Caching with Dual Costs

Anirban Dasgupta
IIT Gandhinagar
anirbandg@iitgn.ac.in

Ravi Kumar
Google
ravi.k53@gmail.com

Tamás Sarlós
Google
stamas@gmail.com

ABSTRACT

Caching mechanisms in distributed and social settings face the issue that the items can frequently change, requiring the cached versions to be updated to maintain coherence. There is thus a trade-off between incurring cache misses on read requests and cache hits on update requests. Motivated by this we consider the following dual cost variant of the classical caching problem: each request for an item can be either a read or a write. If the request is read and the item is not in the cache, then a read-miss cost is incurred and if the request is write and the item is in the cache, then a write-hit cost is incurred. The goal is to design a caching algorithm that minimizes the sum of read-miss and write-hit costs. We study online and offline algorithms for this problem.

For the online version of the problem, we obtain an efficient algorithm whose cost is provably close to near-optimal cost. This algorithm builds on online algorithms for classical caching and metrical task systems, using them as black boxes. For the offline version, we obtain an optimal deterministic algorithm that is based on a minimum cost flow. Experiments on real and synthetic data show that our online algorithm incurs much less cost compared to natural baselines, while utilizing cache even better; furthermore, they also show that the online algorithm is close to the offline optimum.

1. INTRODUCTION

The concept of caching is classical. A cache is a fixed and limited memory that can store items for rapid future accesses. In the caching setting, requests for reading items arrive in an online manner, and once the item is fetched (from a slow source, say, the disk or the network) there is an option of storing this item in cache to rapidly serve possible future requests. Caching has ever remained a powerful paradigm in many areas of computing, ranging from chips and mobile devices to Web servers and search engine results. The success of caching largely depends on the temporal locality properties of the requests. The main technical challenge in caching involves deciding which items to keep in cache at any point in time and this problem has been studied for more than half a century [7]. The notion of competitive analysis [9], despite its shortcomings, has been the theoretical foundation upon which many caching algorithms have been studied.

The traditional view of caching has largely focused on a single computing device and a workload that is read-heavy. With the advent of large-scale distributed systems, and with factors such as the main memory becoming cheaper, the network becoming faster, and applications starting to run on multiple servers, the notion of distributed caching has become more attractive. In this setting, the cache spans across a network of machines. Distributed caching is highly scalable and popular, however, it works well if the workload mostly consists of reads. This holds in applications such as Web servers and Web results where the content does not change much and most accesses to the content are read accesses. To handle the occasional change, distributed cache systems have developed elaborate policies for evictions and expiration that decide how long an item can stay in a cache before it is declared stale.

In this work we consider the increasingly common scenario of workload in which there are a lots of writes interspersed with reads, i.e., workloads that are not predominantly reads. To appreciate such settings, consider the following three examples:

(i) In a large distributed database such as PNUTS [12], applications can query and update different records at different rates, which are often skewed, and the system uses a notification mechanism to signal the updates. Given the skew in access pattern, a careful cache management becomes critical to systems performance. To utilize the cache most efficiently, the system must take into account the relative frequencies of reads and writes. (Note that this problem is also relevant to many pub-sub systems.)

(ii) Consider a social network such as Facebook or Twitter and the status updates by individual users in the network. Depending on how frequently a user accesses her network and the status update rate of her friend, it makes sense to cache the friend's status (or not). This becomes particularly important if the friend happens to be a popular individual (i.e., a node with a large indegree).

(iii) In a collaborative editing application such as Google Docs or DropBox, consider a document that is shared among thousands of users (e.g., a policy document in a company). Typically, these documents are read by many users but updated by only a few users. If the updates are very frequent, it may be better off not caching the document on the client device.

In all these cases, it is not clear if frequently updated items are worth caching, since invalidating them, especially if there are several copies, increases the execution time. Moreover, caching such items has the secondary effect of poor utilization of the cache since they take up space that could have been used more judiciously. This has performance implications especially for devices with limited cache such as sensors or routers or mobile devices. Hence there is a need to trade-off the benefits of caching an item arising from

©2017 International World Wide Web Conference Committee (IW3C2), published under Creative Commons CC BY 4.0 License.
WWW 2017, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/3041021.3054187

reads against the frequency of its updates. The question then becomes how to quantify this precisely and in a formal manner.

