Low-Complexity Online Learning for Caching

Damiano Carra[a], Giovanni Neglia[b], Xufeng Zhang[b]

[a]*Computer Science Department, University of Verona, Italy*
[b]*Inria, Universite Cote d'Azur, France*

Abstract

Commonly used caching policies, such as LRU (Least Recently Used) or LFU (Least Frequently Used), exhibit optimal performance only under specific traffic patterns. Even advanced machine learning-based methods, which detect patterns in historical request data, struggle when future requests deviate from past trends. Recently, a new class of policies has emerged that are robust to varying traffic patterns. These algorithms address an online optimization problem, enabling continuous adaptation to the context. They offer theoretical guarantees on the *regret* metric, which measures the performance gap between the online policy and the optimal static cache allocation in hindsight. However, the high computational complexity of these solutions hinders their practical adoption.

In this study, we introduce a new variant of the gradient-based online caching policy that achieves groundbreaking logarithmic computational complexity relative to catalog size, while also providing regret guarantees. This advancement allows us to test the policy on large-scale, real-world traces featuring millions of requests and items—a significant achievement, as such scales have been beyond the reach of existing policies with regret guarantees. The regret guarantees and the low complexity are also maintained in cases where items have non-uniform sizes. To the best of our knowledge, the proposed solution is the only low-complexity no-regret policy for such a case, and our experimental results demonstrate for the first time that the regret guarantees of gradient-based caching policies offer substantial benefits in practical scenarios.

Keywords: Caching, Low-complexity, No-regret.

1. Introduction

Caching is a fundamental building block employed in various architectures to enhance performance, from CPUs and disks to the Web. Numerous caching policies target specific contexts, exhibiting different computational complexities. These range from simple policies like Least Recently Used (LRU), First In First Out (FIFO) [1] or Least Frequently Used (LFU) [2] with constant complexity to more sophisticated ones like Greedy Dual Size (GDS) [3] with logarithmic complexity, up to learned caches [4] that necessitate a training phase for predicting the next request.

All caching policies, implicitly or explicitly, target a different traffic pattern. For instance, LRU and FIFO favor recency, assuming requests for the same item are temporally close, while LFU performs well with stationary request patterns [5]. Learned caches [4], which use Machine Learning for request prediction, identify patterns in historical request sequences, and assume the same pattern will persist. In a dynamic context with continuously changing traffic patterns, none of these policies can consistently provide high performance, and it is indeed easy to design adversarial patterns that may hamper the performance of a specific caching policy [6, 7].

Some recent caching policies [6, 7, 8, 9, 10, 11] have been designed to be robust to any traffic pattern, making no assumptions about the arrival process. These policies are based on the *online optimization* [12] framework. The key performance metric in this context is *regret*, which is the performance gap (e.g., in terms of the hit ratio) between the online policy and the optimal static cache allocation in hindsight. Online caching policies aim for sub-linear regret with respect to the length of the time horizon, as this guarantees that their time-average performance is asymptotically at least as good as the optimal static allocation with hindsight. Policies with sub-linear regret are typically referred to as *no-regret* policies.

Limitation of the prior work. The major drawback of the no-regret policies is their computational complexity. Indeed, most of these policies have at least $\mathcal{O}(N)$ complexity per request, where N is the catalog size [11]. This high complexity has limited the adoption and *experimental validation* of no-regret policies. Figure 1 shows the trace length T and the catalog size N used in the literature proposing no-regret regret policies versus those in the more broader literature on caching policies (the corresponding references are listed in Table 1). No-regret policies have typically been evaluated using traces (labeled $\texttt{no-regr}_n$) that are either generated synthetically, or sub-sampled from much larger traces. Their catalogs and trace lengths are several orders of magnitude smaller than those considered in the rest of the literature. In Sec. 7, we provide evidence that the actual running time of these policies scales linearly with the catalog size N, which severely limits the number of requests they can process per unit of time (see, e.g., Fig. 9 and the corresponding discussion).

To the best of our knowledge, only the Follow The Perturbed Leader (FTPL) policy [10] can be, in some cases, implemented with $\mathcal{O}(\log N)$ complexity (see Sec. 2.2). Nevertheless, this policy—despite being, in practice, a noisy version of LFU, and thus performing well only when LFU does—has been evaluated, like other no-regret policies, primarily on short traces with small catalogs (see Table 1 and Fig. 1). Additionally, the regret guarantees have been proven for the uniform item size case, with no low complexity extension to handle non-uniform item sizes.

Contributions. In this work, we introduce a new online caching policy, OGB, based on the *gradient descent* approach that has $\mathcal{O}(\log N)$ amortized computational complexity per request, and sub-linear regret guarantees in the *integral caching* setting, i.e., when the cache stores entire items. This complexity

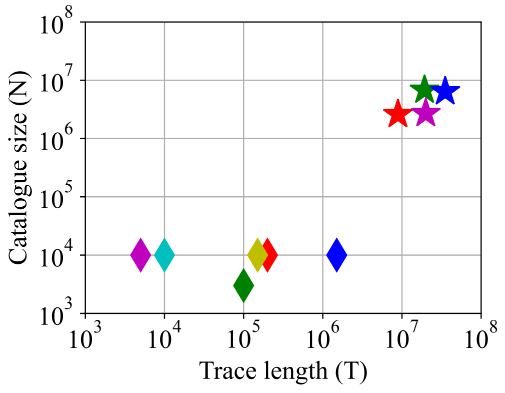

Figure 1: Trace length (T) and catalog size (N) used in no-regret caching papers (♦), and used commonly for evaluating caching policies (★).

Table 1: Trace references.

	Paper		Year	Label
♦	Paschos *et al.*	[6]	2019	`no-regr`$_1$
♦	Bhattacharjee *et al.*	[7]	2020	`no-regr`$_2$
♦	Paria *et al.*	[8]	2021	`no-regr`$_3$
♦	Mhaisen *et al.* (a)	[9]	2022	`no-regr`$_4$
♦	Mhaisen *et al.* (b)	[10]	2022	`no-regr`$_5$
♦	Si Salem *et al.*	[11]	2023	`no-regr`$_6$
★	Kavalanekar *et al.*	[13]	2007	`ms-ex`
★	Lee *et al.*	[14]	2016	`systor`
★	Song *et al.*	[15]	2019	`cdn`
★	Yang *et al.*	[16]	2020	`twitter`

matches that of commonly used caching policies, making it suitable for real-world contexts. Our solution relies on the clever joint design of two steps: (i) the selection of item storage probabilities, for which we modify the classic Online Gradient-Based (OGB_{cl}) policy originally proposed for *fractional caching*—when the cache can store arbitrarily small fractions of every item—and (ii) a sampling process in which we select the actual set of items stored in the cache, minimizing the number of new items retrieved upon each update.

Our solution also works when the cache can be updated only after a *batch* of requests are served. This case could be of interest in high-demand settings, or to amortize the computational cost of the caching policy and/or to reduce the load on the authoritative content server. The design maintains regret guarantees and logarithmic complexity independently of the batch size.

We then extend the proposed caching policy to the case where items have non-uniform sizes, showing that it maintains the regret guarantees and it can be implemented with $\mathcal{O}(\log N)$ amortized computational complexity. Finally, by removing the sampling process, our policy also works in the fractional caching setting, with amortized complexity $\mathcal{O}(N/B)$ per request, where B is the batch size, matching state-of-the-art no-regret policies in the fractional setting.

The low complexity of OGB allows us to evaluate it on traces with millions of requests and items—an accomplishment previous works could not achieve. We explore various scenarios where OGB outperforms traditional caching policies, demonstrating that gradient-based schemes adapt effectively to changes in traffic patterns.

In summary, our contributions are as follows:

- We propose OGB, the first integral online *gradient-based* caching policy that enjoys both sub-linear regret guarantees and $\mathcal{O}(\log N)$ amortized computational complexity per request, and can handle typical catalogs that characterize real-world applications.
- We extend these results to the non-uniform item size case, showing that our policy continues to enjoy regret guarantees and $\mathcal{O}(\log N)$ amortized complexity. To the best of our knowledge, this is the only low-complexity no-regret policy for such a case.
- The regret guarantees are confirmed also for the fractional caching, which has $\mathcal{O}(N/B)$ amortized complexity when operating over batches of B requests, allowing to mitigate the linear complexity intrinsically associated with this case.
- Through experiments on different public request traces with millions of items and tens of millions of requests, we show for the first time that the regret guarantees of online caching policies bring significant benefits in scenarios of practical interest.

Roadmap. The remainder of the paper is organized as follows. In Sec. 2, we provide background information on gradient-based caching policies and discuss the motivation behind the work. In Sec. 3, we offer a high-level description of the elements composing our solution, and we prove its regret guarantees. Sections 4 and 5 provide detailed descriptions of the building blocks of our scheme for the uniform size case. Section 6 discusses the extensions to more general settings, like heterogeneous file sizes, and the fractional case. In Sec. 7, we apply our solution to a set of publicly available traces. Finally, in Sec. 8 and Sec. 9, we discuss related work and conclude the paper.

2. Background and motivation

2.1. Online Gradient Based caching policy

No-regret caching policies make no assumptions about the arrival pattern and adapt dynamically to the received requests. They aim to solve an *online convex optimization* problem [7]. We consider a catalog of N items, denoted by $\mathcal{N} = 1, 2, \dots, N$, where each item i has a size s_i. The cache has a total capacity of C. We let the current time t coincide with the number of requests received so far. Each request is represented as a one-hot vector $\boldsymbol{r}_t$, with $r_{t,j} = 1$ for the requested item $j \in \mathcal{N}$, while all other components are set to zero.

We consider a batch operation, where the cache is allowed to update its state every $B \geq 1$ requests, as in [17, 11]. The batched operation may be motivated by the fact that requests indeed arrive in batches, or by the need to amortize the computational cost of the caching policy and/or to reduce the load on the authoritative content server. In the first case, the ordering of B consecutive requests at times $nB+1, nB+2, \dots, (n+1)B$ for $n \in \mathbb{N}$ is arbitrary.

We start describing the *fractional setting*, considered for example in [18, 19, 20, 6], where the cache can store arbitrarily small fractions of each item. The fractional setting is useful for modeling caching systems that work with item chunks much smaller than the item size, as is the case in video caches. The cache fractional state at time t is the vector $\boldsymbol{f}_t \in [0,1]^N$ satisfying $\sum_{i=1}^N s_i f_{t,i} = C$, where each $f_{t,i}$ denotes the fraction of item i stored in the cache, with the fractions chosen so as to completely fill the cache.

For the t-th request, the cache receives a reward $\phi_t(\boldsymbol{f}_t) \triangleq \sum_{i=1}^N w_{t,i} r_{t,i} f_{t,i}$, where $w_{t,i}$ is a weight that may be correspond to the cost of retrieving the whole item from the origin server. Upon a request for item i ($r_{t,i} = 1$), $w_{t,i} f_{t,i}$ can be interpreted as the cost reduction due to the fact that the fraction $f_{t,i}$ of item i is stored locally. To simplify the presentation, we assume that all items have the same size and weight, i.e., $s_i = w_{t,i} = 1$ for all t and i. In this case, the cache capacity C corresponds to the number of items that can be stored, and the rewards correspond to the (fractional) cache hits. In Sec. 6, we extend our results to the general case.

The performance of an online caching policy $\mathcal{A}$ is characterized by the static *regret* metric, defined as:

$$R_T(\mathcal{A}) \triangleq \sup_{\boldsymbol{r}_0, \boldsymbol{r}_1, \dots, \boldsymbol{r}_{T-1}} \left\{ \sum_{t=0}^{T-1} \phi_t(\boldsymbol{f}^*) - \mathbb{E}\left[\sum_{t=0}^{T-1} \phi_t(\boldsymbol{f}_t) \right] \right\} \tag{1}$$

where $\boldsymbol{f}^* = \arg\max_{\boldsymbol{f} \in \mathcal{F}} \sum_{t=1}^T \phi_t(\boldsymbol{f})$ is the best-in-hindsight static cache allocation knowing all future requests,[1] and the expectation is over possible random choices of the caching policy. Such an allocation $\boldsymbol{f}^*$ is usually referred to as OPT. The regret measures the accumulated reward difference between the baseline $\boldsymbol{f}^*$ and the online decisions $\{\boldsymbol{f}_t\}_{t=0,\dots,T-1}$ by algorithm $\mathcal{A}$. An algorithm is said to achieve sub-linear regret (or have no-regret), if the regret can be bounded by a sub-linear non-negative function of the time-horizon T. In this case, the ratio R_T/T is upper-bounded by a vanishing function as T diverges, *i.e.*, the time-average performance of the online policy is asymptotically at least as good as the performance of the best static caching allocation with hindsight.[2] Since the supremum in (1) is taken over all possible request sequences, the sequence of requests can be viewed as being generated by an *adversary* with the goal of maximizing the cache's regret.

[1]Note that one can always select $\boldsymbol{f}^*$ so that only full items are stored, i.e., $f_i^* \in \{0,1\}$ for all $i \in \{1, \dots, N\}$.

[2]Note that the regret may also be negative as it is the case in Fig. 6, right.

The first no-regret caching policy was proposed in [6] for the fractional setting and later extended to batched operation ($B > 1$) and the integral setting in [11]. We denote this "classic" policy by $\mathrm{OGB}_{\mathrm{cl}}$ and introduce it directly in the batched setting. The cache is updated every B requests at time instants $\mathcal{T}_B \triangleq nB : n \in \mathbb{N}$ as follows:

$$\boldsymbol{f}_t = \begin{cases} \Pi_{\mathcal{F}}\left(\boldsymbol{f}_{t-B} + \eta \sum_{\tau=t-B}^{t-1} \nabla \phi_\tau(\boldsymbol{f}_{t-B})\right), & \text{if } t \in \mathcal{T}_B, \\ \boldsymbol{f}_{t-1}, & \text{o.w.}, \end{cases} \tag{2}$$

where $\Pi_{\mathcal{F}}(\cdot)$ is the Euclidean projection onto the feasible state space $\mathcal{F}$, $\sum_{\tau=t-B}^{t-1} \phi_\tau(\boldsymbol{f}_{t-B})$ is the total reward accumulated by the cache for the previous B requests, and $\eta \in \mathbb{R}^+$ is the learning rate. The learning rate is a parameter of the scheme, and selecting $\eta = \sqrt{\frac{C}{BT}\left(1 - \frac{C}{N}\right)}$ guarantees that $R_T(\mathrm{OGB}_{\mathrm{cl}}) \leq \sqrt{BTC\left(1 - \frac{C}{N}\right)}$ [11, Corollary 4.5].

In the *integral setting*, the cache is forced to store each item in its entirety. The cache state at time t can then be characterized by a binary vector $\boldsymbol{x}_t \in \{0,1\}^N$. A hard cache capacity constraint imposes $\sum_{i=1}^N x_{t,i} = C$, but a soft cache capacity constraint has also been considered in the literature, e.g., in [21, 22, 23, 24][25, 26], and in practical system implementation [27], requiring that the constraint is satisfied only in expectation over some random choices of the caching policy, i.e. $\mathbb{E}\left[\sum_{i=1}^N x_{t,i}\right] = C$. Besides satisfying the cache size constraint, one would also like to minimize the changes between the current and the previous cache states. In LRU, for instance, at most one item is added and another item is removed, which has a limited impact on the amount of data exchanged between the cache and the origin server.

