Reinforcement Learning for Smart Caching at the CMS experiment

Tommaso Tedeschi[*†]
Università degli Studi di Perugia and INFN Sezione di Perugia, Perugia, Italy
E-mail: tommaso.tedeschi@pg.infn.it

Mirco Tracolli
Università degli Studi di Perugia and INFN Sezione di Perugia, Perugia, Italy
Università degli Studi di Firenze, Florence, Italy
E-mail: mirco.tracolli@pg.infn.it

Diego Ciangottini
INFN Sezione di Perugia, Perugia, Italy
E-mail: diego.ciangottini@pg.infn.it

Daniele Spiga
Università degli Studi di Perugia and INFN Sezione di Perugia, Perugia, Italy
E-mail: daniele.spiga@pg.infn.it

Loriano Storchi
INFN Sezione di Perugia, Perugia, Italy
E-mail: loriano.storchi@pg.infn.it

Marco Baioletti
Università degli Studi di Perugia, Perugia, Italy
E-mail: marco.baioletti@unipg.it

Valentina Poggioni
Università degli Studi di Perugia, Perugia, Italy
E-mail: valentina.poggioni@unipg.it

In the near future, High Energy Physics experiments' storage and computing needs will go far above what can be achieved by only scaling current computing models or current infrastructures. Considering the LHC case, for 10 years a federated infrastructure (Worldwide LHC Computing Grid, WLCG) has been successfully developed. Nevertheless, the High Luminosity (HL-LHC) scenario is forcing the WLCG community to dig for innovative solutions. In this landscape, one of the initiatives is the exploitation of Data Lakes as a solution to improve the Data and Storage management. The current Data Lake model foresees data caching to play a central role as a technical solution to reduce the impact of latency and network load. Moreover, even higher efficiency can be achieved through a smart caching algorithm: this motivates the development of an AI-based approach to the caching problem. In this work, a Reinforcement Learning-based cache model (named QCACHE) is applied in the CMS experiment context. More specifically, we focused our attention on the optimization of both cache performances and cache management costs. The QCACHE system is based on two distinct Q-Learning (or Deep Q-Learning) agents seeking to find the best action to take given the current state. More explicitly, they try to learn a policy that maximizes the total reward (i.e. hit or miss occurring in a given time span). While the addition Agent is taking care of all the cache writing requests, clearly the eviction agent deals with the decision to keep or to delete files in the cache. We will present an overview of the QCACHE framework an the results in terms of cache performances, obtained using using "Real-world" data, will be compared respect to standard replacement policies (i.e. we used historical data requests aggregation used to predict dataset popularity filtered for Italian region). Moreover, we will show the planned subsequent evolution of the framework.

International Symposium on Grids & Clouds 2021, ISGC 2021
22-26 March, 2021
Academia Sinica, Taipei, Taiwan (online)

[*]Speaker.
[†]On behalf of the CMS collaboration

1. Introduction

The LHC (Large Hadron Collider, [1]) is going to face big challenges in the near future. In fact, approximately in 2026, with the next planned upgrade, it will enter in the High Luminosity era: the upgraded machine (High Luminosity LHC, HL-LHC) 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 5×10^{34} cm^{-2}s^{-1} and 14 TeV of last 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, getting far above what can be achieved by only scaling today's technologies: as can be seen in fig. 1 and fig. 2, looking at the CMS experiment [2] projections for year 2030, the gap between CPU and disk requirements and actual availability, assuming a flat budget, will be approximately of a factor 3. Such a scenario is clearly demanding a new computational model that should optimize both hardware and operational cost.

