete*, *DeleteAll*, *DeleteHalf*, *DeleteQuarter* and *DeleteOne*, that delete respectively no files, all the files, a random half, a random quarter or a single random file belonging to the category. These methods identify how a selected category has to be managed. The choice of considering a finite number of actions for each specific category, instead of having a different delete action for each file stored in the cache, reduces the agent search space.

Since the decision of storing a file f affects the cache composition, and its actual contribution cannot be determined immediately, we decided to use a delayed reward approach. Therefore, after each file request, we store the action chosen by the agent. Then, later in time, the agent will evaluate that decision with a positive or a negative reward depending on specific rules. For the Addition Agent, we assign a positive reward of $r = +1$ to all *Store* actions that allowed a later-requested file to be in memory. The action takes an extra $+1$ if the situation passed from a miss to a hit with that action. Similarly, the agent is penalized with a reward $r = -1$ if the file was not in memory, and it chose the *NotStore* action. In the latter case, if the file passed from hit to miss, there is an extra malus of -1. For the Eviction Agent the rules are very similar, but with the file category as target. In details, a positive reward $r = +1$ is assigned to the action *NotDelete* if the file is found in the cache at the request. Moreover, there is an extra bonus of $+1$ if the cache occupation is not increased in the current request iteration. Conversely, a negative reward $r = -1$ is assigned to the action which deleted the file when a file of a specific category is not found, and an additive malus of -1 is given if the file passed from hit to miss.

To summarize, the environment chooses to penalize those actions that cause the cache to perform more work, such as writing new files and removing files to free space. Thus, the agent tries to avoid non-useful operations, and to minimize the cache actions.

Algorithm 2 Smart Cache for Data Lake 2 (SCDL2) algorithm pseudocode.

```
function SCDL2(request)
    file ← request.filename
    update the statistics with request
    hit ← cache_search(file)
    if not hit then
        if random< $\epsilon$ then
            action←random_action_from_addition_agent(state)
        else
            action←best_action_from_addition_agent(state)
        end if
        if action is Store then
            cache_add(file)
        end if
    end if
    if trigger for eviction agent then
        call_eviction_agent(request)
    end if
    delayed_rewards()
end function
```

Finally there is the DQN QCache approach which is still based on the two agents depicted in Fig. 3.

Algorithm 3 *DQN QCache* algorithm pseudocode.

```
function DQN QCACHE(request)
    file ← request.filename
    update the statistics with request
    hit ← cache_search(file)
    if not hit then
        if random< $\epsilon$ or addition memory size <
        addition agent warm up counter then
            action←random_action_from_addition_agent(state)
        else
            action←best_action_from_addition_agent(state)
        end if
        if action is Store then
            cache_add(file)
        end if
        if addition memory size > addition agent warm up counter then
            batch←sample from cache addition memory
            train addition agent on batch
        end if
    end if
    if trigger for eviction agent then
        for file in cache do
            if random< $\epsilon$ or eviction memory size <
            eviction agent warm up counter then
                action←random_action_from_eviction_agent(state)
            else
                action←best_action_from_eviction_agent(state)
            end if
            if eviction memory size > eviction agent warm up
            counter then
                batch←sample from cache eviction memory
                train eviction agent on batch
            end if
        end for
    end if
    if trigger look for elapsed time windows then
        find_and_reward_elapsed_actions_and_add_to_memory()
    end if
end function
```

5 Experimental Environment

Having introduced the three different caching algorithms we implemented, in the following we present the results and the metrics used to compare the different approaches.

When the cache decides not to store a file, the latter is served in proxy mode, which means that it will fall back on the network. The Fig. 4 shows a schema of the environment and the main statistics collected to evaluate the cache behavior. The data allow us to define three evaluation metrics, that will be detailed in following.

Accordingly to the previous description of the environment, and to the schema represented in Fig. 4, the data read from the storage is split into two sets: *Read on Miss* (i.e., data served in proxy mode because files are not stored in the cache memory), and *Read on Hit* (i.e., data served directly from the cache memory). An ideal cache should be able to keep the *Read on Hit* as high as possible, while maintaining the *Read on Miss* as low as possible, aiming to unload as much as possible the main storage server.