The concept of regret can be immediately extended to the integer setting by replacing $\boldsymbol{f}_t$ in (1) by $\boldsymbol{x}_t$. Interestingly, a no-regret fractional caching policy can be transformed in a no-regret integral caching policy by augmenting it with an item *sampling* procedure that guarantees $\mathbb{E}[\boldsymbol{x}_t] = \boldsymbol{f}_t$ [11, Sec. 6]. This sampling procedure is also often referred to as a *rounding scheme.* We discuss different rounding schemes in Sec. 5.

Projection Time Complexity. The most expensive operation in $\mathrm{OGB}_{\mathrm{cl}}$ is the projection on the capped simplex $\mathcal{F}$ in Eq. 2, required both in the fractional and the integral setting. After B requests are processed, the vector $\boldsymbol{y}_t = \boldsymbol{f}_{t-B} + \eta \sum_{\tau=t-B}^{t-1} \nabla \phi_{t-B}(\boldsymbol{f}_{t-B})$ does not belong to the feasible set $\mathcal{F}$ (some components of $\boldsymbol{f}_{t-B}$ have been increased) and needs to be projected back to $\mathcal{F}$. The projection corresponds to solving the following problem:

$$\begin{aligned} \min_{\boldsymbol{f} \in \mathbb{R}^N} \quad & \frac{1}{2}\|\boldsymbol{f} - \boldsymbol{y}_t\|^2 \\ \text{s.t.} \quad & 0 \leq f_i \leq 1 \ \ \forall i \in \{1, \dots, N\}, \\ & \sum_{i=1}^N f_i = C. \end{aligned} \tag{3}$$

For the general case in which more than one component of $\boldsymbol{y}_t$ differs from the current $\boldsymbol{f}_{t-B}$, the best known Euclidean projection algorithm has $\mathcal{O}(N^2)$ complexity [28]. If a single component changes with respect to the current $\boldsymbol{f}_t$, Paschos *et al.* [6] show that the complexity can be reduced to $\mathcal{O}(N \log N)$, later refined to $\mathcal{O}(N)$ [29]. The best known per-request amortized cost is then $\mathcal{O}(N^2/B)$ for $B > 1$ and $\mathcal{O}(N)$ for $B = 1$. We observe that no-regret algorithms based on online mirror descent (OMD) with a negative entropy mirror map, rather than on classic gradient descent, rely on a different projection (Bregman projection) and may achieve $\mathcal{O}(N/B)$ per-request amortized cost [11]. However, OGB_{cl} has been shown to outperform OMD in most settings of practical interest [11].

Sampling Time Complexity. In the integral setting, item sampling incurs additional costs. A naïve solution to update the cache is to consider each item independently and select it with probability f_i. This approach satisfies the soft cache constraint but not the hard one. Moreover, it can potentially lead to changing a large number of items in the cache. To select exactly C items, it is possible to adopt the *systematic sampling* approach devised by Madow *et al.* [30]. While this represents an elegant solution to a complex issue, it still requires $\mathcal{O}(N)$ operations and does not guarantee minimizing the changes in the cache, although experimental evidence shows that the number of replaced items is indeed small [11]. By formulating the problem as an optimal transport problem, we can guarantee the selection of exactly C items and minimize the number of replacements. However, this approach comes with a high computational cost, namely $\mathcal{O}(N^3)$ complexity [31]. Overall, it is not possible to simultaneously achieve these three desired properties—selection of exactly C items, minimization of replacements, and low computational complexity.

Both building blocks described above have an amortized cost of $\Omega(N/B)$, which is linear in the catalog size. In what follows, we demonstrate that, in the integral setting with soft cache capacity constraints, a slight modification of the OGB_{cl} policy in (2), combined with a smart joint design of the projection and sampling steps, results in a per-request cost of $\mathcal{O}(\log N)$ for any batch size $B \geq 1$, while still providing the same regret bound. In the fractional setting, our policy incurs a cost of $\mathcal{O}(N/B)$, improving upon OGB_{cl} by a factor N when $B > 1$.

2.2. Motivation

Works proposing no-regret caching policies usually evaluate them in adversarial settings, where the request sequence is specifically designed to limit the effectiveness of common caching policies such as LRU and LFU. Such an adversarial setting may be considered unrealistic. Figure 2 (top left), shows the results with a real-world trace. This trace contains 10^5 requests for 10^4 items (cache size $C = 500$ items) and it has been subsampled from the publicly available trace from [15, 32] (trace with label `cdn` in Table 1), to generate a trace comparable, in terms of catalog size and number of requests, with the traces considered in previous works on no-regret policies (traces with labels `no-regr` in Table 1). While LRU works reasonably well on this trace, it is matched by

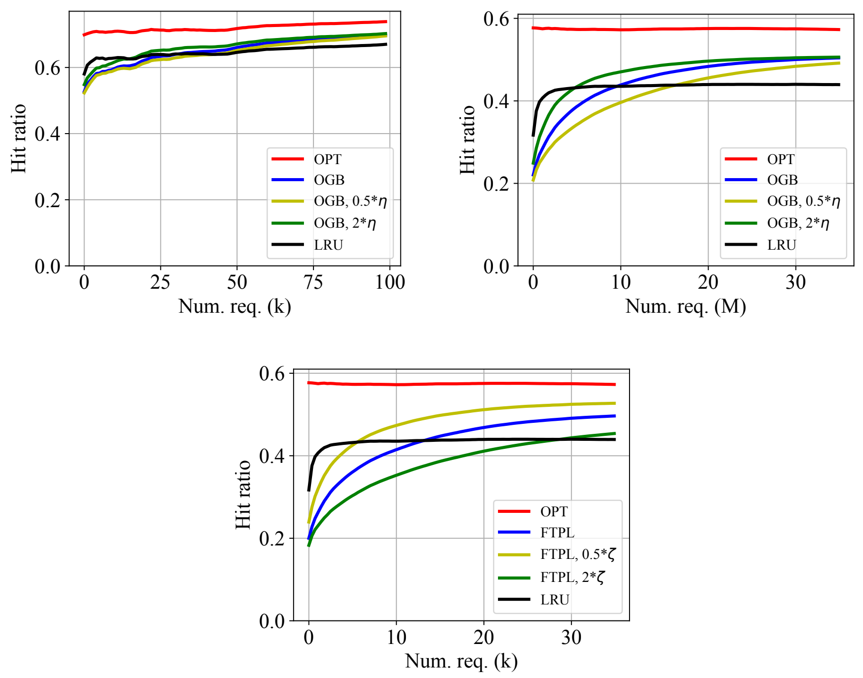

Figure 2: Real-world traces: sensitivity of OGB_{cl} with a short trace (top, left) and a long trace (top, right), and sensitivity of FTPL (bottom). In OGB_{cl}, the parameter η is the learning rate, while in FTPL the parameter ζ is the noise added to the LFU counters.

online gradient-based policies and, in any case, does not provide theoretical guarantees.

The main issue with the OGB_{cl} policy is its $\mathcal{O}(N)$ computational complexity, which may limit it to a theoretical exercise with no practical application. To the best of our knowledge, no work shows that OGB_{cl} performs well with large catalogs and long traces that have heterogeneous patterns, like the ones used commonly in the caching literature (last 4 lines of Table 1). Thanks to OGB's reduced complexity of $\mathcal{O}(\log N)$, we can easily evaluate its performance on real-world traces with tens of millions of items and hundreds of millions of requests. For example, in Fig. 2 (top right), we tested the trace `cdn` ($6.8 \cdot 10^6$ items and $3.5 \cdot 10^7$ requests) without the need to subsample it.

The only existing no-regret policy with sublinear complexity is a variant of the *Follow The Perturbed Leader* (FTPL) policy [7]. FTPL operates as LFU, but adds to each counter an independent Gaussian random variable with zero mean and standard deviation ζ. By tuning $\zeta = \frac{1}{(4\pi \log N)^{1/4}} \sqrt{\frac{T}{C}}$, FTPL guarantees sublinear regret [7]. The variant generates the Gaussian noise only once at the beginning [10], rather than for each request as in the original proposal [7], thereby reducing FTPL's complexity from $\mathcal{O}(N \log N)$—since the counter vector must be sorted at every request—to $\mathcal{O}(\log N)$. Nevertheless, the FTPL policy presents some limitations. First, FTPL is equivalent to LFU with some initial noise, causing it to suffer from the same limitations as LFU, such as poor adaptability to dynamic traffic patterns. Second, FTPL has a high sensibility to variations in its parameter ζ, as shown in Fig. 2 (bottom). This variability can result in a lower hit ratio, which has a significant impact on serving average latency.. By contrast, OGB is robust to the setting of its parameter η (Fig. 2, top). A possible solution to the sensitivity of FTPL to ζ would be to make the parameter change using online learning algorithms, but this approach would be orthogonal to the main objective of this paper and represents an interesting future research direction.

In addition, our experiments in Sec. 7 show that OGB is better suited than FTPL to track changes in request patterns. Finally, FTPL usually assumes that all items have uniform size. The approximate optimistic FTPL algorithm proposed in [10] achieves a sublinear $\frac{1}{2}$-regret bound with an unequal-size solution. However, its complexity is $\mathcal{O}(N)$, even when the optimistic predictions are disregarded in the algorithmic design. In contrast, OGB can be easily extended to handle non-uniform item sizes, as we do in Sec. 6, while still maintaining $\mathcal{O}(\log N)$ complexity and sublinear regret guarantees.

3. Solution overview

Our scheme is presented in Figure 3 and Algorithm 1 for the integral setting. We denote our scheme as OGB because it is almost identical to the $\mathrm{OGB_{cl}}$ policy in (2). Indeed, as $\mathrm{OGB_{cl}}$, OGB maintains a vector $\boldsymbol{f}_t$ indicating the probability with which each item is stored, and updates the integral cache status every B requests according to $\boldsymbol{f}_t$, guaranteeing that $\mathbb{E}[\boldsymbol{x}_t] = \boldsymbol{f}_t$. The only difference is that, while $\mathrm{OGB_{cl}}$ updates the vector $\boldsymbol{f}_t$ once every B requests, OGB updates it after every request as follows:[3]

$$\boldsymbol{f}_t = \Pi_{\mathcal{F}}\left(\boldsymbol{f}_{t-1} + \eta \nabla \phi_{t-1}(\boldsymbol{f}_{t-1})\right). \tag{4}$$

As we are going to show, OGB achieves $\mathcal{O}(\log N)$ per-request amortized cost for any $B \geq 1$ against $\mathrm{OGB_{cl}}$'s $\mathcal{O}(N)$ and $\mathcal{O}(N^2/B)$ for $B = 1$ and $B > 1$, respectively. This may appear surprisingly because OGB performs more projections and, as observed in Sec. 2, the projection is the most expensive step in $\mathrm{OGB_{cl}}$. A first explanation is that projections of vectors perturbed in a single component are less expensive to compute ($\mathcal{O}(N \log N)$ vs $\mathcal{O}(N^2)$, see [6]). However, this difference alone does not account for OGB's $\mathcal{O}(\log N)$ complexity. The key lies in the joint design of the projection and the sampling steps which allows the new cache state $\boldsymbol{x}_t$ to be sampled at $t \in \mathcal{T}_B$ without needing to update the vectors of probabilities $\boldsymbol{f}_{t-B+1}, \ldots, \boldsymbol{f}_t$—a procedure which would require at least $\mathcal{O}(N)$ operations.

Before presenting OGB's implementation in detail, we observe that $\mathrm{OGB_{cl}}$ and OGB lead to different sequences $(\boldsymbol{f}_t)_{t \in \mathcal{T}_B}$. The first question is whether

[3]Note that $\mathrm{OGB_{cl}}$ and OGB coincide for $B = 1$.

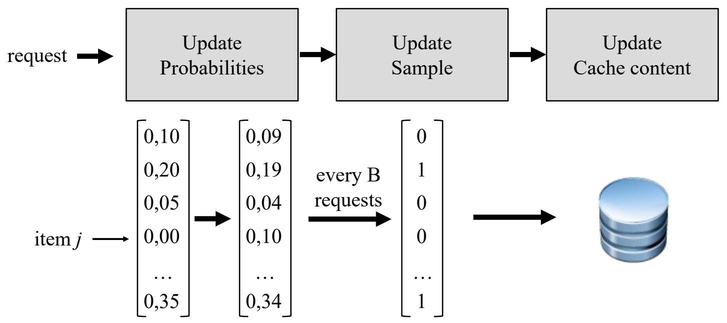

Figure 3: High level view of the building blocks that compose the solution: the item selection passes through the computation of the caching probabilities.

Algorithm 1: OGB scheme

```
1  $\boldsymbol{f}_0 \leftarrow \frac{C}{N}\mathbf{1}$;
2  $\boldsymbol{x}_0 \leftarrow$ BERNOULLI($\boldsymbol{f}_0$);
3  for $t = 1, \dots, T$ do
4      $\boldsymbol{f}_t \leftarrow$ UPDATEPROBABILITIES($\boldsymbol{f}_{t-1}, \boldsymbol{r}_{t-1}$);
       // update according to (4)
5      if $t$ % $B == 0$ then
           // batch processed
6          $\boldsymbol{x}_t \leftarrow$ UPDATESAMPLE($\boldsymbol{f}_t, \boldsymbol{x}_{t-B}$);
           // update the cache, $\mathbb{E}[\boldsymbol{x}_t] = \boldsymbol{f}_t$
```

OGB enjoys the same regret guarantees of OGB$_{\text{cl}}$. The following proposition confirms that this is the case.

Theorem 3.1. *The policy OGB in Algorithm 1 with* $\eta = \sqrt{\frac{C\left(1-\frac{C}{N}\right)}{TB}}$ *has regret upperbounded by* $\sqrt{C\left(1-\frac{C}{N}\right)TB}$.

The proof can be found in Appendix. A.

In the following, we summarize the challenges of each individual step, UPDATEPROBABILITIES and UPDATESAMPLE. For ease of presentation, in Secs. 4 and 5 we consider the uniform weight and uniform size case, *i.e.*, $w_{t,i} = s_i = 1$ for all t and i. The extension to the general case is presented in Sec. 6 and involves some cumbersome adaptations that do not alter the overall structure of our solution.

4. Updating the Probabilities

When an item is requested, the caching strategy should increase the probability of that item being cached, while decreasing the other items' probabilities. These two operations are indeed accomplished in two different steps by the OGB policy. First, we increment the component related to the requested item by a value equal to the step size η. Then, the projection $\Pi_{\mathcal{F}}(\cdot)$ decreases all the components, such that their sum is equal to C. In order to design a low complexity algorithm, we need to understand how the projection actually operates.