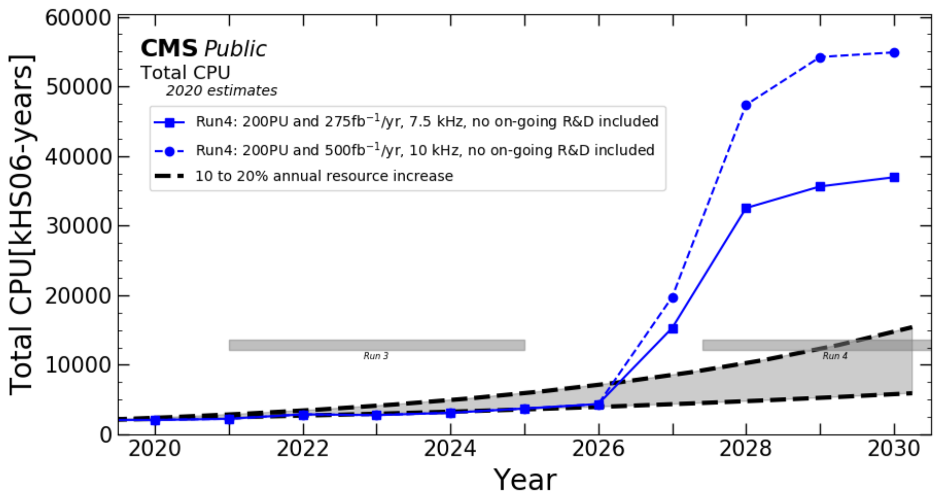

Figure 1: CPU time requirements (in kilo-HEPSpec06 years) estimated to be required annually for CMS processing and analysis needs. These results are taken from CMS Offline and Computing Public Results [3].

At the moment, the community is strongly focusing on the storage problem, mainly due to cost reasons and the lack of a straightforward solution. Aside from the introduction of a new reduced data format (e.g., nanoAOD for the CMS experiment [4]), the storage problem can be addressed with the introduction of a new distributed model: the proposed solution is a Data Lake model [5]. It envisions fewer number of storage endpoints with respect to the current tiered configuration, with a mix of distributed caches directly accessed from the various compute nodes. In this design, network and caches are crucial for the development of an optimized system and indeed a lot of effort has been carried out in the last years on both topics (a non-comprehensive list of projects that are working on this includes WLCG DOMA [6], EU projects like ESCAPE [7], US CMS Data lake proposal [8], SoCal Cache [9], INFN distributed cache [10], the IDDLS initiative [11]).

CMS already extensively tested the benefits deriving from the use of a cache system [12]. In

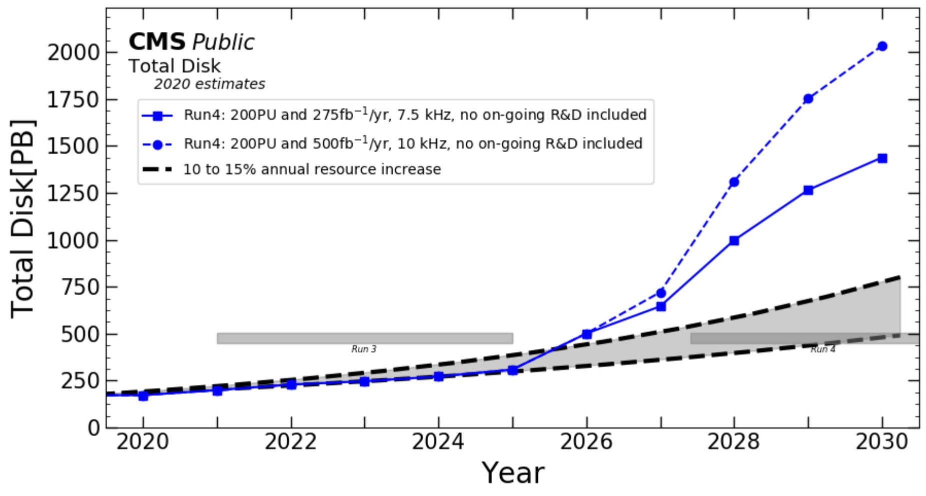

Figure 2: Disk space requirements (in PBs) estimated to be required annually for CMS processing and analysis needs. These results are taken from CMS Offline and Computing Public Results [3].

particular it has been shown that cache systems allow to reduce the overall network traffic, thus making the processing jobs more efficient by reducing I/O wait time for remote data. Similarly a cache system is capable of providing read ahead capability and to save disk space. A local cache is also seen as a key component of the future CMS Analysis Facility that is being prototyped at INFN.

Once that the advantages related to caches are evident in the CMS experiment context, the main goal is to enhance the efficiency of the cache system with a smart caching approach. Thus, the subject of Smart Caching project [12] is the development of such a smart cache system using AI algorithms within the CMS experiment. The main idea is to develop an AI system that directly manipulates the cache file content, deciding what to write or delete. The ultimate goal is to obtain an algorithm that uses fewer storage resources with respect to classic cache algorithms (write everything + LRU, LFU, etc..., [13]) while maintaining similar performance. In order to do this, we developed the QCACHE framework, which is based on Reinforcement Learning methods.

Finally, it is worth to notice that the impact of this work could be even broader, since many communities beyond WLCG are facing the same problems in data distribution (e.g. Astroparticle Physics [14]) and could greatly benefit from an efficient data caching approach, too.