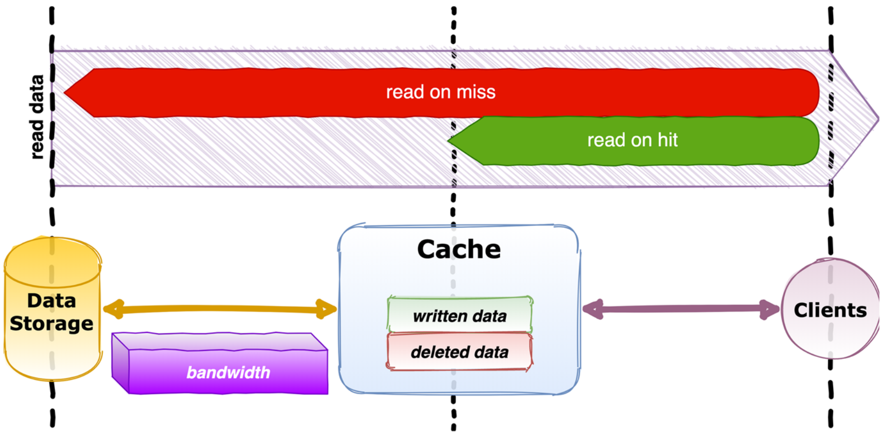

Fig. 4 Simulation environment schema showing the several aspects taken into account and the units measured

In addition, since the simulator is used to stress the cache decisions, in order to simulate the bandwidth limit a simple threshold for daily requests is used, i.e., if the given limit is exceeded the request is processed as a remote call and, consequently, is counted as a miss (a similar mechanism is used in the real-world caching systems where if a cache is overloaded the requests are *redirected* to other caches).

To conclude, there are several parameters to keep under control for the cache content management improvement. It is not trivial to translate the gain obtained by a specific algorithm with respect to the final user experience that is strictly related to data access. Of course, the better the cached content is managed, the greater is expected to be the end-user experience. However, the main goal of the project is to automate and facilitate the management of the cache layer for the system maintainers.

5.1 Dataset

This work uses information on historical user analysis activities at CMS. In order to get a first feedback on the effectiveness of these approaches we tested caches with different sizes using data coming from the real world. A dataset obtained from historical monitoring data of the CMS experiment analysis jobs related to year 2018 [32, 33], filtered for the Italian region, has been used.

To give the reader an overview of the dataset used, in Table 1 we are reporting some overall statistics (see SI for extra plots).

We can clearly notice that the number of tasks, i.e., group of jobs, is two orders of magnitude lower than the number of jobs that can request several files as input. Moreover, given the low standard deviations, we can also assert that the daily number of users and sites (i.e., the place of the request representing a computing center) stay relatively constant over the year. More importantly, the number of requests per file is low on average (i.e., the average number of requests per file is ≈ 6 per file), and the number of files requested per day is comparable to the total number of requests per day (the value is greater than 10^4).

To summarize, there are a lot of requests per day but the majority are unique requests, thus not an easy scenario for a caching system.

5.2 Evaluation metrics

In order to evaluate and compare the different approaches proposed in Section 4, we decide to monitor two main aspects of the cache environment (Fig. 4), the *Throughput* (TP) and the *Cost*. The TP is defined as following:

$$TP = \frac{RHD}{RHD_\infty} \tag{7}$$

where RHD (Read on Hit Data) represents the total amount of data that are read directly from the cache. Since RHD is an absolute quantity that depends on the cache size, we decided to normalize it with respect to the ideal upper bound computed on an infinite cache RHD_∞. In this case, the amount of data that can be read directly from the cache corresponds to the total amount of data that has been written to the cache (i.e., if the cache is infinite we can write any data).

Table 1 Overall dataset statistics

Quantity	Mean	St. Dev
File requests per day	35557	23495
Unique file requested per day	21792	12245
Average number of jobs per task (yearly)	116	493
Unique requesting users per day	46	12
Unique requesting sites per day	6.1	0.7
Average number of requests per file (yearly)	6	46

The *Cost* metric is defined as:

$$Cost = \frac{WD + DD}{2 \cdot WD_\infty} \quad (8)$$

where WD and DD represent the total amounts of written and the deleted data, respectively. They are used to measure how much the cache is working in terms of pure cache operations with respect to WD_∞, the amount of data we can write to an ideal infinite cache.

It is important to note that we cannot evaluate our approach considering the sole *hit rate* (i.e. the standard measure used in cache evaluation) because this measure assumes that all the files have the same size.

An evident desirable outcome is that the *Throughput* is higher than the *Cost*, because the target is to maximize the cache memory content given a small operational cost. Consequently, we decide to use a *Score* measurement defined as follows:

$$Score = \frac{TP}{Cost} \quad (9)$$

This metric penalizes the cache if the amount of data served from the memory is too low with respect to the cache writing and deleting activity.