From the previous vector $\boldsymbol{f}_{t-1}$, when item j is requested, we obtain the following vector $\boldsymbol{y}_t$:

$$y_{t,i} = \begin{cases} f_{t-1,i} & \text{if } i \neq j, \\ f_{t-1,i} + \eta & \text{if } i = j. \end{cases}$$

Clearly we have $\sum_{i=1}^{N} y_{t,i} = C + \eta$ and the aim of the projection is to remove the excess η. Let $\mathcal{M}_p = \{i : y_{t,i} > 0\}$ be the set of indexes of the positive components of $\boldsymbol{y}_t$. In order to minimize the squared norm in (3), the excess should be *uniformly taken from each positive component*. Let $\rho = \eta/|\mathcal{M}_p|$, then the projection $\boldsymbol{f}_t$ would be:

$$f_{t,i} = \begin{cases} y_{t,i} - \rho & \text{if } i \in \mathcal{M}_p, \\ 0 & \text{otherwise.} \end{cases}$$

Note that the set $\mathcal{M}_p$ also includes the index of requested item j. As an example, Fig. 4 shows a small vector with six components. The initial state is represented by the blue bars. When a new request for item 1 arrives, we first add the orange contribution to component 1. We then uniformly decrease all the components (resulting in the green bars) to ensure the sum of all components remains constant. To ensure that all components of the projection fall within the range [0,1], there are some corner cases that need to be considered.

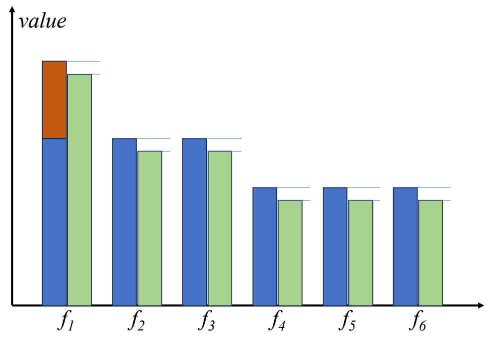

Figure 4: Example of a projection on a small vector. Starting from a feasible solution (blue bars), a new request for item 1 generates an excess (orange bar), which is taken evenly from all the components (green bars) to obtain a new feasible solution.

The first case occurs when a component becomes negative, *i.e.*, $y_{t,i} > 0$, but $y_{t,i} - \rho < 0$. This leads to changes in the set $\mathcal{M}_p$ and the excess to be redistributed. Specifically, $\eta' = \eta - \sum_{i:y_{t,i}<\rho} y_{t,i}$, $\mathcal{M}'_p = \mathcal{M}_p \setminus \{i : y_{t,i} < \rho\}$, and $\rho' = \eta'/|\mathcal{M}'{}_p|$. This adjustment may be needed multiple times, but, as we show in Sec. 4.2, on average, a single item goes to 0 after each request and then the amortized per-request cost is bounded.

The second case occurs when the component related to the requested item j exceeds one, i.e., $f_{t-1,j} \leq 1$, but $f_{t-1,j} + \eta - \rho > 1$. This case is easy to manage: the component $f_{t,j}$ is set to one, the excess becomes $\eta' = 1 - f_{t-1,j}$ and we remove j from $\mathcal{M}_p$, i.e., we consider the new set $\mathcal{M}'_p = \mathcal{M}_p \setminus \{j\}$. Note that in the trivial case where $f_{t-1,j} = 1$, the j-th component of $\boldsymbol{y}_t$ becomes $y_{t,j} = 1 + \eta$, and the final projection is simply $\boldsymbol{f}_t = \boldsymbol{f}_{t-1}$. We observe that changing the excess and the set over which this should be distributed may lead to reconsider also which components become negative, but the amortized per-request cost is still bounded (Sec. 4.2).

Although these corner cases require some considerations, the key observation remains: the projection fundamentally involves a uniform redistribution of excess. This characteristic can be leveraged to design a low-complexity algorithm for updating the projection.

4.1. Improving Projection: Main Idea

Consider two items, a and b, with $f_a > 0$ and $f_b > 0$.[4] Assume that they are not requested during two consecutive time slots and that their updated values do not become zero. After the first request, their values are decreased by the same quantity ρ_1. After the second request, their value are decreased by the same quantity ρ_2. In this case, we do not need to actually change f_a and f_b, but we can maintain an external variable $\rho = \rho_1 + \rho_2$, knowing that the projections of the components f_a and f_b are indeed $f_a - \rho$ and $f_b - \rho$. We denote the "unadjusted" version of $\boldsymbol{f}$ by $\tilde{\boldsymbol{f}}$, and keep track of the adjustment coefficient ρ separately, so that

$$f_i = \begin{cases} \tilde{f}_i - \rho & \text{if } \tilde{f}_i > 0, \\ 0 & \text{otherwise.} \end{cases}$$

To manage the corner cases in the projection step, we need to detect when a component goes below zero or is set to one. To this aim, we keep, in new data structure $\boldsymbol{z}$, a copy of the (positive components of the) vector $\tilde{\boldsymbol{f}}$ sorted by value. When we update ρ to a new value ρ', we check which components are smaller than ρ' (these are the components set to zero), remove them from $\boldsymbol{z}$ and adjust ρ'. The only component which may be set to one, is the component corresponding to the requested item, we deal with it separately, updating the corresponding value in $\tilde{\boldsymbol{f}}$ and in the data structure $\boldsymbol{z}$.

Managing the sorted copy of the coefficients has $\mathcal{O}(\log N)$ complexity. If we need the actual projection $\boldsymbol{f}$, we must compute each component from $\tilde{\boldsymbol{f}}$ and ρ. This implies that any algorithm updating the projection at each step cannot have a complexity less than $\mathcal{O}(N)$. However, in our case, since the projection is used to determine the probability of selecting items to cache, we do not need to update all components. We only need to identify which components fall below a given threshold, as explained in detail in Sec. 5.

[4]In the notation, we sometimes omit the time slot t when it is clear from the context.

Algorithm 2: UpdateProbabilities

```
   input: $\tilde{\boldsymbol{f}}$, current state
   input: $\rho$, current adjustment
   input: $\boldsymbol{z}$, ordered tree with positive coeffs. of $\tilde{\boldsymbol{f}}$
   input: $j$, index of the requested item
   input: $\eta$, OGB step size
   // The requested item is already 1
1  if $\tilde{f}_j - \rho == 1$ then
2      return;
   // The requested item was 0
3  if $\tilde{f}_j == 0$ then
4      $\tilde{f}_j \leftarrow \rho + \eta$;
5      $z_j \leftarrow \rho + \eta$;
6      $\boldsymbol{z} \leftarrow \boldsymbol{z} \cup \{z_j\}$;
7  else
       // Update with the OGB step
8      $\tilde{f}_j \leftarrow \tilde{f}_j + \eta$;
9      $z_j \leftarrow z_j + \eta$;
   // Remove items with negative values
10 $\eta' = \eta$;
11 repeat
12     $\rho' = \eta'/|\boldsymbol{z}|$;
13     $\mathcal{B} \leftarrow \emptyset$;
14     $\mathcal{B} \leftarrow \{i : z_i - \rho - \rho' < 0\}$ ;
15     $\eta' \leftarrow \eta' - (z_i - \rho), \quad \forall i \in \mathcal{B}$;
16     $\boldsymbol{z} \leftarrow \boldsymbol{z} \setminus \{z_i : i \in \mathcal{B}\}$ ;
17     $\tilde{f}_i \leftarrow 0, \quad \forall i \in \mathcal{B}$ ;
18 until $\mathcal{B}$ is empty;
   // Check the value of the requested item
19 if $(z_j \in \boldsymbol{z})$ and $(z_j - \rho - \rho' > 1)$ then
20     $\eta' = \eta - ((z_j - \rho) - 1)$;
21     $\boldsymbol{z}, \tilde{\boldsymbol{f}} \leftarrow$ RestoreRemoved();
22     $\boldsymbol{z} \leftarrow \boldsymbol{z} \setminus \{z_j\}$ ;
23     $\tilde{f}_j \leftarrow -1$ ;
24     GoTo line 11 ; // This can happen only once
   // Update $\rho$
25 $\rho \leftarrow \rho + \rho'$;
26 if $z_j \notin \boldsymbol{z}$ then
27     $z_j \leftarrow 1 + \rho$;
28     $\boldsymbol{z} \leftarrow \boldsymbol{z} \cup \{z_j\}$ ;
29     $\tilde{f}_j \leftarrow 1 + \rho$ ;
30 return $\rho, \tilde{\boldsymbol{f}}, \boldsymbol{z}$
```

The proposed Algorithm 2 is based on the fact that the projection is a constrained quadratic program and the objective function is strictly convex with a unique solution characterized by the KKT conditions [33]. As done in [28] and [6], we look at the set of coefficients that are respectively greater than one, between zero and one, and zero, and we adjust them with the uniform redistribution of the excess. Differently from the above mentioned papers, following the ideas presented in Sec. 4.1, we maintain an ordered data structure $\boldsymbol{z}$, which contains the positive coefficients of the vector $\tilde{\boldsymbol{f}}$, and an external adjustment coefficient ρ. When a new request for item j arrives, we first check if $\tilde{f}_j - \rho$ is already equal to one, in which case there is no update. Otherwise, we assign the step size η to the component j. If $\tilde{f}_j$ was equal to zero, we adjust its value considering the current adjustment coefficient ρ.

The next block—lines 11–18—handles the corner case in which the redistribution of the excess produces negative coefficients. Although this block may potentially be executed multiple times, we prove that, on average, a single element will be set to 0 after each request, resulting in the block being executed only once. Consider the cache after the first t requests. At any time, there can be at most $N - C$ components of the vector $\boldsymbol{f}_t$ equal to 0. Moreover, at each request, at most one component can turn positive (because the corresponding item has been requested). It follows that the total number of components set to 0 over the t requests is upper-bounded by $N - C + t$. Therefore, on average, $1 + \frac{N-C}{t}$ new components are set to 0 at each request, and the average time complexity of the cycle at lines 11–18 is $\mathcal{O}(1)$. In practice, our experiments show that the loop is executed at most twice per request, as also observed in [6, IV.B] for the corresponding operations in their projection algorithm.

Lines 19–24 consider the corner case in which a coefficient is greater than one. There can be at most one such coefficient, corresponding to the requested item, therefore this block is executed at most once. If executed, we first derive the new excess to be distributed, and we restore the components removed in lines 11–18. We take out the component of the requested item (it will be restored at the end, at lines 26–29), and we repeat lines 11–18 with the updated excess η'.

Overall, the time-complexity of projection update is then determined by the need to keep the data structure $\boldsymbol{z}$ ordered, which is $\mathcal{O}(\log N)$.

5. Updating the Sample

After the update of the caching probabilities, the cache may change the set of stored items. In the literature, this is often referred to as a *rounding scheme* [8, 11], since the fractional state $\boldsymbol{f}_t$ with continuous components between 0 and 1 is mapped to a caching vector $\boldsymbol{x}_t$ whose components are either zero (the item is not cached) or one (the item is cached).

The selection can be seen as a *sampling process*, in which the probability that an item i belongs to the sample is proportional to its component f_i. Considering the sampling literature, this is known as *probability proportional to size* (PPS) sampling [34].

Problem definition. Since $\boldsymbol{f}$ varies as new requests arrive, we are interested in finding a scheme that, starting from the previous sample, and considering the updated $\boldsymbol{f}$, draws a new sample that has a large overlap with the previous one. Sample coordination refers to the process of adjusting the overlap between successive samples [35]. With positive coordination the number of common items between two consecutive samples is maximized, while with negative coordination this number is minimized. We aim at finding a positive coordination design that is computationally efficient.

Current solutions. PPS sampling schemes that provide an exact number of samples—such as systematic sampling used in [8, 11]—have $\mathcal{O}(N)$ complexity. To the best of our knowledge, there is no extension of systematic sampling to successive sampling that would allow for a reduction in the computational cost of consecutive sampling. The only option is to re-apply the scheme from scratch, without guarantees on the level of coordination across consecutive samplings.

Any scheme that aims at maximizing the overlap of successive samples with an exact sample size, requires the computation of joint inclusion probabilities over subsequent samples, which in turn involves an update on every pair of items [35], so it has at least $\mathcal{O}(N^2)$ complexity.

Our approach. The conclusion is that under hard cache capacity constraints the sampling scheme complexity is $\mathcal{O}(N)$ and $\mathcal{O}(N^2)$ if we want to maximize the overlap between consecutive cache allocations. We show that under a soft cache constraint, which allows the number of item to satisfy the contraint in expectation, we can design a low-complexity scheme with a tunable level of sample coordination.

5.1. Coordination of Samples

First sample. The components of $\boldsymbol{f}$ satisfy the following constraints: $0 \leq f_i \leq 1, \forall i \in \mathcal{N}$, and $\sum_{i=1}^{N} f_i = C$. Therefore, we can apply *Poisson sampling* [36], *i.e.*, we decide to include an item in the sample independently from the other items. To this aim, we associate a random number p_i, drawn from a uniform distribution between zero and one, to each item i, and we include the item i in the sample if and only if $p_i \leq f_i$. The resulting sample has a random sample size with expectation C, the cache size. With sufficiently large C the variability around the expectation is limited. In particular, the variability is the largest when each item has the same probability to be stored in the cache, i.e., $f_i = C/N$ for each $i \in \mathcal{N}$. Considering this setting, the coefficient of variation (the ratio between the standard deviation and the expected value) for the number of items sampled can be easily upper-bounded by $1/\sqrt{C}$. It is then smaller than 1% as far as $C \geq 10000$. Moreover, under a Gaussian approximation, the probability that the number of items sampled exceeds the average cache size C by ϵC is upper-bounded by $\Psi(\epsilon\sqrt{C})$, where $\Psi(\cdot)$ is the complementary cumulative distribution function of a standard normal variable. For example, we can conclude that, in the same setting, the probability that the instantaneous cache occupancy exceeds C by more than 3% is smaller than 0.13%. Theorem 1 in [37] presents a similar result based on the Chernoff bound.

It is possible to decrease cache occupancy variability further by adopting the *collocated sampling scheme* [38]. However, as our experiments will show, the variability with Poisson sampling is limited and never exceeds 0.5% of the cache size. Therefore, the impact of a collocated sampling scheme would be minimal.

Subsequent samples. Brewer *et al.* [39] showed that Poisson sampling guarantees positive coordination if the random value p_i is *permanently* associated to item i. In particular, the expected number of overlapping items between two consecutive samples is maximized [40, Prop. 1]. The sample selection rule then becomes: the item is included as long as $p_i \leq f_{t,i}$. The values $\{p_i\}$ may periodically be randomly redrawn to reduce potential over/under-representation of some items.

Putting the pieces together. In Sec. 4, we proposed a scheme where, instead of computing f_i at every request, we update the global adjustment ρ, along with $\tilde{f}_i$ if item i has been requested, knowing that $f_i = \tilde{f}_i - \rho$. After B updates, we need to determine which items should be included in or evicted from the cache. We distinguish three groups of items.

The first group includes items that have been requested in the last B requests. The value of f_i for these items changed. If the item is not currently cached, we check whether $f_i \geq p_i$, i.e., $\tilde{f}_i - \rho \geq p_i$. If that is the case, we add it to the cache.

The second group contains the items that have not been requested and were not in the cache: since f_i decreased, the items continue to stay out of the sample, *i.e.*, we do not need to manage all the items not cached and not requested.