2. Methodology

In the present section we will introduce the main idea beyond the development of an AI system that directly manipulates the cache file content, as reported in fig. 3. In the last decades, many studies regarding the optimization of cache systems have been published (e.g. [15–18]), but none of them focused on a Data Lake architecture. The ultimate goal of the Smart Caching project, as already partially discussed, is to obtain a framework (named QCACHE) that while using fewer resources with respect to classic cache algorithms should be able to guarantee similar performances.

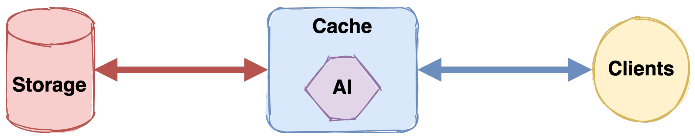

Figure 3: A schematic view of Smart Caching approach: AI directly manipulates cache memory.

2.1 Reinforcement Learning

Reinforcement Learning [19] (RL) is the branch of Machine Learning that studies agents that learn through an iterative trial and error process while interacting within an environment. At each step, a RL agent receives as input from the environment a state s (or a partial observation o) and chooses a certain action a, getting a reward (or punishment) r from the environment, which is used to update the agent itself (see fig. 4). The ultimate goal is to maximize its cumulative reward (the so-called *return*).

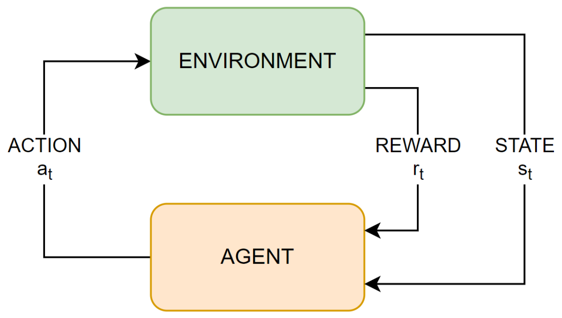

Figure 4: A Reinforcement Learning step.

More precisely, a RL agent learns a policy π, that is the function used by the agent to choose which action to take given a certain state or observation and could be either stochastic or deterministic. Seen in these terms, the aim of a RL agent becomes finding optimal policy π^*, which maximizes the expected return when the agent acts accordingly.

You can then define the Optimal Action-Value Function $Q^*(s,a)$ as the function that gives the expected return if, starting from s, you take an arbitrary action a and from then on you act accordingly to the optimal policy. The optimal action is then trivially defined as follows:

$$a^*(s) = \arg\max_a Q^*(s,a) \tag{2.1}$$

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

$$Q^*(s,a) = \mathop{\mathbb{E}}_{s' \sim P(\cdot|s,a)} [r(s,a) + \gamma \max_{a'} Q^*(s',a')] \tag{2.2}$$

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

2.1.1 Q-learning

Q-Learning is one of the best known sets of RL methods: in its simplest form, an ε-greedy Q-learning agent tries to learn the $Q^*(s,a)$ function by acting ε-greedily, i.e. selecting a random action a with probability ε (that decays over time), otherwise selecting action a according to eq. (2.3).

$$a(s) = \arg\max_{a'} Q(s,a') \tag{2.3}$$

The first behavior is related to the exploration of all possible actions, whereas the second one is related to the exploitation of the knowledge gained by the agent. Learned Q-values are stored in a tabular form for each (s,a) pair, and an update is performed at each step accordingly to eq. (2.4) which is based on the Bellman Equation (see eq. (2.2)).

$$Q^{new}(s,a) \longleftarrow Q(s,a) + \alpha(r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)) \tag{2.4}$$

The Deep Learning extension of RL is called Deep Reinforcement Learning (Deep RL), where RL functions are approximated by Deep Neural Networks (DNNs). The most classic and fundamental Deep RL extension of Q-learning is the DQN algorithm, which was firstly introduced in 2015 [20].

In the present work, following the approach proposed by Mnih et al. [20], the Q-value function $Q(s,a)$ is approximated by a DNN, while the objective function is still based on the Bellman Equation (eq. (2.2)), and an experience replay buffer as well as a target network are used to guarantee a stable training.

2.2 QCACHE

As a solution to the Smart Caching problem, we developed the RL-based QCACHE framework. In its general form it is based on two distinct Q-Learning (or Deep Q-Learning) agents, as reported in fig. 5, that take care of two different fundamental aspects, that is writing and removing files:

- *Addition agent*: chooses whether to write or not a requested file
- *Eviction agent*: chooses whether to remove or not a cached file

The first implementation of QCACHE (SCDL QCACHE) has already been shown in a related work [21] and it is based on a simple Q-Learning addition agent associated to LRU policy for eviction. In this work, the approach is extended to the Deep RL world with the creation of the DQN QCACHE, which exploits the DQN algorithm for both agents and will be described in detail in the next sections, along with the SCDL approach.