6 Experimental Results

All the algorithms have been tested with the data described in Section 5.1. Different cache sizes have been simulated: 100 TiB, 200 TiB, 500 TiB, 1000 TiB. ϵ decay rate is set to high values for SCDL and SCDL2, whereas in the DQN QCache is tuned to dedicate the first part of the year to the action-space exploration, and the second part to the exploitation of the gained knowledge.

Moreover, in DQN QCache approach, both DNNs are 2-hidden-layer (using sigmoid activation) feed-forward networks with 2 output nodes (using linear activation), implemented with Adam optimizer (with the Tensorflow's default value of 0.001 as learning rate, [34,35]) and Huber loss function (with Tensorflow's implementation default values, [36]). h_{window} is set to 100000 for Addition agent, and to 200000 for the Eviction agent.

Similarly to what has been done in our previous work [9], we compared the results obtained with the aforementioned algorithms SCDL, SDCL2 (implemented with different eviction approaches: simple LRU, eviction when memory full, eviction at the end of the day, eviction every K requests) and DQN QCache (implemented with different values of eviction frequency), with the results achieved with a "write everything" approach implemented with different eviction algorithms (LRU, LFU, Biggest Files first, Smallest Files first), since the latter are the most used in caching environments. Results are shown in Table 2 and in Fig. 5.

Looking at the results reported in Table 2 we can observe that all the algorithms tested in [9], for caches of size greater than 100 GiB, are outperformed in terms of $Score$ by one of the newly-tested approaches, namely SCDL2 or DQN QCache with eviction frequency different from 50000. More generally, RL methods always show overall better performances, in terms of *Score*, if compared to the standard cache policies, even though the LRU method always reached the best *Throughput* value. This is related to the fact that RL approaches generally make the cache less active by doing the minimum number of operations to maintain a good cache composition: this results in a lower amount of written and deleted files, and therefore in a lower $Cost$, although the presence of missed files may still affect the network. More specifically, the Addition

Table 2 Comparison of results of different algorithms (daily values averaged across the year): SCDL; SCDL2, implemented with different eviction policies: simple LRU (noEviction), eviction when memory full (onFree), eviction at the end of the day (onDayEnd), eviction every K requests (onk) where $K = 8192$; DQN QCache, implemented with different eviction frequencies (indicated as kN, where N is the frequency); Write everything approaches with different eviction policies: Least Recently Used (LRU), Least Frequently Used (LFU), delete biggest files first (Size Big) and delete smallest files first (Size Small). The best result for each metric is displayed in bold

Algorithm - 100 TiB	**Score**	**Throughput**	**Cost**
DQN (k50000)	**0.34**	0.40	**1.19**
SCDL	0.26	0.45	1.74
SCDL2 - noEviction	0.25	0.45	1.82
SCDL2 - onFree	0.22	0.41	1.87
SCDL2 - onDayEnd	0.20	0.39	1.93
Write everything + LRU	0.19	**0.50**	2.66
SCDL2 - onK	0.17	0.34	2.04
Write everything + LFU	0.15	0.43	2.86
Write everything + Size Big	0.12	0.37	3.05
Write everything + Size Small	0.11	0.36	3.09
Algorithm - 200 TiB	**Score**	**Throughput**	**Cost**
DQN (k100000)	**0.41**	0.45	**1.12**
DQN (k50000)	0.35	0.41	1.16
SCDL	0.33	0.55	1.65
SCDL2 - noEviction	0.32	0.54	1.65
SCDL2 - onFree	0.28	0.48	1.73
SCDL2 - onDayEnd	0.25	0.45	1.83
Write everything + LRU	0.24	**0.59**	2.40
Write everything + LFU	0.20	0.52	2.58
SCDL2 - onK	0.17	0.35	2.04
Write everything + Size Big	0.15	0.42	2.89
Write everything + Size Small	0.13	0.39	2.98
Algorithm - 500 TiB	**Score**	**Throughput**	**Cost**
DQN (k250000)	**0.54**	0.53	**0.99**
SCDL2 - noEviction	0.53	0.69	1.30
SCDL	0.51	0.72	1.41
SCDL2 - onFree	0.39	0.60	1.55
Write everything + LRU	0.39	**0.74**	1.90
Write everything + LFU	0.32	0.67	2.11
DQN (k50000)	0.31	0.47	1.51
SCDL2 - onDayEnd	0.28	0.49	1.75
Write everything + Size Big	0.22	0.54	2.52
Write everything + Size Small	0.18	0.48	2.70
SCDL2 - onk	0.17	0.36	2.04
Algorithm - 1000 TiB	**Score**	**Throughput**	**Cost**
SCDL2 - noEviction	**0.78**	0.80	1.01
SCDL	0.71	0.83	1.17
DQN (k500000)	0.69	0.60	**0.86**