The third group includes all the cached items (excluding the requested ones). We need to check if, for some items, f_i became smaller than p_i, *i.e.*, $\tilde{f}_i - \rho < p_i$. For these items, both $\tilde{f}_i$ and p_i remained unchanged, only ρ has been updated. The key observation is that, after B updates, for all the cached items (except the ones that have been requested), the difference $d_i = \tilde{f}_i - p_i$ remains constant. We can then maintain the values d_i in an ordered data structure and evict the items for which $d_i < \rho$ as ρ changes. The eviction, like any other operation on the ordered data structure, has logarithmic complexity.

5.2. Item selection: The algorithm

Algorithm 3 summarizes the sampling operation to be executed every B requests. For each requested item i, if the item is already in the cache, we update the difference d_i. If it is not, we potentially insert it into the cache (lines 2-8). Overall, we can update or insert at most B values in the data structure.

Then, we evict all items whose difference d_i is smaller than the current value of ρ. We can prove that, on average, B elements need to be evicted (similarly to the proof that on average one component of $\boldsymbol{f}$ is set to 0 in Sec. 4.2). The ordered data structure $\boldsymbol{d}$ guarantees $\mathcal{O}(\log N)$ complexity for each eviction, along with $\mathcal{O}(\log N)$ complexity for any other operation involving the data structure (insertion, lookup). Overall, we achieve the target $\mathcal{O}(\log N)$ per-request amortized complexity for the sampling procedure.

Algorithm 3: UpdateSample

```
input: $\tilde{\boldsymbol{f}}$, current fractional state
input: $\boldsymbol{x}$, current cache state
input: $\rho$, current adjustment
input: $\boldsymbol{p}$, permanent random numbers
input: $\boldsymbol{d}$, ordered tree with differences $(\tilde{f}_i - p_i)$
input: $J$, set of indexes of the requested items
// Manage the requested items
1  for $j \in J$ do
2      if $j$ index in $\boldsymbol{d}$ then
3          $d_j \leftarrow (\tilde{f}_j - p_j)$;
4      else
5          if $\tilde{f}_j - \rho \geq p_j$ then
6              $d_j \leftarrow (\tilde{f}_j - p_j)$ ;
7              $\boldsymbol{d} \leftarrow \boldsymbol{d} \cup \{d_j\}$;
               // Add the item to the cache
8              $x_j = 1$;
// Eviction
9  $x_i = 0, \quad \forall i : d_i < \rho$;
10 $\boldsymbol{d} \leftarrow \boldsymbol{d} \setminus \{d_i : d_i < \rho\}$ ;
```

6. Extensions

In this section we discuss three different extensions: 1) heterogeneous weights across time and across items, 2) non-uniform item sizes, 3) fractional caching. For the non-uniform case, we provide a high-level view of the steps required to adapt the solution: the detailed discussion and proofs can be found in the Appendix (supplementary material).

6.1. Heterogeneous weights

The general reward function $\phi_t(\boldsymbol{f}_t) = \sum_{i=1}^{N} w_{t,i} r_{t,i} f_{t,i}$ introduced in Sec.2 considers a reward $w_{t,i}$ for a hit for item i at time t. For example, $w_{t,i}$ might reflect the cost of retrieving an item from the origin server to the cache. We assume that the values of $w_{t,i}$ are known.

From the perspective of algorithmic execution complexity, the introduction of $w_{t,i}$ has no impact. For each request, the gradient of $\phi_t(\boldsymbol{f}_t)$ becomes $w_{t,j}$ instead of 1, where j is the index of the requested item, and ρ will be adjusted accordingly.

Regret guarantees still hold, as shown in Theorem 6.1 below.

6.2. Heterogeneous item sizes

A common assumption in all previously proposed no-regret caching policies is that items have the same size, but this is far from being the case for example in Content Delivery Networks, where item sizes may differ by many order of magnitudes. To the best of our knowledge, the case of heterogeneous sizes has not been formally analyzed. In particular, we are not aware of any caching policies with sublinear regret guarantees in this setting.

From a problem formulation perspective, in the fractional setting, the feasible cache state space is defined by $\mathcal{F} = \boldsymbol{f} \in [0,1]^N : \sum_{i=1}^{N} f_i s_i = C$, where s_i represents the size of item i. Consequently, the projection step in the OGB policy, as defined in (3), must account for the constraint $\sum_{i=1}^{N} f_i s_i = C$. In the integral case, once the updated f_i values are computed, they can be treated as probabilities, and a sampling scheme can be applied.

Addressing heterogeneous sizes involves solving the following issues: (i) proving that OGB retains its regret guarantees; (ii) devising a computationally efficient algorithm for the projection step and the sampling scheme.

Regret guarantees. The generalization of OGB to non-uniform sizes and general weights $w_{t,i}$ maintains the sublinear regret guarantees, as confirmed by the following proposition (proof in Appendix B). Let $\bar{s} \triangleq \sum_{i=1}^{N} s_i/N$, $w_{\max} \triangleq \max\{w_{t,i}, t = 1, \dots, T, i = 1, \dots, N\}$, k_C such that $\sum_{i=1}^{k_C - 1} s_{j_i} < C \leq \sum_{i=1}^{k_C} s_{j_i}$, where $(j_1, j_2, \dots, j_N)$ is a permutation such that $s_{j_1} \leq s_{j_1} \leq \cdots \leq s_{j_N}$.

Theorem 6.1. *The policy OGB, with non-uniform sizes $\boldsymbol{s}$, weights $\{\boldsymbol{w}_t, t = 1, \dots, T\}$, and step size $\eta = \frac{\sqrt{\frac{C^2}{N\bar{s}^2} + k_C\left(1 - 2\frac{C}{N\bar{s}}\right)}}{w_{\max}\sqrt{TB}}$ has regret upperbounded by $w_{\max}\sqrt{\left(\frac{C^2}{N\bar{s}^2} + k_C\left(1 - 2\frac{C}{N\bar{s}}\right)\right)TB}$.*

We note that if $w_{t,i} = s_i = 1$ for all t and i, then $k_C = C$, and we recover the regret bound from Theorem 3.1. If $C \geq 1/2 \sum_{i=1}^{N} s_i$, the regret constant can be further improved, as indicated in Appendix B.

The OGB caching policy can then be used in the general case where a cache needs to manage items with different sizes and, to the best of our knowledge, is the only policy that retains its sublinear regret guarantees in this setting.

Projection. The projection step at the core of the OGB policy needs to be adapted to reflect the constraints on the item sizes. The main observation we used in Algorithm 2 was that, except for the corner cases, all the components are decreased by the same amount ρ. To understand if we can still find some common simple transformation in the non-uniform size case, we adopt the Lagrangian multiplier methods to solve the optimization problem—details in Appendix C.

As for the uniform size case, the key point is to find the set of items for which the fraction will be set to 0, the set for which it will be set to 1, while the remaining will be modified with a common constant. The solution of the problem shows that the common constant is actually applied to the quantity y_i/s_i, *i.e.*, the vector components normalized by their size, so in Appendix C we provide an algorithm with $\mathcal{O}(N^2)$ complexity for the projection of general real vector $\boldsymbol{y}$. Such a complexity is achieved with the use of two ordered data structures, one with y_i/s_i to check which coefficients need to be set to 0, and the other with $(y_i - 1)/s_i$ to check which ones will be set to 1. We note that this algorithm may be of general interest beyond our specific application to caching.

OGB updates the projection after each request. The vector $\boldsymbol{y}_t$ to be projected has been obtained by summing a positive constant to a component of the feasible vector $\boldsymbol{f}_t \in \mathcal{F}$, in particular to the component j corresponding to the requested item. In this case, there can be at most one component to map to 1. In this case, the complexity of our general algorithm reduces to $\mathcal{O}(N)$. Moreover, since the quantity to remove is the same for all the normalized components y_i/s_i, we can still keep track of the total amount to subtract with $\mathcal{O}(\log N)$ amortized complexity.

We modify Algorithm 2 as follows. First, the ordered data structure $\boldsymbol{z}$ maintains the positive normalized components $\tilde{f}_i/s_i$. Second, the excess to be removed becomes $\eta' = \eta s_j$, *i.e.*, the OGB step size multiplied by the size of the requested item and a new variable keeps track of the partial sum of the squared sizes of the items in $\boldsymbol{z}$. This allows us to easily compute the portion of the excess to be removed equally from all the positive components $\eta'/\sum_{i|z_i\in\boldsymbol{z}} s_i^2$. Finally, in the corner cases (the components go below zero or remain above one) the excess is adjusted to reflect the removal of the items from $\boldsymbol{z}$ considering the size of the removed items. The modified algorithm can be found in Appendix D.

Sampling. The sampling scheme considers the values of f_i as probabilities, and we can still use the approach based on the permanent random numbers, *i.e.*, an item is selected if $f_i > p_i$, where p_i is the permanent random number associated to item i. Since we maintain a data structure with the values of $\tilde{f}_i/s_i$ and we update the value of ρ at each iteration, we have that $f_i/s_i = \tilde{f}_i/s_i - \rho$ and we put the item in the sample if $\tilde{f}_i/s_i - \rho > p_i/s_i$, or $\tilde{f}_i/s_i - p_i/s_i > \rho$. The difference $\tilde{f}_i/s_i - p_i/s_i$ remains constant for all items (except the one being requested), so we can store such a difference in a tree data structure, and the eviction in process in Algorithm 3 remains unchanged. Therefore, the sampling update process maintain $\mathcal{O}(\log N)$ complexity per request.

6.3. Fractional caching

In the fractional setting, item selection is straightforward. Since $\sum_{i=1}^{N} f_{t,i}s_i = C$, we cache a fraction $f_{t,i}$ of each item i for which $f_{t,i} > 0$. The main challenge in this case is the computational complexity. While Algorithm 2 updates $\tilde{\boldsymbol{f}}$ and ρ with $\mathcal{O}(\log N)$ complexity per request, the computation of all the components of $\boldsymbol{f}$ has necessarily $\Omega(N)$ complexity. The batched operation lead to $\mathcal{O}(N/B)$ amortized complexity.

7. Experiments

The low complexity of our caching policy OGB makes it a practical candidate for real-world applications. Furthermore, it allows us to test it on traces with millions of requests and millions of items, a scale that other no-regret policies have not been tested on, likely due to the impracticality of doing so in a reasonable time. We consider a set of *representative* real-world cases. OGB's theoretical no-regret guarantees (Theorem 3.1) should translate into increased robustness to variations in request patterns.

Our main reference for comparison is the optimal static allocation in hindsight (OPT), considered in the regret definition (1). The only no-regret policy we can evaluate on the traces we consider (assuming items with uniform size) is the FTPL policy with noise added only at the beginning, which also enjoys $\mathcal{O}(\log N)$ complexity. As anticipated, FTPL suffers from high sensitivity to its configuration parameter and has difficulty tracking changes in request patterns. Finally, we also show results for LRU to demonstrate that the OGB achieves hit ratios comparable to commonly used caching policies. We do not compare to any state-of-the-art policies, such as those based on machine learning approaches, for various reasons. While these policies work well for specific traffic patterns, no-regret policies are designed to be robust under arbitrary request sequences. Furthermore, none of these policies provide any theoretical guarantees, so a comparison would not be fair. Finally, the hit ratio used as a reference is that obtained by OPT. Any other policy may also reach such a hit ratio for a given traffic pattern, but no-regret policies achieve this consistently in any scenario.

Unless otherwise stated, OGB and FTPL are configured as required for their theoretical guarantees to hold. In particular, OGB's learning rate η is set according to Theorem 3.1. We implemented the eviction policies in Python, using the ordered data structure `SortedDict` from the `sortedcontainers` library, along with the Numpy library.

7.1. Traces

We consider a set of publicly-available block I/O traces from SNIA IOTTA repository [41], a Content Delivery Network (CDN) trace from [15, 32], and a Twitter production cache traces from [16, 42]—see Table 1, the last 4 lines. From the SNIA IOTTA repository, we have considered the most recent traces—labeled as `systor` [14]—along with older traces collected at Microsoft [13], labeled as `ms-ex`. The `cdn` traces refer to a CDN cache serving photos and other media content for Wikipedia pages (21 days, collected in 2019). The `twitter` traces consider their in-memory cache clusters and have been collected in 2020.

Note that the full traces sometimes contain different subtraces, and each subtrace may include hundreds of millions of requests over several days. In our experiments, we consider mostly the subtraces related to Web content or e-mail servers—the only exception is the `systor` trace, which considers storage traffic from a Virtual Desktop Infrastructure. For experimental reproducibility, we report here the details of the subtrace we use. The `ms-ex` is a trace taken from [13] named "Microsoft Enterprise Traces, Exchange Server Traces," which has been collected for Exchange server for a duration of 24-hours—we consider the first 3.5 hours. The `systor` traces [14] collect requests for different block storage devices over 28 days: we consider 12 hours (March, 9th) for the device called "LUN2." The `cdn` trace contains multiple days of traffic, of which we consider portions of 6 hours—we tested different intervals finding similar qualitative results. Finally, for `twitter` we considered the first 20 millions requests from cluster 45. Also in this case, we obtained similar qualitative results for other clusters.

7.2. Results

First, we consider the case where the cache is updated after each request ($B = 1$), leaving the discussion of batched operations ($B > 1$) for the next section. The main metric of interest is the hit ratio, i.e., the ratio between the number of hits and the number of requests. Typically, given a trace, the computation considers the *cumulative* number of hits and requests, which is also what we presented in the results in Sec. 2. While the traffic pattern may change over time, the cumulative representation may smooth out such variability. For this reason, in this section, we present the hit ratio computed over non-overlapping windows of 10^5 requests, i.e., each point in the graph represents the number of hits for the last 10^5 requests, divided by 10^5. Unless otherwise stated, the cache size is set to 5% of the catalog size.

Figure 5 shows the results for the less recent traces, `ms-ex` and `systor`. In both cases, we can observe a highly variable hit ratio over time for the OPT policy. LRU, FTPL and OGB are able to follow such variability. In the `ms-ex` trace, the FTPL and OGB policies take some time to reach the hit ratio of OPT. This behaviour is qualitatively consistent with the fact that their time-average regret progressively vanishes (or even become negative). In particular, OGB achieves at most $\alpha\sqrt{T}$ fewer hits than OPT, where α is a constant defined in Theorem 3.1. In terms of hit ratio, OGB can be at most $\alpha/\sqrt{T}$ below OPT. Thus, as T grows, OGB's hit ratio converges to that of OPT, while for small T a gap may appear. For the `systor` traces, the convergence is faster.

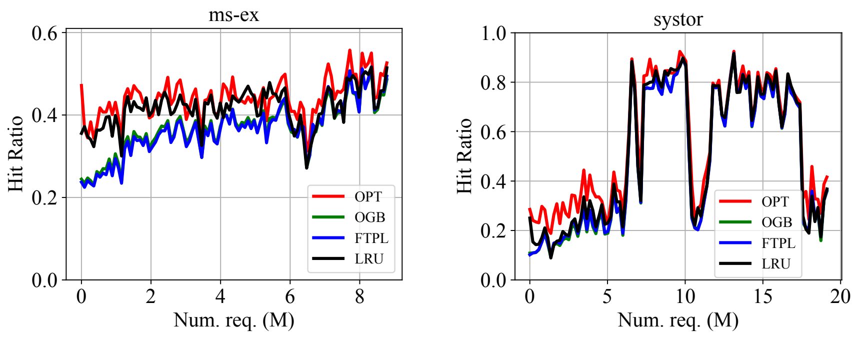

Figure 5: Real-world long (less recent) traces: Performance of OGB policy.