2.2.1 SCDL QCACHE

The SCDL QCACHE [21] approach relies on the Q-Learning technique, since an ε-greedy Q-Learning agent is used for the addition task, while files are evicted via the LRU policy. Each file request corresponds to a specific input state s for the addition agent, and it is composed with the basic information (features) taken from the file f statistics collected during the environment

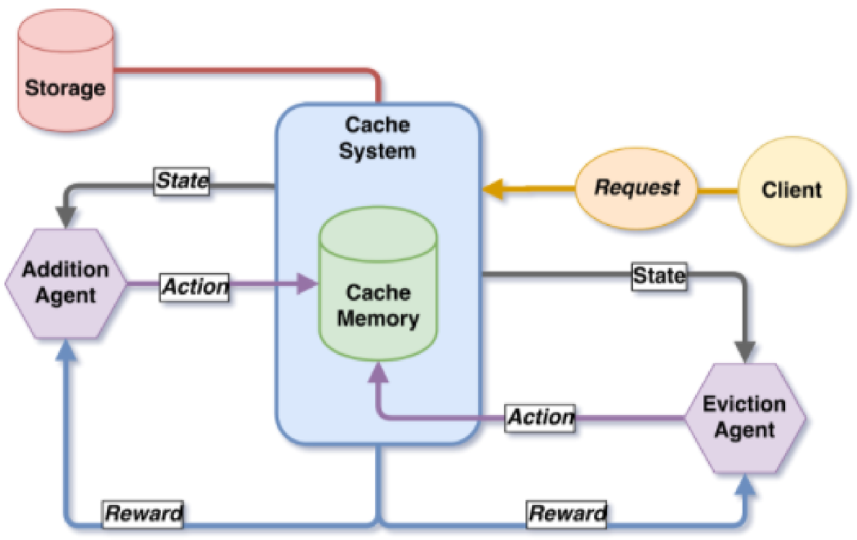

Figure 5: A schematic view of the QCACHE framework.

lifetime, enriched with information about the cache status. The features we are considering are: the file size, the frequency, the delta time since the last request of the file, cache occupancy percentage and cache hit rate. Moreover, since the same input state is considered as the next state, the agent learns which action is the best for the current state in a way similar to that of a Contextual Bandit approach but using the delayed rewards as stimuli of past decision traces. Of course, since the number of states must be finite because the agents use a Q-Table to store Q-Values for each state-action pair, the input features are discretized in a finite number of classes (using a simple binning technique with ranges).

The system also takes into account the two cache watermarks (a high watermark W_{high} and a low watermark W_{low}) which are set accordingly to the amount of available space: 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.

As for the rewards, the SCDL algorithm assigns a positive or negative reward equal to the size of the file requested according to the increase or decrease of the data read from the cache memory.

At the very beginning of the cache lifetime, the algorithm sets the value of ε to 1.0, which then decays exponentially over time to a lower value of 0.1 while still being able to rise again thanks to an unleash mechanism: when the agent is losing in performance (i.e. if the total reward score is decreasing for 8 days in a row) the ε is reset to 1.0. The aim of this strategy is to make the agents able to adapt automatically to new request flow situations. For the same reason, the learning rate α is set to 0.9 and the discount factor γ is equal to 0.5, because we want to appreciate the request changing patterns but without lost completely the past made choices.

2.2.2 DQN QCACHE

In the DQN QCACHE framework, both addition and eviction agents are implemented as DQN agents, and each file request or each cached file, respectively, corresponds to a specific input state s. When a file request comes in, the addition agent chooses whether that file has to be cached or not and every k requests, or if a high watermark W_{high} is reached, eviction agent is iteratively applied on every cached file choosing which one has to be evicted or kept.

The reward-giving process is based on the number of times a certain file is requested by users after a choice has been made about it by one of the agents. That is, if we define as hit a request of a file that is already present in cache, and as miss a request of a file that is not present in cache, the reward is computed for both agents observing h_{window} requests after the action, and considering the *size* of that file. More specifically:

- if the original action was "$keep$-like" (i.e. $store$ for the addition agent, $notevict$ for the eviction agent), the reward is:

$$r = \begin{cases} n_{hit} \cdot size, & \text{if } n_{hit} > 0 \\ -size, & \text{otherwise} \end{cases} \tag{2.5}$$

  where n_{hit} is the number of hits for that file in the next h_{window} requests, and *size* is the size of the file;

- if the original action was "$notkeep$-like" (i.e. $notstore$ for the addition agent, *evict* for the eviction agent), the reward is:

$$r = \begin{cases} -n_{miss} \cdot size, & \text{if } n_{miss} > 0 \\ size, & \text{otherwise} \end{cases} \tag{2.6}$$

  where n_{miss} is the number of misses for that file in the next h_{window} requests, and *size* is the size of the file.