Table 2 continued

Write everything + LRU	0.59	**0.86**	1.45
SCDL2 - onFree	0.55	0.70	1.27
Write everything + LFU	0.48	0.80	1.65
DQN (k50000)	0.34	0.41	1.21
Write everything + Size Big	0.33	0.68	2.07
SCDL2 - onDayEnd	0.30	0.53	1.78
Write everything + Size Small	0.26	0.59	2.32
SCDL2 - onk	0.18	0.36	2.04

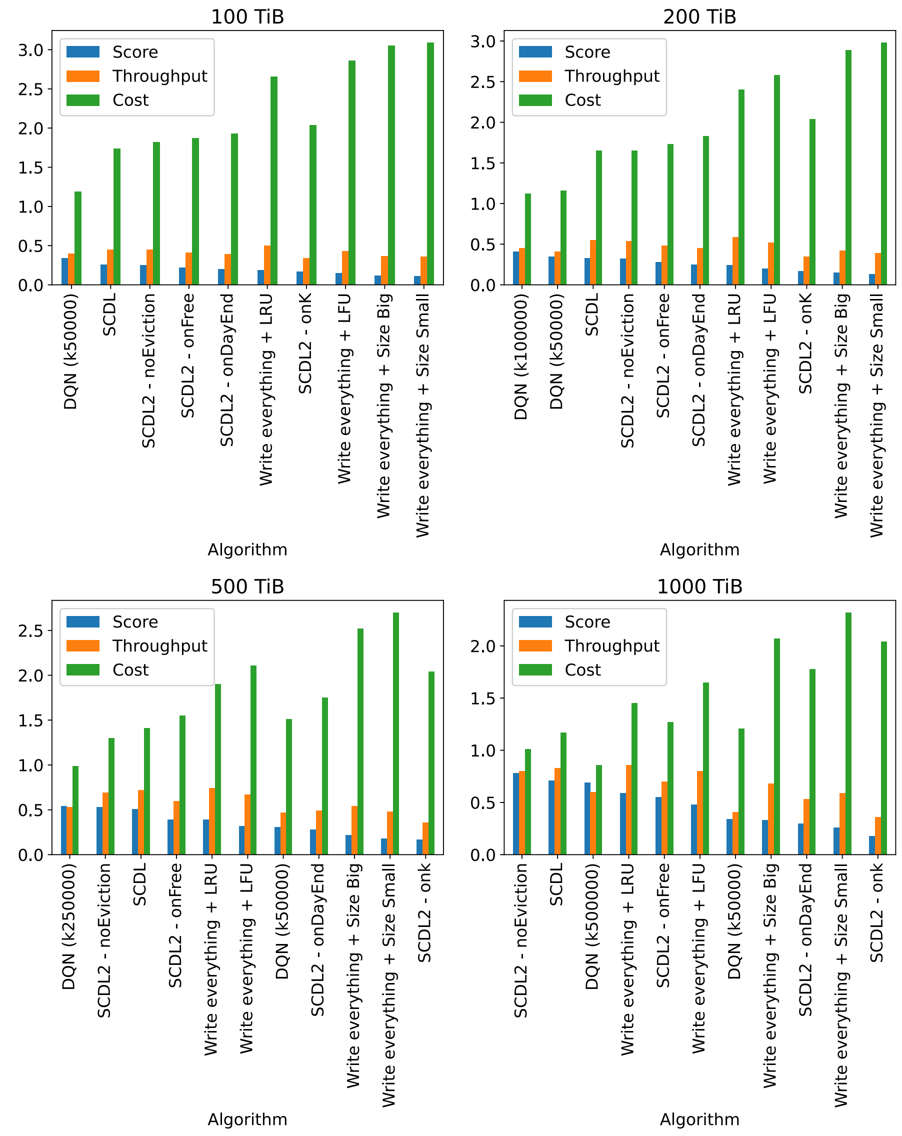

Fig. 5 Histograms reporting all the metrics for each cache size and each algorithm, see Table 2

Agent is the main responsible for reducing the amount of written data and selecting files to store in a more rigorous way, contributing to the *Throughput*. The Eviction Agent affects the presence of the files in the cache, evicting those which are expected not to be requested anymore, contibuting to the overall decrease of the *Cost* of the system. Furthermore, we can attest that the different mechanisms used to free the cache memory content have a deeper impact on the general caching performances with respect to the simple file-filtering. In the case of DQN QCache, these results show that the eviction frequency plays a key role, since DQN QCache with different values of k perform differently in terms of $Cost$, and therefore in terms of $Score$. Indeed, to maintain the $Cost$ at lower values, as the cache size increases, the eviction frequency should be increased accordingly.