A similar behavior appears in the `cdn` trace (Fig.,6, left). Here, the traffic pattern is much more stable, and OPT significantly outperforms the recency-based LRU, with the two no-regret policies approaching OPT. In contrast, the `twitter` trace exhibits a pattern with a higher level of temporal locality, as revealed by LRU achieving the highest hit ratio. OGB also outperforms OPT, and its consistent performance across diverse traces like `cdn` and `twitter` demonstrates its robustness.

Interestingly, FTPL approaches but does not exceed OPT's performance.

FTPL is essentially a noisy LFU and is thus more suited for stationary traces.[5]

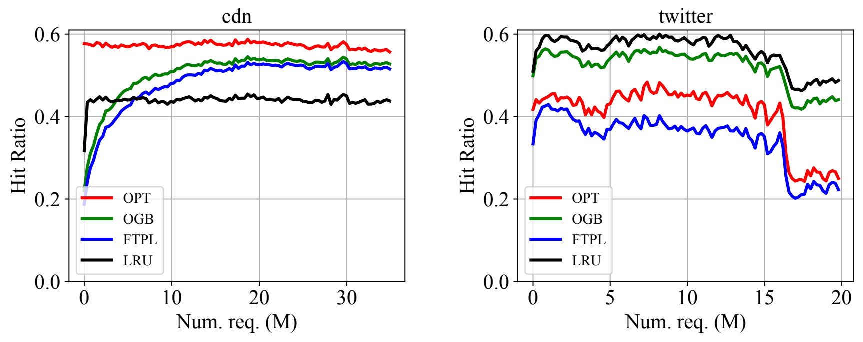

Figure 6: Real-world long (more recent) traces: Performance of OGB policy.

Notice that, in the above figures, the number of requests is shown on the x-axis. From an operational viewpoint, the temporal distribution of such requests has no impact on the complexity of the scheme. Nevertheless, it is interesting to note that the traces include periods during which the arrival rate of requests may change abruptly. Being computationally efficient means that the system can process requests quickly as they arrive. Figure 7 shows the number of requests received per second over time, where time (in hours) is measured from the start of the trace.

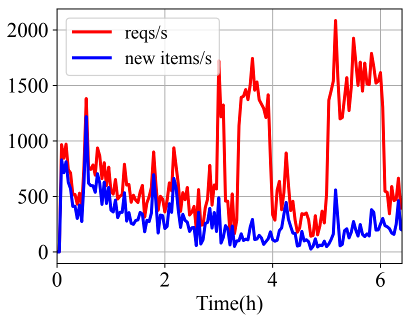

Figure 7: `systor` trace: requests per seconds over time, and how many of them are for items requested for the first time.

Other statistics. The sampling scheme we adopt does not guarantee to have exactly C items in the cache. To understand the variability in the number of stored items, we record at regular intervals the cache occupancy. Figure 8, left, shows the percentage of items stored in the cache with respect to the nominal cache size. Since we have traces with different lengths, we normalized the x-axis

[5]We tested FTPL with different values of its parameter ζ and report its best performance in Fig.,6.

with respect to the trace length. In all cases, the variability is limited to 0.5% of the cache size. Therefore, it is still possible to use OGB under hard cache constraints as far as we set the expected cache size to be slightly inferior to the actual storage available.

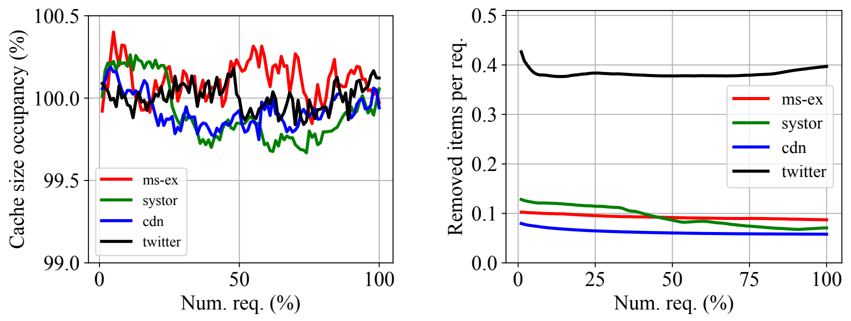

Figure 8: Cache occupancy over time (left) and average removed items per request (right).

Figure 8, right, shows the average number of items that are removed from $\tilde{\boldsymbol{f}}$ in Algorithm 2 (lines 11–18) over windows of 10^5 requests. In all cases we are below 0.5 removed items per requests, confirming the theoretical analysis at the end of Sec. 4.

Running times. By design, OGB operates with $\mathcal{O}(\log N)$ complexity. While the improvement over the linear complexity of OGB_{cl} is evident, it is interesting to evaluate its actual runtime in comparison to widely used constant complexity policies, such as LRU. As noted in Sec. 1, regret guarantees come at a computational cost, but this cost may not be prohibitive.

To evaluate the runtime as a function of the catalogue size N, we generate a set of traces with specific characteristics. In particular, we aim to maintain a similar empirical popularity distribution and a consistent number of requests. Starting with a complete trace—here, we focus on the `cdn` trace due to its regular pattern, which facilitates a straightforward item selection and sampling process—we sample N_i items (*e.g.*, 1000, 5000, ...) and create a subtrace with T^s requests.

Figure 9 (left) compares the runtime of OGB and OGB_{cl} on a commodity server, using a trace with $T^s = 1$ million requests and a catalogue size of up to 100,000 items. The values presented are averages from five trials. Both schemes were implemented in Python, with OGB_{cl} utilizing Numpy arrays and Numpy functions. Regardless of the absolute runtime values, the linear growth observed in OGB_{cl} makes it impractical for large-scale use.

Figure 9 (right) compares OGB with FTPL and LRU. As the catalogue size increases by two orders of magnitude (from 1,000 to 100,000 items), the rise in processing time for OGB remains limited. The gap between OGB, LRU, and FTPL is due to the more complex operations required by OGB. A more efficient implementation (*e.g.*, in C) could reduce such a gap.

Overall, the experiments show that, from a practical standpoint, the loga-

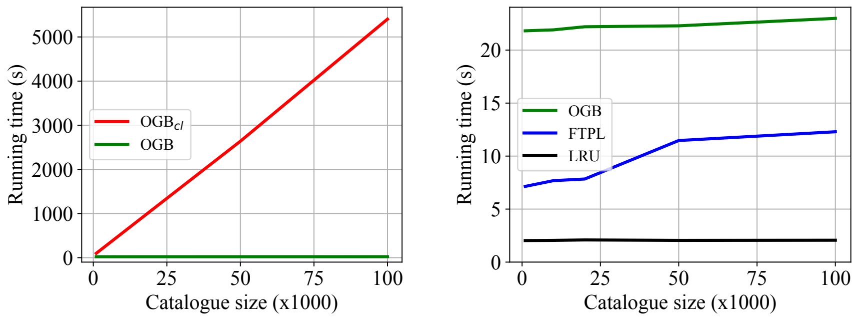

Figure 9: Running time: comparison between OGB and OGB$_{\text{cl}}$ (left), and between OGB and FTPL (right).

rithmic complexity of OGB does not impose a significant burden on the system as the catalogue size grows. The benefits provided by the regret guarantees are achieved with only a modest increase in running time.

7.3. Batched arrivals and fractional case

We consider now a batched operation and evaluate the impact of B, the number of requests in the batch, on the hit ratio for the two most recent traces, `cdn` and `twitter`. We perform our experiments both in the integral and in the fractional setting. The hit ratios in the two cases are practically indistinguishable. This confirms the intuition that, for large catalogs and cache sizes, storing fractions of the items or using the fractions as probabilities to sample the items. We report then only the results for the fractional setting in Fig. 10.

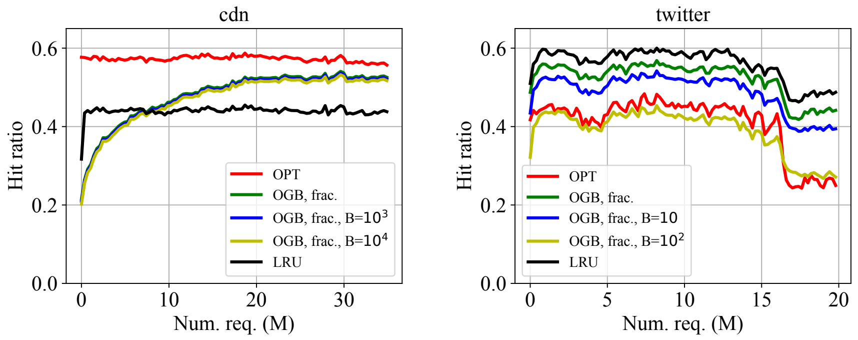

Figure 10: Real-world long traces: Performance of OGB policy in the fractional case for variuos values of the batch size B.

For the `cdn` trace, large values of B do not affect the performance, while for the `twitter` trace even small batches of 100 requests have a significant impact on the hit ratio. The main reason lies in the temporal locality of the requests, *i.e.*, how concentrated are the requests for each item. If the requests are highly concentrated, *e.g.*, a burst of few requests for the same item in less than B consecutive requests, then batching the requests together affects negatively the hit ratio. The cache, in fact, is updated after the batch, but that item is not requested in the following batches. Clearly for popular items that are continuously requested this effect does not appear. But the presence of many less popular items requested in burst has a significant impact.

We have analyzed in detail the `twitter` trace in Appendix E and observed that there are a set of items that are requested in short bursts but still account for up to 20% of the hit ratio. In the `cdn` trace, items with similar request patterns have much less influence, which explains why the results are less affected by B.

Overall, in the fractional case, since the computational complexity scales as $\mathcal{O}(N/B)$, the parameter B helps reduce computational cost, though potentially at the expense of hit ratio, since the cache contents may not be refreshed frequently enough. The policy administrator can thus select the appropriate configuration depending on system priorities.

7.4. Non-uniform size items

The most recent traces, `twitter` and `cdn`, include information about item sizes. In the `twitter` trace, the size distribution exhibits limited variability, with 75% of items ranging between 140 and 160 bytes. In contrast, the `cdn` trace shows a much broader range of item sizes, varying by more than two orders of magnitude—see Fig. 11 left. Therefore, we present the representative results from the `cdn` trace.

The optimal static allocation (OPT) for this case can be framed as a Knapsack Problem. Each item is characterized by a *value* (the number of requests it received) and a *volume* (its size), with the total volume constrained by the cache size C. As a reference, we consider the fractional setting, where fractions of items can be selected. For this case, a polynomial-time greedy algorithm exists to determine the optimal set of items. The algorithm works by sorting items based on their *density*, *i.e.*, the ratio between the value and the size, and selecting full items, starting from the highest density. The only exception is the last item, for which only a fraction is selected to fill the remaining capacity in the knapsack.

Regarding other policies for comparison, LRU may perform poorly in the case of non-uniform item sizes. Therefore, in addition to LRU, we include the Greedy Dual Size (GDS) policy [3], which is specifically designed to account for item size and has logarithmic complexity, like OGB. We are not aware of any extension of FTPL to handle non-uniform sizes, so we cannot provide results for that policy. When proving regret guarantees, the appropriate amount of noise to add is determined, and such a value is not available for the non-uniform size case.

Figure 11 (center) confirms the observations made for the uniform case: OGB closely approaches OPT, outperforming basic LRU, and achieving performance similar to GDS, which has comparable computational complexity but lacks regret guarantees. As for the cache occupancy, the variability remains below 1% of the cache size, indicating that the soft cache constraint does not impose a significant burden on the system.

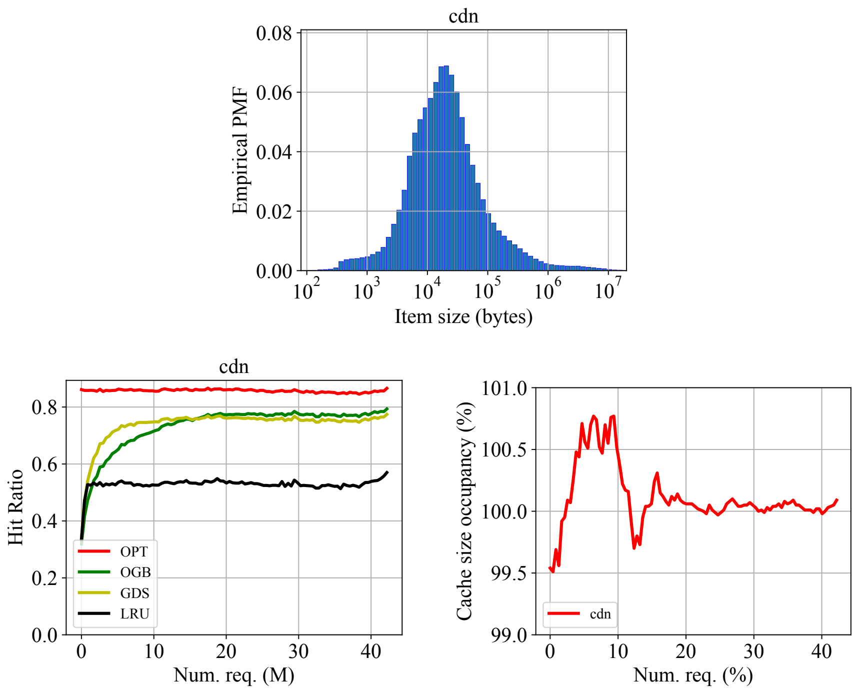

Figure 11: Non-uniform size case, `cdn` trace: Empirical distribution of the sizes (top), performance of the OGB policy compared to other caching policies (bottom, left), and cache occupancy over time (bottom, right).

8. Related work

The caching problem has been extensively studied in the literature. Since our work concern the efficient implementation of a policy with specific characteristics, we discuss here the literature considering these aspects.

Well known caching policies, such as LRU, LFU [2], FIFO [43, 44] and ARC [45] are widely adopted due to their constant complexity: at each new requests they update the cache in $\mathcal{O}(1)$ steps. In some contexts where the arrival rate is extremely high and the speed of the cache is a key issue, these policies are the only viable solution. The price paid is represented by their performance in terms of hit ratios. These policies work well with specific traffic patterns, but do not provide regret guarantees.

Another class of policies, which includes GDS [3], trade computational efficiency for often higher hit ratios (GDS has $\log C$ time complexity). Although they are able to adapt to different contexts, they do not provide regret guarantees. This is true also for policies that are based on Machine Learning tools to predict future request, and therefore improve the performance [4]. Such predictions are based on past observations, and no guarantees are provided.