Our contributions. In this paper we formulate the dual cost caching problem, which we also call the *read-write caching* problem. In this formulation, each request for an item can be either a read or a write, along with a read-miss cost or a write-hit cost. If the request is a read and the item is not in the cache, then a read-miss cost is incurred and if the request is write and the item is in the cache, then a write-hit cost is incurred. As in the classical read-only version of caching, the goal is to design online and offline caching algorithm that minimizes the sum of read-miss and write-hit costs.

For the online version of the read-write caching problem, we obtain a simple algorithm with provable performance guarantees. This algorithm carefully combines two black boxes, namely, a generic algorithm for the classical (i.e., read-only) caching problem and a generic algorithm for a two-state metrical task system (MTS). Roughly speaking, it maintains a cache by utilizing the decisions of the classical caching, combined with the decisions of the MTS algorithm, which is run for each item in the cache. We show that the competitive ratio of our online algorithm is the sum of that of these two black boxes. While the form of this bound appears deceptively simple, establishing it formally involves subtle arguments, owing to the online nature of the problem. A key practical point is that since our algorithm uses prevailing algorithms as black boxes, using our algorithm in an existing caching system is easy.

Next, for the offline version of the read-write caching problem, we obtain an optimal deterministic algorithm. Our algorithm can be thought of as a true generalization of Belady's famous "evict furthest read item" algorithm [7] that works in the read-only case. We illustrate that naive generalizations of Belady's policy can be sub-optimal for the read-write case. We then show that by using appropriate transformations and graph gadgets, one can obtain a minimum-cost flow instance for a workload, solving which would optimally solve the offline version of the read-write caching problem. Developing the offline optimal algorithm becomes important to gauge how well the online algorithm does on workloads.

Finally, we conduct several experiments on both real and synthetic workloads to demonstrate the effectiveness of our algorithms. Our experiments show the following. First, the online algorithm for read-write caching incurs far less cost than natural baselines such as read-only caching algorithms made to work with modified costs to take both reads and writes into account; they also utilize the cache far more effectively. Second, the performance of the online algorithm is close to that of the offline algorithm, which is optimal.

2. RELATED WORK

Caching is a well-studied problem, from both theoretical and applied points of view. On the theoretical front, caching has been studied mostly using the competitive analysis framework; see the book by Borodin and El-Yaniv [9] for details on competitive analysis. On the practical front, there have been several developments in the systems community, on the conceptual and the implementation aspects. In particular, for caching in the context of the World Wide Web, see the book by Rabinovich and Spatscheck [29].

There have been several practical variants of the basic caching paradigm that take oddities of specific settings into account. This includes distributed caching in which the cache spans across multiple machines, which makes updates challenging [28]. As stated earlier, updates are handled in distributed caches using expirations. Snoopy cache [30], on the other hand, uses bus sniffing to maintain cache coherence in the presence of updates; this works only if there is a shared bus. None of these variants adopts a principled approach to handling writes. They rely on mostly time-based heuristics to decide when to evict an item from the cache. Similar practical strategies for maintaining cache coherency in a distributed setting are in wide use through file systems, e.g., Andrew's file system (AFS), Sprite and Sun's Networked File System (NFS) [31]. Our work could be seen as a first attempt to cast this problem in the formal setting of competitive analysis.

Caching has been extensively studied in the context of the World Wide Web, in terms of content as well as search results. A typical use of caching in search engines is to store the search results for certain queries so that recomputing them, which can be expensive, can be avoided. Lempel and Moran [21] studied the problem of predictive caching and query result prefetching; see also the survey by Lempel [22]. Frances et al. [15] proposed caching strategies for search sites that are geographically distributed. Blanco et al. [8] studied the caching problem over incremental indices; see also the work of Zhang et al. [35] and Long and Suel [23] for caching in the context of inverted lists. Pandey et al. [27] explored nearest-neighbor caching for content-match applications. While all these works stress the importance of caching in multiple web applications, they do not consider the read-write version of the problem. Note that the 'write' issues arise when results go stale or are updated. Several protocols have been proposed to handle this [2, 4, 11]. However, none of these approaches the problem from a competitive analysis viewpoint, and instead approach cache invalidation/expiration mostly as a timestamp-driven mechanism. Practical caching architectures for social networks are indeed more complex than the setting considered here [17]. They also often use other approaches, e.g., utilizing weaker notions of consistency [24], to handle the coherence issue. We leave it as interesting future work to see whether such practical strategies can be analyzed under the same framework as ours.