The state s_t input variables that are fed to both agents are only related to file statistics and cache status, that is: file size, file frequency, the delta time since the last request of the file and its datatype (e.g. data or Monte Carlo), cache occupancy percentage and cache hit rate. Every N requests (N is equal to 30000 in the present study), the algorithm looks for actions for which the h_{window} time is elapsed: when a "terminated window" is found, the next state s_{t+1} is defined (same size of previous state s_t, frequency increased by one, same delta time, same datatype and current cache occupancy and hit rate), the reward r_t is computed and the 4-tuple (s_t, a_t, r_t, s_{t+1}) is stored in the agent's experience replay memory.

The decay of ε is exponential, starting from a value of 1 until reaching 0.1 as a lower limit, ensuring an everlasting exploration component. DQN experience replay memory size is set to 1 million, whereas the discount factor γ value is 0.5. The target models are updated every 10000 iteration steps for both agents. h_{window} is set to 100000 requests for addition and to 200000 for eviction, and the eviction k is set to 50000 requests.

Moreover, a relatively small DNN is used (only two hidden layers with low number of neurons) for both agents. All implementation hyperparameters are shown in table 1.

Parameter	**Value**
Hidden layers	2 (16 and 32 neurons, sigmoid activation)
Output layer	2 neurons, linear activation
Optimizer	Adam (lr = 0.001)
Loss function	Huber (δ = 1)
Weights initialization	Glorot Uniform
Batch size	32

Table 1: DQN QCACHE DNN parameters, valid for both agents.

2.3 Dataset and simulation

As previously stated, in order to get a first feedback on the effectiveness of these approaches (before moving to a real testbed), we simulated caches with different sizes using data coming from the real world. Thus, we used a dataset obtained from historical monitoring data about CMS experiment analysis jobs in 2018 [22, 23] filtered for the Italian region: these data are used to simulate daily file requests flow for all 2018. In order to give some hints about the complexity of the problem, fig. 6 displays two plots showing the daily number of files and requests and the daily average number of requests per file, respectively, as functions of the day of the year: it is clear that, on average, the number of requests per file is low, thus resulting in a high variability.

Different cache sizes have been simulated: 100TiB, 200TiB, 500TiB and a bandwidth limit is considered and set as a daily limit to 103 TiB. When the bandwidth limit is reached, the request is processed as a remote call and is counted as a miss.

Besides, we use a W_{high} equal to 95% of the cache size and a W_{low} of 75% (only for SCDL QCACHE and classic cache algorithms).

In the DQN QCACHE case, the ε decay rate is set in order to make sure that the first part of the year is dominated by exploration (with higher ε values), while the second part is the one where the agent tends to exploit the gained knowledge (with lower ε values). This behavior is shown in fig. 7 that displays the daily mean value of ε as a function of the day of the year, plotted for both the addition agent and the eviction agent, in DQN QCACHE 100TiB simulation. For SCDL QCACHE, ε decay rate is much higher, but the unleash mechanism ensures a fast response to environment changes.

2.4 Performance evaluation

In order to evaluate and compare the performance of our solution to those of classic caching algorithm, we defined a series of quality metrics. In particular, defining $readOnHitData$ as the incremental sum of the size of data read from the cache (thus associated to hits), and $writtenData$ and $deletedData$ as the incremental sum of the data size written and deleted by the cache respectively, the cache quality metrics are:

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

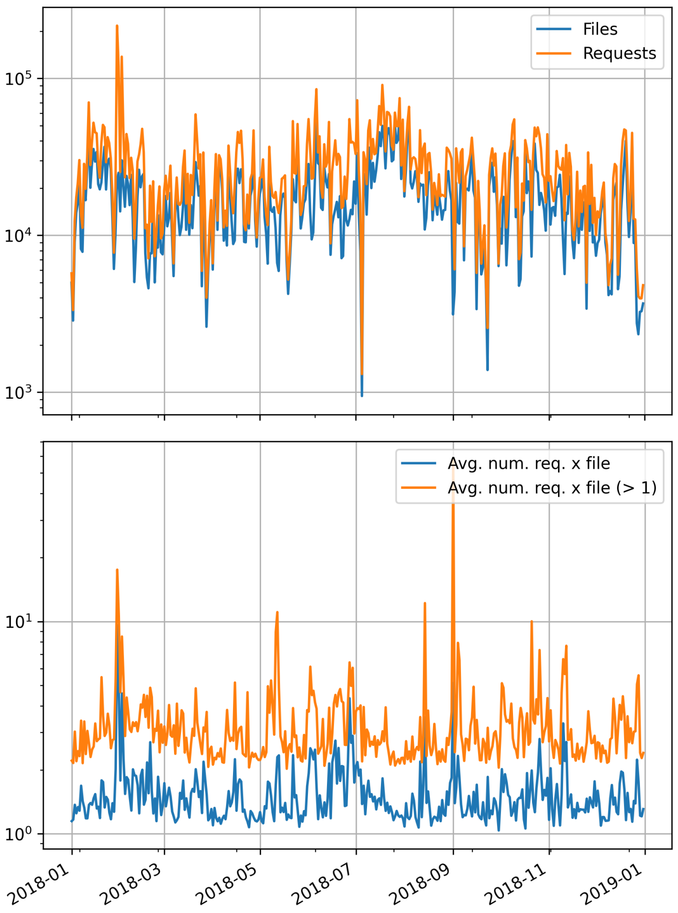

Figure 6: In the plot above: daily number of files and requests as functions of the day of the year. In the plot below: daily average number of requests per file (divided in two categories: all, and the ones requested more than once).

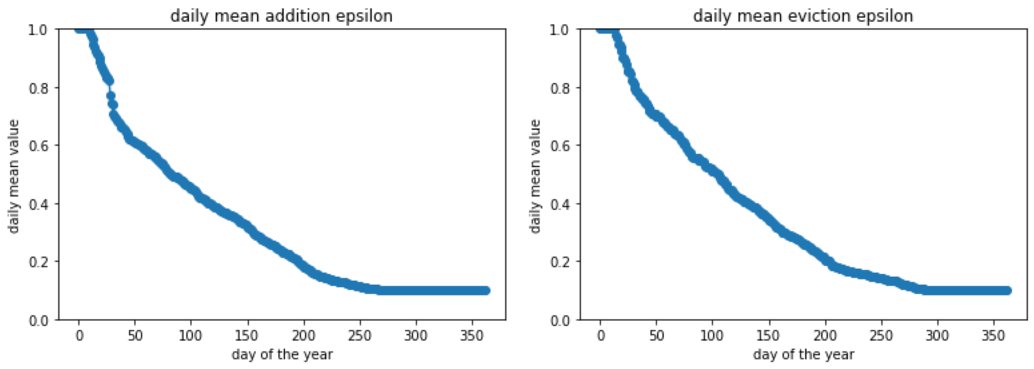

Figure 7: DQN QCACHE 100TiB simulation: mean ε daily value as a function of the day of the year for the addition agent and the eviction agent, respectively.

$$throughput = \frac{readOnHitData}{N_{throughput}} \tag{2.7}$$

- *cost*: to evaluate the operational cost of the cache

$$cost = \frac{writtenData + deletedData}{N_{cost}} \tag{2.8}$$

- *score*: to evaluate the trade-off between the aforementioned metrics

$$score = \frac{throughput}{cost} \tag{2.9}$$

$N_{throughput}$ and N_{cost} are normalization factors that are obtained using the same simulation described above, but with an infinite cache size that accepts everything. The resulting $readOnHitData$ value coming from that simulation is the highest possible considering the same time window and the dataset used. As a consequence, it is used as the $N_{throughput}$ normalizing factor. Moreover, we take into consideration the $2 \cdot writtenData$ from that simulation as a baseline reference for the cost value (using it as N_{cost}), since it represents the cost of a cache that writes and deletes all files requested by the clients in a specific time window.

3. Results

The results of the simulations with the different cache sizes are shown in table 2 and in fig. 8. More precisely, the daily values of the metrics defined above are averaged across all the year, and QCACHE results are compared to those obtained with classic caching algorithms. The latter include a Write Everything approach for the addition task associated with different eviction policies: LRU (Least Recently Used), LFU (Least Frequently Used), Size Big (biggest files are deleted first) and Size Small (smallest files are deleted first). Results are sorted by *score* values.

The QCACHE approach shows the best performance in terms of *score* in all three cases. Moreover, DQN QCACHE outperforms SCDL QCACHE only with 100TiB and 200TiB cache sizes: this could be related to the fact that DQN QCACHE parameters used for all simulations have been optimized for 100TiB cache. Thus, DQN could be tuned better on higher cache sizes.