Finally, and most importantly, it is crucial to underline a key aspect: in the present simulation each delete or write operation is considered to be timeless. Reason why, we are expecting that, in a real-world scenario, a cache system that is less busy in writing and removing files will be surely readier to distribute the requested files to the clients, i.e, it will be more efficient and will provide a final better use experience. Indeed the RL approaches are always the top ranked with respect to the *Score*, as clear evidence of this fact.

7 Conclusions

Recently the CMS community at CERN has started to experiment with new models to manage the whole computing infrastructure due to upcoming updates and the huge amount of data foreseen for the next years, exploring the possibility of moving towards a Data Lake model. This new scenario imposes to find more effective solutions to the data caching problem. Thus, the role of the cache becomes a key to effective and efficient data access.

In this work we extended our previous results [9] introducing additional RL-based approaches and an additional simulated cache size: results show that the newly-introduced algorithms enhance the performance gain related to the usage of RL with respect to the standard caching policies.

More generally, we can conclude that the RL caching algorithms we implemented showed better overall performances in terms of *Score*, and especially in terms of *Cost*, with respect to the standard policies using, for example, LRU eviction strategy. Our RL approaches make the cache less active by doing a lower number of operations to maintain a good cache composition. This results in a lower amount of written and deleted data. While the presence of missed files still affects the network, we want to underline that we are expecting that in a real-world scenario (where the time domain is taken into account), a cache system that is less busy in writing and removing files will be surely more responsive and quicker to serve the requested files to the clients.

Acknowledgements The authors thank the CMS collaboration, and in particular the Machine Learning and Offline Software and Computing groups for the valuable discussions that helped the development of this work.

Author contributions T.T. M.T. L.S. D.S. wrote the manuscript text and prepared figures. All authors reviewed the manuscript.

Funding This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors. Open access funding provided by Universitá degli Studi di Perugia within the CRUI-CARE Agreement.

Compliance with ethical standards

The authors declare no potential conflicts of interest.

Conflict of interest The authors declare no competing interests.

References

1. Pettersson, T.S., Lefèvre, P.: The Large Hadron Collider: conceptual design. Technical report (Oct 1995). https://cds.cern.ch/record/291782
2. The ATLAS Collaboration: The ATLAS experiment at the CERN Large Hadron Collider. J. Instrum. **3**, 08003 (2008)
3. The CMS Collaboration: The CMS experiment at the CERN LHC. J. Instrum. **3**(08), 08004–08004 (2008)