Inspired by the online optimization framework, policies that are able to provide theoretical guarantees on their performance without any assumption on the arrival traffic pattern have been proposed recently. Most of the works consider the fractional case: in this scenario, as discussed in Sec. 6.3, there is an intrinsic linear complexity, so all the solutions have at least $\mathcal{O}(N)$ complexity. This is the case for example of the policies in [29]—which showed that OGB_{cl} in [6] may be implemented with $\mathcal{O}(N)$ complexity—as well as those in [11]. In the integral setting, it is common to sample the cache content starting from a fractional solution computed by a gradient method. A naive implementation leads then to $\mathcal{O}(N)$ complexity, as discussed in Sec. 4.1. Another possibility is to use a variant of the FTL approach [46]. While the original proposal has $\mathcal{O}(N \log N)$ complexity [7], these variant that associates a single initial noise [46, 10, 47] have $\mathcal{O}(\log N)$ complexity. These works, despite using a low complexity scheme, provide experimental results for traces with at most 10^4 items and 10^5 requests, which may not capture the complexity of real-word request patterns. Our experiments with longer traces and larger catalogs highlight some limits of FTPL.

Differently from all the above works, our solution is computationally efficient, with $\mathcal{O}(\log N)$ complexity, and, since it is based on a robust online gradient approach, can be used in real-world scenarios. Our solution combines low complexity and robustness to different request patterns, making it a practical caching policy.

Bura *et al.* [48] demonstrate that LFU achieves $\mathcal{O}(1)$ regret in the stochastic arrival setting. They also introduce a change detection mechanism to address changes in the popularity distribution, although no regret guarantees are provided for this scenario. In contrast, our work focuses on the adversarial setting, where LFU incurs linear regret [49]. Unlike their approach, we do not make any assumptions about the request arrival distribution, allowing our proposed scheme to naturally handle changing popularity distributions without requiring additional modifications.

Another important performance metric for studying caching under adversarial requests is the competitive ratio, first introduced by Sleator and Tarjan [50]. The competitive ratio is defined as the worst-case ratio between the cost incurred by an online algorithm and that incurred by the optimal offline algorithm, which has full knowledge of the request sequence in advance. An algorithm is said to be α-competitive if its competitive ratio is bounded by α across all possible input sequences. For instance, both LRU and GDS achieve a competitive ratio of C [50, 3]. The problem introduced in [50] was later generalized as the k-server problem [51], and further extended to the framework of metrical task systems (MTS) [52]. Andrew et al. [53] provide a formal comparison between the competitive ratio and the regret, highlighting an inherent trade-off: no algorithm can achieve both sublinear regret and a constant competitive ratio.

9. Conclusions and perspectives

No-regret caching policies are able to offer performance guarantees with no assumption on the arrival traffic pattern. Their computational complexity has limited their application to small scale, unrealistic experiments, where the requested items belongs to a limited catalog.

In this paper we provide a new variant of the gradient-based online caching policy. We coupled the maintenance of the caching probabilities for each item with a sampling scheme that provides coordinated samples, thus minimizing the replacement in the cache. Thanks to its low complexity, we were able to test the online gradient based caching policy in real-world cases, showing that it can scale to long traces and large catalogs.

As future work, we will explore the possibility of introducing correlation in the item sampling scheme to further reduce the variability of the instantaneous cache occupancy. In addition, we will investigate the FTPL policy in the case of non-uniform sizes to understand whether such a policy with a soft cache constraint has sublinear regret and and how its performance compares to that of OGB.

Acknowledgements

This research was supported in part by the French government, through the 3IA Côte d'Azur Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-19-P3IA-0002, in part by the European Network of Excellence dAIEDGE under Grant Agreement Nr. 101120726, in part by the EU HORIZON MSCA 2023 DN project FINALITY (G.A. 101168816). This work was also partially supported by the European Union (EU) and the Italian Ministry for Universities and Research (MUR), National Recovery and Resilience Plan (NRRP), through the project "CSI-Future (Joint Communication and Sensing: CSI-Based Sensing for Future Wireless Networks) PRIN 2022 PNRR (CUP B53D23023920001)".

References

[1] J. Yang, Y. Zhang, Z. Qiu, Y. Yue, R. Vinayak, Fifo queues are all you need for cache eviction, in: Proceedings of the 29th Symposium on Operating Systems Principles, 2023, pp. 130–149.

[2] D. Matani, K. Shah, A. Mitra, An o (1) algorithm for implementing the lfu cache eviction scheme, arXiv preprint arXiv:2110.11602 (2021).

[3] P. Cao, S. Irani, {Cost-Aware}{WWW} proxy caching algorithms, in: USENIX Symposium on Internet Technologies and Systems (USITS 97), 1997.

[4] L. V. Rodriguez, F. Yusuf, S. Lyons, E. Paz, R. Rangaswami, J. Liu, M. Zhao, G. Narasimhan, Learning cache replacement with CACHEUS, in: 19th USENIX Conference on File and Storage Technologies (FAST 21), 2021, pp. 341–354.

[5] S. Romano, H. ElAarag, A quantitative study of recency and frequency based web cache replacement strategies, in: Proceedings of the 11th communications and networking simulation symposium, 2008, pp. 70–78.

[6] G. S. Paschos, A. Destounis, L. Vigneri, G. Iosifidis, Learning to cache with no regrets, in: IEEE INFOCOM 2019-IEEE Conference on Computer Communications, IEEE, 2019, pp. 235–243.

[7] R. Bhattacharjee, S. Banerjee, A. Sinha, Fundamental limits on the regret of online network-caching, Proceedings of the ACM on Measurement and Analysis of Computing Systems 4 (2) (2020) 1–31.

[8] D. Paria, A. Sinha, LeadCache: Regret-optimal caching in networks, Advances in Neural Information Processing Systems 34 (2021) 4435–4447.

[9] N. Mhaisen, G. Iosifidis, D. Leith, Online caching with optimistic learning, in: 2022 IFIP Networking Conference (IFIP Networking), IEEE, 2022, pp. 1–9.

[10] N. Mhaisen, A. Sinha, G. Paschos, G. Iosifidis, Optimistic no-regret algorithms for discrete caching, Proceedings of the ACM on Measurement and Analysis of Computing Systems 6 (3) (2022) 1–28.

[11] T. Si Salem, G. Neglia, S. Ioannidis, No-regret caching via online mirror descent, ACM Transactions on Modeling and Performance Evaluation of Computing Systems 8 (4) (2023) 1–32.

[12] S. Shalev-Shwartz, et al., Online learning and online convex optimization, Foundations and Trends® in Machine Learning 4 (2) (2012) 107–194.

[13] S. Kavalanekar, B. Worthington, Q. Zhang, V. Sharda, Characterization of storage workload traces from production Windows servers, in: 2008 IEEE International Symposium on Workload Characterization, IEEE, 2008, pp. 119–128.

[14] C. Lee, T. Kumano, T. Matsuki, H. Endo, N. Fukumoto, M. Sugawara, Understanding storage traffic characteristics on enterprise virtual desktop infrastructure, in: Proceedings of the 10th ACM International Systems and Storage Conference, SYSTOR '17, ACM, 2017, pp. 13:1–13:11.

[15] Z. Song, D. S. Berger, K. Li, A. Shaikh, W. Lloyd, S. Ghorbani, C. Kim, A. Akella, A. Krishnamurthy, E. Witchel, et al., Learning relaxed belady for content distribution network caching, in: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), 2020, pp. 529–544.

[16] J. Yang, Y. Yue, K. Rashmi, A large scale analysis of hundreds of in-memory cache clusters at twitter, in: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), 2020, pp. 191–208.

[17] F. Faticanti, G. Neglia, Optimistic online caching for batched requests, Computer Networks 244 (2024) 110341.

[18] N. Bansal, N. Buchbinder, J. S. Naor, Randomized competitive algorithms for generalized caching, in: Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing, STOC '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 235–244. doi:10.1145/1374376.1374412.
URL `https://doi.org/10.1145/1374376.1374412`

[19] L. Wang, S. Bayhan, J. Kangasharju, Optimal chunking and partial caching in information-centric networks, Computer Communications 61 (2015) 48–57. doi:https://doi.org/10.1016/j.comcom.2014.12.009.
URL `https://www.sciencedirect.com/science/article/pii/S0140366414003740`

[20] M. Ji, K. Shanmugam, G. Vettigli, J. Llorca, A. M. Tulino, G. Caire, An efficient multiple-groupcast coded multicasting scheme for finite fractional caching, in: 2015 IEEE International Conference on Communications (ICC), 2015, pp. 3801–3806. doi:10.1109/ICC.2015.7248916.

[21] N. C. Fofack, P. Nain, G. Neglia, D. Towsley, Analysis of ttl-based cache networks, in: 6th International ICST Conference on Performance Evaluation Methodologies and Tools, 2012, pp. 1–10.

[22] D. S. Berger, P. Gland, S. Singla, F. Ciucu, Exact analysis of ttl cache networks: the case of caching policies driven by stopping times, SIGMETRICS Perform. Eval. Rev. 42 (1) (2014) 595–596. doi:10.1145/2637364.2592038.
URL `https://doi.org/10.1145/2637364.2592038`

[23] G. Neglia, D. Carra, P. Michiardi, Cache policies for linear utility maximization, IEEE/ACM Trans. Netw. 26 (1) (2018) 302–313. doi:10.1109/TNET.2017.2783623.
URL `https://doi.org/10.1109/TNET.2017.2783623`

[24] D. Carra, G. Neglia, P. Michiardi, Elastic provisioning of cloud caches: A cost-aware ttl approach, IEEE/ACM Transactions on Networking 28 (3) (2020) 1283–1296. doi:10.1109/TNET.2020.2980105.

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

[26] A. O. Al-Abbasi, V. Aggarwal, Ttlcache: Taming latency in erasure-coded storage through ttl caching, IEEE Transactions on Network and Service Management 17 (3) (2020) 1582–1596.

[27] M. Smiley, How we diagnosed and resolved redis latency spikes with bpf and other tools (2022).
URL `https://about.gitlab.com/blog/how-we-diagnosed-and-resolved-redis-latency-spikes/`

[28] W. Wang, C. Lu, Projection onto the capped simplex, arXiv preprint arXiv:1503.01002 (2015).

[29] G. S. Paschos, A. Destounis, G. Iosifidis, Online convex optimization for caching networks, IEEE/ACM Transactions on Networking 28 (2) (2020) 625–638.

[30] H. O. Hartley, Systematic sampling with unequal probability and without replacement, Journal of the American Statistical Association 61 (315) (1966) 739–748.

[31] G. Peyré, M. Cuturi, et al., Computational optimal transport: With applications to data science, Foundations and Trends® in Machine Learning 11 (5-6) (2019) 355–607.

[32] Z. S. et at., Cdn traces (2020).
URL `https://github.com/sunnyszy/lrb`

[33] J. Nocedal, S. J. Wright, Numerical optimization, Springer, 1999.

[34] E. Ohlsson, Sequential poisson sampling, Journal of official Statistics 14 (2) (1998) 149.

[35] Y. Tillé, Sampling and estimation from finite populations, John Wiley & Sons, 2020.

[36] E. Ohlsson, Coordination of samples using permanent random numbers, Business Survey Methods (1995) 153–169.

[37] M. Dehghan, L. Massoulié, D. Towsley, D. S. Menasché, Y. C. Tay, A utility optimization approach to network cache design, IEEE/ACM Transactions on Networking 27 (3) (2019) 1013–1027. doi:10.1109/TNET.2019.2913677.

[38] K. Brewer, L. Early, M. Hanif, Poisson, modified poisson and collocated sampling, Journal of Statistical Planning and Inference 10 (1) (1984) 15–30.

[39] K. R. Brewer, L. Early, S. Joyce, Selecting several samples from a single population, Australian Journal of Statistics 14 (3) (1972) 231–239.

[40] A. Matei, Y. Tillé, Maximal and minimal sample co-ordination, Sankhyā: The Indian Journal of Statistics (2003-2007) 67 (3) (2005) 590–612.
URL `http://www.jstor.org/stable/25053451`

[41] SNIA, Snia iotta repository block i/o traces (2008).
URL `http://iotta.snia.org/traces/block-io`

[42] J. Y. et at., Twitter production cache traces (2020).
URL `https://github.com/twitter/cache-trace`

[43] O. Eytan, D. Harnik, E. Ofer, R. Friedman, R. Kat, It's time to revisit LRU vs. FIFO, in: 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20), 2020.

[44] Y. Zhang, J. Yang, Y. Yue, Y. Vigfusson, K. Rashmi, SIEVE is simpler than LRU: an efficient Turn-Key eviction algorithm for web caches, in: 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), 2024, pp. 1229–1246.

[45] N. Megiddo, D. S. Modha, ARC: A Self-Tuning, low overhead replacement cache, in: 2nd USENIX Conference on File and Storage Technologies (FAST 03), 2003.

[46] S. Mukhopadhyay, A. Sinha, Online caching with optimal switching regret, in: 2021 IEEE International Symposium on Information Theory (ISIT), IEEE, 2021, pp. 1546–1551.

[47] F. Z. Faizal, P. Singh, N. Karamchandani, S. Moharir, Regret-optimal online caching for adversarial and stochastic arrivals, in: EAI International Conference on Performance Evaluation Methodologies and Tools, Springer, 2022, pp. 147–163.

[48] A. Bura, D. Rengarajan, D. Kalathil, S. Shakkottai, J.-F. Chamberland, Learning to cache and caching to learn: Regret analysis of caching algorithms, IEEE/ACM Transactions on Networking 30 (1) (2021) 18–31.

[49] R. Bhattacharjee, S. Banerjee, A. Sinha, Fundamental Limits on the Regret of Online Network-Caching, Proceedings of the ACM on Measurement and Analysis of Computing Systems 4 (2) (Jun. 2020).

[50] D. D. Sleator, R. E. Tarjan, Amortized Efficiency of List Update and Paging Rules, Communications of the ACM 28 (2) (1985) 202–208.

[51] M. Manasse, L. McGeoch, D. Sleator, Competitive Algorithms for On-Line Problems, in: Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, STOC '88, Association for Computing Machinery, New York, NY, USA, 1988, p. 322–333.

[52] A. Borodin, N. Linial, M. E. Saks, An Optimal On-Line Algorithm for Metrical Task System, Journal of the ACM (JACM) 39 (4) (1992) 745–763.

[53] L. Andrew, S. Barman, K. Ligett, M. Lin, A. Meyerson, A. Roytman, A. Wierman, A Tale of Two Metrics: Simultaneous Bounds on Competitiveness and Regret, SIGMETRICS Performance Evaluation Review 41 (1) (2013) 329–330.

[54] M. Zinkevich, Online convex programming and generalized infinitesimal gradient ascent, in: Proceedings of the Twentieth International Conference on International Conference on Machine Learning, ICML'03, AAAI Press, 2003, p. 928–935.

Appendix A. Proof of Theorem 3.1

Proof. We prove the result for a general online convex optimization problem with L-Lipschitz convex cost functions $c_t : \mathcal{F} \to \mathbb{R}$ over the bounded convex set $\mathcal{F} \subset \mathbb{R}^N$ with $\text{radius}(\mathcal{F}) = \min_{\boldsymbol{f}\in\mathcal{F}} \sup_{\hat{\boldsymbol{f}}\in\mathcal{F}} \|\boldsymbol{f} - \hat{\boldsymbol{f}}\|$.