Nevertheless, these results show anyway that the QCACHE approach is able to reduce *cost* while keeping *throughput* at a reasonable value at the simulation level (where to delete and write are two “zero-time” operations). As a consequence, we expect that a lower *cost* has a big impact in a real environment where fewer file deletion and writing operations will surely imply better performances in terms of file serving.

100 TiB

Algorithm	**Score**	**Throughput**	**Cost**
DQN QCACHE	**0.33**	0.40	**1.23**
SCDL QCACHE	0.26	0.45	1.74
Write everything + LRU	0.19	**0.50**	2.66
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

200 TiB

Algorithm	**Score**	**Throughput**	**Cost**
DQN QCACHE	**0.34**	0.41	**1.20**
SCDL QCACHE	0.33	0.55	1.65
Write everything + LRU	0.24	**0.59**	2.40
Write everything + LFU	0.20	0.52	2.58
Write everything + Size Big	0.15	0.42	2.89
Write everything + Size Small	0.13	0.39	2.98

500 TiB

Algorithm	**Score**	**Throughput**	**Cost**
SCDL QCACHE	**0.51**	0.72	1.41
Write everything + LRU	0.39	**0.74**	1.90
DQN QCACHE	0.35	0.41	**1.16**
Write everything + LFU	0.32	0.67	2.11
Write everything + Size Big	0.22	0.54	2.52
Write everything + Size Small	0.18	0.48	2.70

Table 2: Comparison of results (daily values averaged across the year). The best result for each metric is displayed in bold.

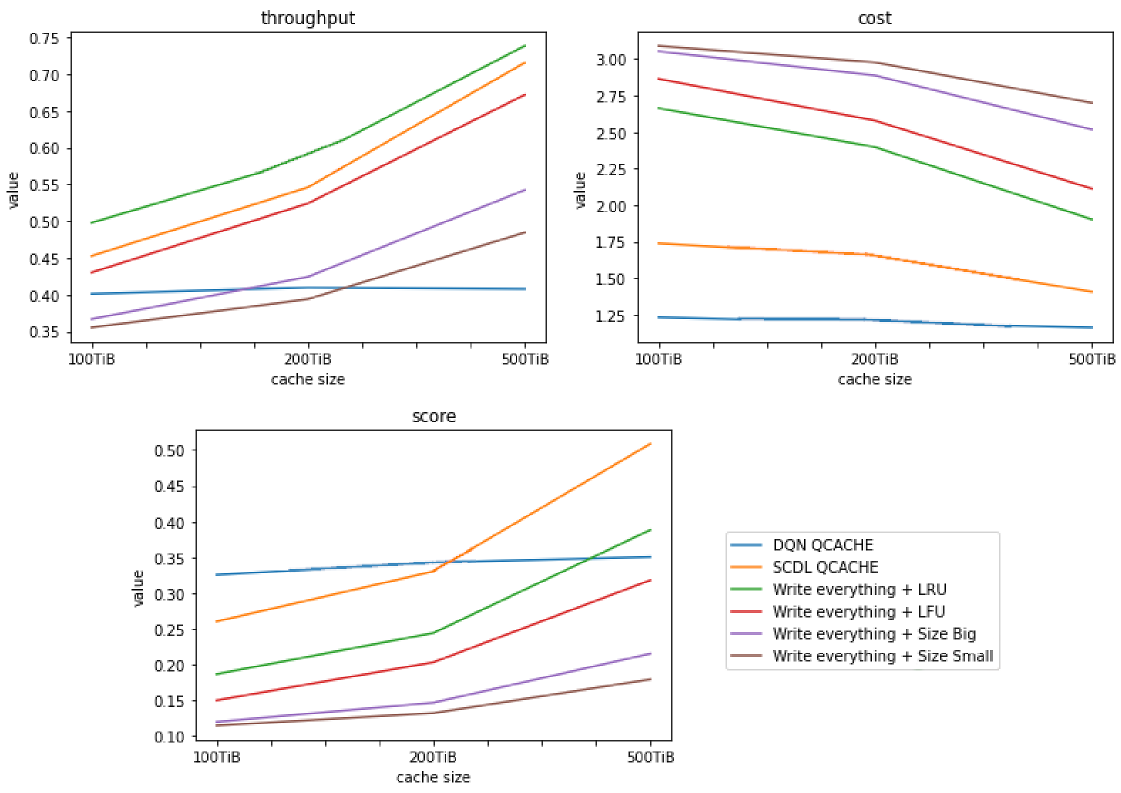

Figure 8: Throughput, cost and score as functions of cache size for all considered caching algorithms.