4. The ALICE Collaboration: The ALICE experiment at the CERN LHC. J. Instrum. **3**(08), 08002 (2008)
5. The LHCb Collaboration: The LHCb detector at the LHC. J. instrum. **3**(08), 08005 (2008)
6. CMS Offline Software and Computing: CMS Phase-2 Computing Model: Update Document. CERN-CMS-NOTE-2022-008, available on the CERN Document Server as https://cds.cern.ch/record/2815292. (2022)
7. Bird, I., Campana, S., Girone, M., Espinal, X., McCance, G., Schovancová, J.: Architecture and prototype of a WLCG data lake for HL-LHC. In: EPJ Web of Conferences, vol. 214, p. 04024 (2019). EDP Sciences
8. Kadochnikov, I., Bird, I., McCance, G., Schovancova, J., Girone, M., Campana, S., Currul, X.E.: WLCG data lake prototype for HL-LHC. Advisory committee, 127 (2018)
9. Tedeschi, T., Tracolli, M., Ciangottini, D., Spiga, D., Storchi, L., Baioletti, M., Poggioni, V.: Reinforcement Learning for Smart Caching at the CMS experiment. In: Proceedings of International Symposium on Grids & Clouds 2021 PoS(ISGC2021), vol. 378, p. 009 (2021)
10. Dixon, J.: Pentaho, Hadoop and Data Lakes. https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/. Last check April 9, 2020 (2010)
11. Adhikari, V.K., Guo, Y., Hao, F., Varvello, M., Hilt, V., Steiner, M., Zhang, Z.-L.: Unreeling netflix: Understanding and improving multi-CDN movie delivery. In: 2012 Proceedings IEEE INFOCOM, pp. 1620–1628 (2012). IEEE
12. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT press, Cambridge (2018)
13. Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: a survey. J. Artif. Intel. Res. **4**, 237–285 (1996)
14. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing atari with deep reinforcement learning. arXiv:1312.5602 (2013)
15. Wiering, M.A., Van Otterlo, M.: Reinforcement learning. Adapt. Learn. Optim. **12**(3), 729 (2012)
16. Watkins, C.J., Dayan, P.: Q-learning. Mach. Learn. **8**, 279–292 (1992)
17. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature. **518**(7540), 529–533 (2015)
18. Zhang, M., Luo, H., Zhang, H.: A survey of caching mechanisms in information-centric networking. IEEE Commun. Surv. Tutor. **17**(3), 1473–1499 (2015)
19. Podlipnig, S., Böszörmenyi, L.: A survey of web cache replacement strategies. ACM Comput. Surv. (CSUR) **35**(4), 374–398 (2003)
20. Chen, C., Wang, C., Qiu, T., Atiquzzaman, M., Wu, D.O.: Caching in vehicular named data networking: Architecture, schemes and future directions. IEEE Commun. Surv. Tutor. **22**(4), 2378–2407 (2020)
21. Lei, L., You, L., Dai, G., Vu, T.X., Yuan, D., Chatzinotas, S.: A deep learning approach for optimizing content delivering in cache-enabled hetnet. In: 2017 International Symposium on Wireless Communication Systems (ISWCS), IEEE, pp. 449–453 (2017)
22. Narayanan, A., Verma, S., Ramadan, E., Babaie, P., Zhang, Z.-L.: Deepcache: A deep learning based framework for content caching. In: Proceedings of the 2018 Workshop on Network Meets AI & ML, pp. 48–53 (2018)
23. Lykouris, T., Vassilvitskii, S.: Competitive caching with machine learned advice. arXiv:1802.05399 (2018)
24. Herodotou, H.: Autocache: Employing machine learning to automate caching in distributed file systems. International Conference on Data Engineering Workshops (ICDEW), IEEE, pp. 133–139 (2019)
25. Sadeghi, A., Wang, G., Giannakis, G.B.: Deep reinforcement learning for adaptive caching in hierarchical content delivery networks. IEEE Trans. Cogn. Commun. Netw. **5**(4), 1024–1033 (2019)
26. Dulac-Arnold, G., Evans, R., van Hasselt, H., Sunehag, P., Lillicrap, T., Hunt, J., Mann, T., Weber, T., Degris, T., Coppin, B.: Deep reinforcement learning in large discrete action spaces. arXiv:1512.07679 (2015)
27. Zhong, C., Gursoy, M.C., Velipasalar, S.: A deep reinforcement learning-based framework for content caching. In: 2018 52nd Annual Conference on Information Sciences and Systems (CISS), IEEE, pp. 1–6 (2018)
28. Alabed, S.: RLCache: automated cache management using reinforcement learning. arXiv:1909.13839. (2019)
29. Tracolli, M., Baioletti, M., Ciangottini, D., Poggioni, V., Spiga, D.: An intelligent cache management for data analysis at cms. In: International conference on computational science and its applications, Springer, pp. 320–332 (2020)
30. Tracolli, M.: Open Source code. Available at https://github.com/Cloud-PG/smart-cache/tree/master (2022)
31. Tedeschi, T.: Open Source code. Available at https://github.com/Cloud-PG/smart-cache/tree/dQl_add_evic_no_gym (2022)
32. Kuznetsov, V., Li, T., Giommi, L., Bonacorsi, D., Wildish, T.: Predicting dataset popularity for the CMS experiment. arXiv:1602.07226arXiv:1602.07226. (2016)
33. Meoni, M., Perego, R., Tonellotto, N.: Dataset popularity prediction for caching of CMS big data. J. Grid Comput. **16**(2), 211–228 (2018)
34. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: large-Scale machine learning on heterogeneous systems. Softw. available from tensorflow.org. (2015). https://www.tensorflow.org/
35. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. arXiv:1412.6980. (2014). https://doi.org/10.48550/ARXIV.1412.6980
36. Huber, P.J.: Robust Estimation of a Location Parameter. Ann. Math. Stat. **35**(1), 73–101 (1964). https://doi.org/10.1214/aoms/1177703732