Let $\mathcal{A}$ denote the standard online gradient descent algorithm: it updates the state as follows:

$$\boldsymbol{f}_{t+1} = \Pi_{\mathcal{F}}\left(\boldsymbol{f}_t - \eta \nabla c_t(\boldsymbol{f}_t)\right),$$

and experiences a cost $c_{t+1}(\boldsymbol{f}_{t+1})$.

The algorithm $\mathcal{A}$ enjoys the following (fractional) regret guarantees for $\boldsymbol{f}_0 = \arg\min_{\boldsymbol{f}\in\mathcal{F}} \sup_{\hat{\boldsymbol{f}}\in\mathcal{F}} \|\boldsymbol{f} - \hat{\boldsymbol{f}}\|$ and $\eta = \frac{\text{radius}(\mathcal{F})}{L\sqrt{T}}$ [54]:

$$R_T(\mathcal{A}) \triangleq \sup_{c_0, c_1, \ldots c_{T-1}} \left\{ \sum_{t=0}^{T-1} c_t(\boldsymbol{f}_t) - \min_{\boldsymbol{f}\in\mathcal{F}} \sum_{t=0}^{T-1} c_t(\boldsymbol{f}) \right\} \le \frac{\text{radius}(\mathcal{F})^2}{2\eta} + \frac{\eta L^2 T}{2} = \text{radius}(\mathcal{F}) L\sqrt{T}. \tag{A.1}$$

Now, we consider an algorithm $\mathcal{A}'$ which selects the same state as $\mathcal{A}$ at time slots multiple of B and freezes the state at other time slots, i.e., $\boldsymbol{f}'_t = \boldsymbol{f}_{\ell(t)}$, where $\ell(t) \triangleq \lfloor t/B \rfloor B$.

We compute the difference in the total cost experienced by the algorithms $\mathcal{A}$ and $\mathcal{A}'$.

$$c_t(\boldsymbol{f}'_t) - c_t(\boldsymbol{f}_t) = c_t(\boldsymbol{f}_{\ell(t)}) - c_t(\boldsymbol{f}_t) v \le L\|\boldsymbol{f}_t - \boldsymbol{f}_{\ell(t)}\| \le L \sum_{\tau=0}^{t-\ell(t)-1} \|\boldsymbol{f}_{\ell(t)+\tau+1} - \boldsymbol{f}_{\ell(t)+\tau}\| \tag{A.2}$$

$$\le L \sum_{\tau=0}^{t-\ell(t)-1} \|\Pi_{\mathcal{F}}\left(\boldsymbol{f}_{\ell(t)+\tau} - \eta \nabla c_{\ell(t)+\tau}\right) - \boldsymbol{f}_{\ell(t)+\tau}\| \tag{A.3}$$

$$\le L \sum_{\tau=0}^{t-\ell(t)-1} \|\boldsymbol{f}_{\ell(t)+\tau} - \eta \nabla c_{\ell(t)+\tau} - \boldsymbol{f}_{\ell(t)+\tau}\| \le \eta L^2 (t - \ell(t)). \tag{A.4}$$

Summing over T we obtain

$$\sum_{t=0}^{T-1} c_t(\boldsymbol{f}'_t) - c_t(\boldsymbol{f}_t) \le \eta L^2 \left(\left\lfloor \frac{T-2}{B} \right\rfloor \frac{B(B-1)}{2} + \sum_{t=\ell(T-1)}^{T-1} (t - \ell(t)) \right) \tag{A.5}$$

$$\le \eta L^2 \left(\left\lfloor \frac{T-2}{B} \right\rfloor \frac{B(B-1)}{2} + \frac{(B-1)(T - \ell(T-1))}{2} \right) \tag{A.6}$$

$$\le \eta L^2 \frac{B-1}{2} \left(\left\lfloor \frac{T-1}{B} \right\rfloor B + T - \ell(T-1) \right) \le \eta L^2 T \frac{B-1}{2} \tag{A.7}$$

Then algorithm $\mathcal{A}'$ enjoys the following regret bound for $\boldsymbol{f}_0 = \arg\min_{\boldsymbol{f}\in\mathcal{F}} \sup_{\hat{\boldsymbol{f}}\in\mathcal{F}} \|\boldsymbol{f} - \hat{\boldsymbol{f}}\|$ and $\eta = \frac{\text{radius}(\mathcal{F})}{L\sqrt{TB}}$:

$$R_T(\mathcal{A}') \leq \frac{\text{radius}(\mathcal{F})^2}{2\eta} + \frac{\eta L^2 T}{2} + \eta L^2 T\frac{B-1}{2} = \frac{\text{radius}(\mathcal{F})^2}{2\eta} + \frac{\eta L^2 TB}{2} = \text{radius}(\mathcal{F})L\sqrt{TB}. \tag{A.8}$$

In order to prove the initial statement, we observe that the regret $R_T(\mathcal{A}) \triangleq \sup_{\boldsymbol{r}_0,\boldsymbol{r}_1,\dots,\boldsymbol{r}_{T-1}} \left\{\sum_{t=0}^{T-1}\phi_t(\boldsymbol{x}^*) - \mathbb{E}\left[\sum_{t=0}^{T-1}\phi_t(\boldsymbol{x}_t)\right]\right\}$ coincides with the fractional regret in (1), because of linearity of expectation and the fact OGB samples $\boldsymbol{x}_t$ so that $\mathbb{E}[\boldsymbol{x}_t] = \boldsymbol{f}_t$ (see Algorithm 1). Finally, one can apply (A.8) by considering $c_t(\boldsymbol{f}_t) = \sum_{i=1}^N w_{t,i}r_{t,i}(1-f_{t,i})$ and observing that $\text{radius}(\mathcal{F}) = \sqrt{C\left(1-\frac{C}{N}\right)}$ and $L = 1$. □

Appendix B. OGB with unequal sizes: Proof of Theorem 6.1

Proof. In the proof of Theorem 3.1, (A.8) only requires convexity of the set $\mathcal{F}$ beside convexity and Lipschitzianity of the cost functions $c_t(\cdot)$. As the set $\mathcal{F} = \{\boldsymbol{f} : f_i \in [0,1]\forall i \in [N], \sum_{i=1}^N f_i s_i = C\}$ is convex, then (A.8) holds. We conclude the result by observing that $\|\nabla c_t(\boldsymbol{f}_t)\| \leq \max_{t,i} w_{t,i}$ and bounding the radius of $\mathcal{F}$ as follows.

Let us assume that $s_1 \leq s_2 \leq \dots \leq s_N$. Let $\bar{s} = \sum_{i=1}^N s_i/N$ and $\boldsymbol{f}_0 = f_0\mathbf{1}$, where $f_0 = \frac{C}{N\bar{s}}$. When $C < \frac{1}{2}\sum_{i=1}^N s_i$ (i.e., $f_0 < \frac{1}{2}$), let $k_C \in [N]$ be such that $\sum_{i=1}^{k_C-1} s_i < C \leq \sum_{i=1}^{k_C} s_i$, and define $\boldsymbol{f}'$ by $f'_i = 1$ for $i \leq k_C$ and $f'_i = 0$ otherwise. Then every feasible vector $\boldsymbol{f} \in \mathcal{F}$ satisfies $\|\boldsymbol{f} - \boldsymbol{f}_0\| \leq \|\boldsymbol{f}' - \boldsymbol{f}_0\|$, i.e., $\boldsymbol{f}$ is closer to $\boldsymbol{f}_0$ than $\boldsymbol{f}'$ is. The radius($\mathcal{F}$) can be bounded as follows:

$$\text{radius}(\mathcal{F}) \leq \|\boldsymbol{f}_0 - \boldsymbol{f}'\| = \sqrt{k_C(1-f_0)^2 + (N-k_C)f_0^2} \tag{B.1}$$

$$= \sqrt{Nf_0^2 + k_C(1-2f_0)} \tag{B.2}$$

$$= \sqrt{\frac{C^2}{N\bar{s}^2} + k_C\left(1 - 2\frac{C}{N\bar{s}}\right)}. \tag{B.3}$$

When $C \geq \frac{1}{2}\sum_{i=1}^N s_i$ (i.e., $f_0 \geq \frac{1}{2}$), let $\tilde{k}_C \in [N]$ be such that $\sum_{i=N-\tilde{k}_C+1}^N s_i \leq C < \sum_{i=N-\tilde{k}_C}^N s_i$, and define $\boldsymbol{f}''$ by $f''_i = 1$ for $i \geq N - \tilde{k}_C + 1$ and $f''_i = 0$ otherwise. Then every feasible vector $\boldsymbol{f} \in \mathcal{F}$ satisfies $\|\boldsymbol{f} - \boldsymbol{f}_0\| \leq \|\boldsymbol{f}'' - \boldsymbol{f}_0\|$, i.e., $\boldsymbol{f}$ is closer to $\boldsymbol{f}_0$ than $\boldsymbol{f}''$ is. The radius($\mathcal{F}$) can be bounded as follows:

$$\begin{aligned}\text{radius}(\mathcal{F}) \leq \|\boldsymbol{f}_0 - \boldsymbol{f''}\| &= \sqrt{\tilde{k}_C(1-f_0)^2 + (N-\tilde{k}_C)f_0^2} && \text{(B.4)}\\ &= \sqrt{Nf_0^2 + \tilde{k}_C(1-2f_0)} && \text{(B.5)}\\ &= \sqrt{\frac{C^2}{N\bar{s}^2} + \tilde{k}_C\left(1-2\frac{C}{N\bar{s}}\right)}. && \text{(B.6)}\end{aligned}$$

As $k_C \geq \tilde{k}_C$, the bound in (B.3) always holds. The bound in (B.6) is tighter when $C \geq \frac{1}{2}\sum_{i=1}^{N} s_i$.

□

Appendix C. OGB with unequal sizes: Projection Algorithm

When we consider OGB with unequal sizes $\boldsymbol{s}$ for items, we get the following optimization problem:

$$\min_{\boldsymbol{f}\in\mathbb{R}^N} \frac{1}{2}\|\boldsymbol{f}-\boldsymbol{y}\|^2 \quad \text{s.t.} \quad 0 \leq f_i \leq 1 \quad \forall i \in \{1,\dots,N\}, \quad \sum_{i=1}^{N} f_i s_i = C. \tag{C.1}$$

The Lagrangian function of this problem is:

$$\mathcal{L}(\boldsymbol{f},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\gamma}) = \frac{1}{2}\|\boldsymbol{f}-\boldsymbol{y}\|^2 - \boldsymbol{\alpha}^\top\boldsymbol{f} - \boldsymbol{\beta}^\top(1-\boldsymbol{f}) + \boldsymbol{\gamma}(\boldsymbol{s}^\top\boldsymbol{f} - C) \tag{C.2}$$

where $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ are the the Lagrange multipliers for constraints. Then, we can get the following KKT conditions:

$$\begin{aligned} f_i - y_i - \alpha_i + \beta_i + \gamma s_i &= 0, \quad i = 1,\dots,N\\ f_i &\geq 0, \quad i = 1,\dots,N\\ f_i &\leq 1, \quad i = 1,\dots,N\\ \alpha_i &\geq 0, \quad i = 1,\dots,N\\ \beta_i &\geq 0, \quad i = 1,\dots,N\\ \alpha_i f_i &= 0, \quad i = 1,\dots,N\\ \beta_i(1-f_i) &= 0, \quad i = 1,\dots,N\\ \sum_{i=1}^{N} s_i f_i &= C. \end{aligned} \tag{C.3}$$

As in [28], without loss of generality, we assume that the elements in our optimal solution are in ascending order:

$$0 = f_1 = \dots = f_a < f_{a+1} \leq \dots \leq f_b < f_{b+1} = \dots f_N = 1, \tag{C.4}$$

Considering the elements at different positions in $\boldsymbol{f}$, we have the following equations:

$$\begin{aligned}
0 = f_i = y_i + \alpha_i - \gamma s_i \geq y_i - \gamma s_i, \quad \alpha_i \geq 0, \quad i = 1, \dots, a \\
1 = f_j = y_j - \beta_j - \gamma s_j \leq y_j - \gamma s_j, \quad \beta_j \geq 0, \quad j = b+1, \dots, N \\
0 < f_k = y_k - \gamma s_k < 1, \quad k = a+1, \dots, b,
\end{aligned} \tag{C.5}$$

For $b > a$, by replacing the equalities for f_i in (C.5) in the constraint $\sum_{i=1}^{N} s_i f_i = C$, we get:

$$\gamma = \frac{\sum_{j=b+1}^{N} s_j + \sum_{k=a+1}^{b} s_k y_k - C}{\sum_{k=a+1}^{b} s_k^2}, \tag{C.6}$$

For the value of γ computed in (C.6), the other KKT conditions are satisfied if:

$$\begin{cases} \frac{y_i}{s_i} \leq \gamma, & i = 1, \dots, a \\ \gamma < \frac{y_k}{s_k} < \gamma + \frac{1}{s_k}, & k = a+1, \dots, b \\ \frac{y_j}{s_j} \geq \gamma + \frac{1}{s_j}, & j = b+1, \dots, N. \end{cases} \tag{C.7}$$

It follows that the components to be set to zero are the a components with the smallest values of y_i/s_i and the components to be set to one are the $N - b$ components with the largest values of $(y_i - 1)/s_i$. When all items have the same size, the two orderings coincide, but here we need to consider them separately.