4. Conclusions

In this work, we presented how the Smart Caching problem in the CMS experiment context can be addressed with the help of Reinforcement Learning methods. In particular, we explained the QCACHE framework which is based on one or two Q-Learning agents for the addition and for the eviction part.

Initial results, obtained by simulating caches using real data from CMS monitoring data filtered for the Italian region, show that QCACHE approach achieves good results in terms of the trade-off between operational cost and efficiency. More specifically, DQN QCACHE outperforms all other algorithms in the case of small caches.

In the next future, we plan to optimize parameters for various cache sizes, with the enhancement of the reward-giving process. We will also consider domain features as inputs for the agents and perform simulations using data filtered for the US region. In the meantime, we are working on a testbed setup in order to do testing in a real environment, which includes the time domain and could give us a real feedback on how cache smartness impacts file serving.

References

[1] L. Evans and P. Bryant, *LHC machine*, *JINST* **3** (2008) S08001.

[2] CMS Collaboration, *The CMS experiment at the CERN LHC*, *JINST* **3** (2008) S08004.

[3] CMS Collaboration, *CMS offline and computing public results (2020)*, `https://twiki.cern.ch/twiki/bin/view/CMSPublic/CMSOfflineComputingResults`.

[4] A. Rizzi, G. Petrucciani and M. Peruzzi, *A further reduction in CMS event data for analysis: the NANOAOD format*, *EPJ Web Conf.* **214** (2019) 06021.

[5] I. Bird, S. Campana, M. Girone, X. Espinal, G. McCance and J. Schovancová, *Architecture and prototype of a WLCG data lake for HL-LHC*, *EPJ Web Conf.* **214** (2019) 04024.

[6] "WLCG DOMA." `https://twiki.cern.ch/twiki/bin/view/LCG/DomaActivities`.

[7] "ESCAPE project." `https://projectescape.eu/`.

[8] "US CMS." `https://uscms.org/index.shtml`.

[9] E. Fajardo, M. Tadel, J. Balcas, A. Tadel, F. Würthwein, D. Davila et al., *Moving the California distributed CMS XCache from bare metal into containers using Kubernetes*, *EPJ Web Conf.* **245** (2020) 04042.

[10] D. Ciangottini, G. Bagliesi, M. Biasotto, T. Boccali, D. Cesini, G. Donvito et al., *Integration of the Italian cache federation within the CMS computing model*, *PoS* **ISGC2019** (2019) 014.

[11] T. Boccali, G. Carlino and L. dell'Agnello, *The INFN scientific computing infrastructure: present status and future evolution*, *EPJ Web Conf.* **214** (2019) 03001.

[12] D. Spiga, D. Ciangottini, M. Tracolli, T. Tedeschi, D. Cesini, T. Boccali et al., *Smart Caching at CMS: applying AI to XCache edge services*, *EPJ Web Conf.* **245** (2020) 04024.

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

[14] A.P. Kryukov and A.P. Demichev, *Architecture of Distributed Data Storage for Astroparticle Physics*, *Lobachevskii J Math* **39** (2018) 1199–1206.

[15] G. Tian and M. Liebelt, *An effectiveness-based adaptive cache replacement policy*, *Microprocessors and Microsystems* **38** (2014) 98.

[16] T. Koskela, J. Heikkonen and K. Kaski, *Web cache optimization with nonlinear model using object features*, *Computer Networks* **43** (2003) 805.

[17] T. Chen, *Obtaining the optimal cache document replacement policy for the caching system of an EC website*, *European Journal of Operational Research* **181** (2007) 828.

[18] 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)*, pp. 1–6, 2018, DOI.

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

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A.A. Rusu, J. Veness, M.G. Bellemare et al., *Human-level control through deep reinforcement learning*, *Nature* **518** (2015) 529.

[21] M. Tracolli, M. Baioletti, V. Poggioni and D. Spiga, *Caching Suggestions Using Reinforcement Learning*, in *Machine Learning, Optimization, and Data Science*, pp. 650–662, Springer International Publishing, 2020, DOI.

[22] V. Kuznetsov, T. Li, L. Giommi, D. Bonacorsi and T. Wildish, *Predicting dataset popularity for the CMS experiment*, in *Journal of Physics: Conference Series*, vol. 762, p. 012048, IOP Publishing, 2016, DOI.

[23] M. Meoni, R. Perego and N. Tonellotto, *Dataset popularity prediction for caching of CMS big data*, *Journal of Grid Computing* **16** (2018) 211.