From now on, we assume then that the components of $\boldsymbol{y}$ are ordered so that

$$\frac{y_1}{s_1} \leq \frac{y_2}{s_2} \cdots \leq \frac{y_N}{s_N}, \tag{C.8}$$

and we denote by $\{i_1, i_2, \dots, i_N\}$ a permutation of the indices which satisfies

$$\frac{y_{i_1} - 1}{s_{i_1}} \leq \frac{y_{i_2} - 1}{s_{i_2}} \cdots \leq \frac{y_{i_N} - 1}{s_{i_N}}. \tag{C.9}$$

Given a positive natural number n, we denote by $[n]$ the set $\{1, 2, \dots, n\}$. Let $D \triangleq [N] \setminus ([a] \cup \{i_{b+1}, i_{b+2}, \dots, i_{b_N}\})$. For $b > a$ and $[a] \cap \{i_{b+1}, i_{b+2}, \dots, i_{b_N}\} = \emptyset$, the KKT conditions become

$$\begin{cases} \gamma = \frac{\sum_{j=b+1}^{N} s_{i_j} + \sum_{k \in D} s_k y_k - C}{\sum_{k \in D} s_k^2}, \\ \frac{y_i}{s_i} \leq \gamma, & i = 1, \dots, a \\ \gamma < \frac{y_k}{s_k} < \gamma + \frac{1}{s_k}, & k \in D \\ \frac{y_{i_j}}{s_{i_j}} \geq \gamma + \frac{1}{s_{i_j}}, & j = b+1, \dots, N. \end{cases} \tag{C.10}$$

A pair of indices (a, b) satisfying the conditions (C.10) correspond to the optimal projection. In particular, it is sufficient to verify the following conditions:

$$\begin{cases} \gamma = \frac{\sum_{j=b+1}^{N} s_{i_j} + \sum_{k \in D} s_k y_k - C}{\sum_{k \in D} s_k^2}, \\ \frac{y_a}{s_a} \leq \gamma, \quad \frac{y_{a+1}}{s_{a+1}} > \gamma, \\ \frac{y_{j_{b+1}} - 1}{s_{j_{b+1}}} \geq \gamma, \quad \frac{y_{j_b} - 1}{s_{j_b}} < \gamma. \end{cases} \tag{C.11}$$

If $b = a$, (C.6) is ill-defined. In this case the constraint $\sum_{i=1}^{N} s_i f_i = C$ becomes $\sum_{i=a+1}^{N} s_i = C$, and it is sufficient to verify the following conditions:

$$\begin{cases} \sum_{i=a+1}^{N} s_i = C, \\ \frac{y_a}{s_a} \leq \frac{y_{i_{a+1}}-1}{s_{i_{a+1}}}. \end{cases} \tag{C.12}$$

Algorithm 4: ProjectionUnequalSizes

```
input  : The components of $\mathbf{y}$ are ordered so that:
         $y_1/s_1 \leq y_2/s_2 \leq \cdots \leq y_N/s_N$.
         The permutation $(i_1, i_2, \ldots, i_N)$ such that:
         $(y_{i_1}-1)/s_{i_1} \leq (y_{i_2}-1)/s_{i_2} \leq \cdots \leq (y_{i_N}-1)/s_{i_N}$
1  for $a = 0,1,\ldots,N$ do
2      if $(C == \sum_{i>a} s_i)$ && $((y_{i_{a+1}}-1)/s_{i_{a+1}} \geq y_a/s_a)$ then
3          Set $b = a$;
4          Set $D = \emptyset$;
5          break;
6      for $b = N, N-1, \ldots, a+1$ do
7          if $[a] \cap \{i_{b+1}, i_{b+2}, \ldots, i_N\} = \emptyset$ then
8              $D = [N] \setminus ([a] \cup \{i_{b+1}, i_{b+2}, \ldots, i_N\}$ ;
9              Compute $\gamma = \frac{\sum_{j=i_{b+1}}^{i_N} s_j + \sum_{k \in D} s_k y_k - C}{\sum_{k \in D} s_k^2}$;
10             if $(\frac{y_a}{s_a} \leq \gamma)$ && $(\frac{y_{a+1}}{s_{a+1}} > \gamma)$&&$(\frac{y_{i_{b+1}}-1}{s_{i_{b+1}}} \geq \gamma)$&&$(\frac{y_{i_b}-1}{s_{i_b}} < \gamma)$
                then
11                 break ;
12 Set $f_i = 0$ for $i$ in $[a]$, $f_j = 1$ for $j$ in $\{i_{b+1}, i_{b+2}, \ldots, i_N\}$, and
   $f_k = y_k - \gamma s_k$ for $k \in D$;
output: $\boldsymbol{f}$
```

Algorithm 4 summarizes the procedure. Lines 2–5 deal with the case $b = a$, lines 6–11 with the case $b > a$. The algorithm needs to check at most $(N+1)(N+2)/2$ combinations for the indices a and b and, as soon as either the conditions in (C.12) or in (C.11) are satisfied, the two indices have been identified and then the projection $\boldsymbol{f}$ is set accordingly. The complexity is $\mathcal{O}(N^2)$, ignoring the initial cost of sorting the vectors $\boldsymbol{y} \oslash \boldsymbol{s}$ and $(\boldsymbol{y} - \mathbf{1}) \oslash \boldsymbol{s}$. The complexity is not influenced by the size of the items, but by the fact that each component must be modified during the projection process.

In our case the vector $\boldsymbol{y}$ is obtained by adding a positive quantity to a vector in $\mathcal{F} = \{\boldsymbol{f} : 0 \leq f_i \leq 1, \forall i = 1, 2, \ldots, N, \sum_{i=1}^{N} s_i f_i = C\}$. Let us denote a vector $\boldsymbol{e}_i = [0\ 0\ 0\ \ldots\ 1\ \ldots\ 0]^\top$, where i means the i-th element in $\boldsymbol{e}_i$ is 1. With this definition, we can make the following assumption for our $\boldsymbol{y}$:

Assumption 1. *The vector $\boldsymbol{y}$ to be projected on $\mathcal{F}$ can be expressed as follows: $\boldsymbol{y} = \boldsymbol{z} + \Delta \boldsymbol{e}_\tau$, for some $\boldsymbol{z} \in \mathcal{F}$, some $\Delta > 0$, and some τ in $[N]$.*

Let us define $\mathcal{F}' = \{\boldsymbol{f} : 0 \leq f_i \leq 1 \forall i \in [N], \sum_{i=1}^{N} s_i f_i \leq C\}$. With this definition and Assumption 1, we have the following lemmas:

Lemma 1. *Under Assumption 1, algorithm 4 yields identical results on $\mathcal{F}$ and $\mathcal{F}'$ for a given $\boldsymbol{y}$. Specifically,*

$$\Pi_{\mathcal{F}}(\boldsymbol{y}) = \Pi_{\mathcal{F}'}(\boldsymbol{y}). \tag{C.13}$$

Proof. With the definitions and assumption, we can prove that:

$$\boldsymbol{f} = \Pi_{\mathcal{F}'}(y) \in \mathcal{F}. \tag{C.14}$$

Suppose that this is not the case, i.e.:

$$\sum_{i=1}^{N} s_i f_i < C \Rightarrow \gamma = 0 \text{ by complementary slackness.} \tag{C.15}$$

Consider the following sets:

$$\begin{cases} \mathcal{Y}_0 = \{i \in [N] : y_i = 0\}, \\ \mathcal{Y}_1 = \{i \in [N] : y_i \geq 1\}, \\ \mathcal{Y} = [N] \setminus (\mathcal{Y}_0 \cup \mathcal{Y}_1). \end{cases} \tag{C.16}$$

From (C.5), it follows that:

$$\begin{cases} \forall i \in \mathcal{Y}_0, f_i \geq y_i, \\ \forall i \in \mathcal{Y}_1, f_i = 1, \\ \forall i \in \mathcal{Y}, f_i = y_i. \end{cases} \tag{C.17}$$

Then,

$$\begin{aligned} \sum_{i=1}^{N} s_i f_i &= \sum_{i \in \mathcal{Y}_0} s_i f_i + \sum_{i \in \mathcal{Y}_1} s_i f_i + \sum_{i \in \mathcal{Y}} s_i f_i \\ &\geq \sum_{i \in \mathcal{Y}_0 \cup \mathcal{Y}} s_i f_i + \sum_{i \in \mathcal{Y}_1} s_i f_i. \end{aligned} \tag{C.18}$$

The only component which may be larger than 1 is the τ-th one.

If $\tau \in \mathcal{Y}_1$, we have:

$$\begin{aligned} \sum_{i \in [N]} s_i f_i &\geq \sum_{i \in [N]} s_i y_i - s_\tau y_\tau + s_\tau \\ &= \sum_{i \in [N]} s_i z_i + \Delta s_\tau - s_\tau z_\tau - \Delta s_\tau + s_\tau \\ &= C + (1 - z_\tau) s_\tau \geq C. \end{aligned} \tag{C.19}$$

If $\tau \in \mathcal{Y}_0 \cup \mathcal{Y}$, we have:

$$\sum_{i\in[N]} s_i f_i \geq C + \Delta s_\tau. \tag{C.20}$$

We can see that both (C.19) and (C.20) contradict (C.15). Finally, Lemma 1 holds. □

Lemma 2. *Under Assumption 1, if the Lagrange multiplier γ corresponding to the constraint $\sum_{i=1}^{N} s_i f_i = C$ equals 0, then $z_\tau = 1$.*

Proof. If $\gamma = 0$, we have (C.19):

$$\sum_{i\in[N]} s_i f_i \geq C + (1 - z_\tau) s_\tau.$$

For $\boldsymbol{f} \in \mathcal{F}$, we need to choose $z_\tau = 1$ so that the constraint is satisfied. □

Lemma 3. *Under Assumption 1, if $z_\tau = 1$, then $\Pi_{\mathcal{F}}(\boldsymbol{y}) = \boldsymbol{z}$.*

Proof. If $z_\tau = 1$, we have $y_\tau = 1 + \Delta$. Then, for any vector $\boldsymbol{f} \in \mathcal{F}$, there is:

$$\|\boldsymbol{f} - \boldsymbol{y}\|^2 \geq (y_\tau - f_\tau)^2 \geq \Delta^2. \tag{C.21}$$

Now $\boldsymbol{f} = \boldsymbol{z}$ achieves the lower bound which means $\boldsymbol{z} = \Pi_{\mathcal{F}}(\boldsymbol{y})$. □

For our $\boldsymbol{y}$, let us consider two cases for γ. First, if $\gamma = 0$, it follows from Lemma 2 that $z_\tau = 1$. In this case, we can obtain the projection directly using Lemma 3, with a computational complexity of $\mathcal{O}(1)$. Conversely, if $\gamma > 0$, then, due to (C.5), only the components of $\boldsymbol{f}$ that are strictly greater than 1 can potentially be mapped to 1. It follows that only the values $b = N - 1$ and $b = N$ need to be tested by algorithm 4, leading to a $\mathcal{O}(N)$ complexity, and $\mathcal{O}(N \log N)$ taking into account the initial sorting steps. However, keeping the vector ordered over multiple consecutive projections only requires $\mathcal{O}(\log N)$ amortized cost. Under fractional caching, the projection and the caching update have $\mathcal{O}(N)$ amortized cost. Under integral caching, projection and ratio updates may be performed jointly with a $\mathcal{O}(\log N)$ amortized cost using the procedure described in Sections 4 and 5.

Appendix D. UpdateProbabilities with non-uniform sizes

Algorithm 5 below adapts the UpdateProbabilities procedure to the non-uniform size case.

Algorithm 5: UpdateProbabilities

```
input: $\tilde{\boldsymbol{f}}$, current state
input: $\rho$, current adjustment
input: $\boldsymbol{z}$, ordered tree with positive
       coeffs. of $\tilde{\boldsymbol{f}}/\boldsymbol{s}$
input: $j$, index of the requested item
input: $\eta$, OGB step size
// The requested item is already 1
1  if $\tilde{f}_j - \rho s_j == 1$ then
2      return;
// The requested item was 0
3  if $\tilde{f}_j == 0$ then
4      $\tilde{f}_j \leftarrow \rho s_j + \eta$;
5      $z_j \leftarrow \rho + \eta/s_j$;
6      $\boldsymbol{z} \leftarrow \boldsymbol{z} \cup \{z_j\}$;
7  else
       // Update with the OGB step
8      $\tilde{f}_j \leftarrow \tilde{f}_j + \eta$;
9      $z_j \leftarrow z_j + \eta/s_j$;
// Remove items with negative
   values
10 $\eta' = \eta s_j$;
11 repeat
12     $\sigma = \sum_{k|z_k \in \boldsymbol{z}} s_k^2$;
13     $\rho' = \eta'/\sigma$;
14     $\mathcal{B} \leftarrow \emptyset$;
15     $\mathcal{B} \leftarrow \{i : z_i - \rho - \rho' < 0\}$ ;
16     $\eta' \leftarrow \eta' - (z_i - \rho)s_i^2, \quad \forall i \in \mathcal{B}$;
17     $\boldsymbol{z} \leftarrow \boldsymbol{z} \setminus \{z_i : i \in \mathcal{B}\}$ ;
18     $\tilde{f}_i \leftarrow 0, \quad \forall i \in \mathcal{B}$ ;
19 until $\mathcal{B}$ is empty;
// Check the value of the requested
   item
20 if $(z_j \in \boldsymbol{z})$ and $(z_j - \rho - \rho' > 1/s_j)$ then
21     $\eta' = \eta s_j - ((z_j - \rho) - 1/s_j)\, s_j^2$;
22     $\boldsymbol{z}, \tilde{\boldsymbol{f}} \leftarrow$ RestoreRemoved();
23     $\boldsymbol{z} \leftarrow \boldsymbol{z} \setminus \{z_j\}$ ;
24     $\tilde{f}_j \leftarrow -1$ ;
25     GoTo line 11 ; // This can happen
        only once
// Update $\rho$
26 $\rho \leftarrow \rho + \rho'$;
27 if $z_j \notin \boldsymbol{z}$ then
28     $z_j \leftarrow 1/s_j + \rho$;
29     $\boldsymbol{z} \leftarrow \boldsymbol{z} \cup \{z_j\}$ ;
30     $\tilde{f}_j \leftarrow 1 + \rho s_j$ ;
31 return $\rho, \tilde{\boldsymbol{f}}, \boldsymbol{z}$
```

Appendix E. Analysis of the request temporal locality

In order to understand the impact of the batch size on the performance, we need to analyze the characteristics of the arrival pattern. In particular, we consider the *lifetime* of the items, *i.e.*, the difference between the timestamps of the last and first requests for each item. If we have an infinitely large cache, then each item contributes with a number of hits that is equal to the number of requests minus 1: this is because, in policies like LRU, the first request is always a cold miss, and it is used to bring the item in the cache, and all the following requests generate hits—the actual number of hits depends on the cache size, but with an infinite cache, the item is never evicted, so this represents an upper bound. We then sort the items by their lifetimes, and cumulatively compute the hit ratio they generate.

Figure E.12 (left) shows that the set of items with lifetime smaller than 100 requests (recall that the timestamp is actually determined by the progressive number of received requests, independently from which item is requested) for the `twitter` trace accounts for almost 20% of the hit ratio. Therefore, if a batch size is bigger than the item lifetime, that item will not generate any hit, because its set of requests is absorbed by the batch. By comparison, in the `cdn` trace, item have a much larger lifetime, *i.e.*, they are continuously requested.

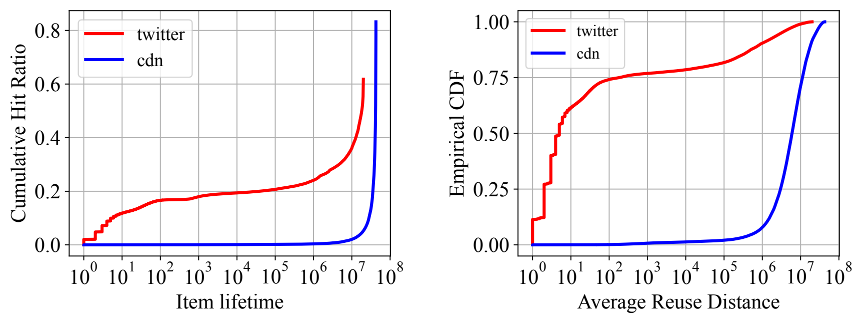

Figure E.12: Cumulative (maximum) hit ratio provided by items sorted by their lifetime (left), and empirical CDF of the reuse distance (right).

For both traces, we have also computed the average *reuse distance*, *i.e.*, the average timestamp difference between two consecutive requests for an item. In Fig. E.12, right, we show the empirical Cumulative Distribution Function (CDF) of the reuse distance. In the `cdn` trace, the reuse distance for most of the items is large: this, combined with the large lifetime, indicates that the items are regularly requested throughout the whole trace, which represents the ideal scenario for a scheme that collects batches of requests. On the contrary, in the `twitter` trace a large percentage of items requested with a small reuse distance, which favors recency-based caching schemes, but limits the benefit of grouping the requests in batches.