Two-state metrical task systems have been used in many applications including automated physical design of databases [10, 25, 26], view materialization [16], index selection and tuning [32, 33], and deciding which files to store on flash (solid state disk) [20]. None of these works takes space constraints into account, i.e., the caching aspect. For instance, they assume flash to be infinite. However, the work of [10] is an exception since it provides some heuristics for handling limited space. Our work, on the other hand, is very much driven by the finiteness of the cache and we also seek non-heuristic solutions with provable performance guarantees.

3. PRELIMINARIES

In this section we set up the read-write caching problem. We also provide necessary background on metrical task systems, whose formulation will be useful in our setting.

Let k be the size of the cache and let U be the universe. We assume each item $i \in U$ occupies one cache location, i.e., each item is of unit size. A sequence of requests for items arrive online and each request to an item $i \in U$ is either a *read* or a *write*. If the request is $\langle i, \texttt{read} \rangle$ and the item i is not present in the cache, then we incur a *read-miss* cost. If the request is $\langle i, \texttt{write} \rangle$ and the item i is present in the cache, then we incur a *write-hit* cost. The goal is to design an online caching algorithm to minimize the sum of read-miss costs and write-hits costs; we call this the *rw-caching problem*. The performance of this online algorithm will be evaluated against the best off-line algorithm that is cognizant of the request sequence.

In the most general version of the rw-caching problem, the read-miss costs and write-hits costs can be arbitrary. In particular, in the $(c_{\texttt{read}}, c_{\texttt{write}})$-rw-caching problem, for an item i, the read-miss cost is $c_{\texttt{read}}(i) \geq 0$ and the write-hit cost is $c_{\texttt{write}}(i) \geq 0$. We will

first consider the (1,1)-rw-caching problem, where all the costs are unit; we call this the *unit-cost rw-caching problem*. We will also consider the $(1, c_{\texttt{write}})$-rw-caching problem. Note that in the absence of write requests, the (1,1)-rw-caching problem corresponds to the classical unweighted caching problem and the $(c_{\texttt{read}}, c_{\texttt{write}})$-rw-caching problem corresponds to the classical weighted caching problem.

A request sequence $\sigma = \sigma_1, \sigma_2, \ldots$, consists of read and write requests for items. Let C represent the set of items currently held in the cache, $|C| \leq k$. A caching algorithm $\mathcal{A}$ maintains the contents of C, with the understanding that if $\sigma_j = \langle i, \texttt{read}\rangle$ and $i \notin C$, then it incurs a cost of $c_{\texttt{read}}(i)$ and if $\sigma_j = \langle i, \texttt{write}\rangle$ and $i \in C$, then it incurs a cost of $c_{\texttt{write}}(i)$. Let $\text{cost}_{\mathcal{A}}(\sigma)$ denote the cost of the algorithm $\mathcal{A}$ on request sequence σ, i.e., the sum of its read costs and write costs for each request in σ. A request sequence for an item $i \in U$, denoted $\sigma_{|_i}$ is the restriction of the entire request sequence σ only to item i. Let OPT denote the optimal offline algorithm that knows σ.

DEFINITION 1 (COMPETITIVE RATIO). *A (possibly randomized) online algorithm $\mathcal{A}$ is said to be* (c, b)-competitive *against an oblivious adversary if for all request sequences σ it holds that*

$$\text{E}[\text{cost}_{\mathcal{A}}(\sigma)] \leq c \cdot \text{cost}_{\text{OPT}}(\sigma) + b,$$

where $c > 0, b \geq 0$.

We call $\mathcal{A}$ to be *c-competitive* if $b = 0$.

3.1 Caching algorithms

We assume a generic online algorithm $\mathcal{A}$ for the (classical) caching problem (for example, the algorithm can be LRU for the unit cost case). This algorithm is stateful and supports the following primitive:

- $\mathcal{A}.process(C, i)$: considers a (read) request to item i given that the current cache is C. If $|C| = k$ and $i \notin C$, then it returns `evict` i', with the semantics that $i' \in C$ is the item to be evicted to make room for the item i.

3.2 Metrical task system (MTS)

In the *d-state metrical task system (MTS)* problem, we have d states (in our setting, the state space will be $\{\texttt{in}, \texttt{out}\}$) and the cost of switching from state i to state j is given by the $d \times d$ switching cost matrix $D \in \mathbb{R}^{d \times d}$. It is assumed that $D_{ii} = 0$ and that the D_{ij} values satisfy the triangle inequality, hence the name metrical. At time t the algorithm is presented with a service cost vector $c_t \in \mathbb{R}^d$ representing the costs for each servicing the request from each state. In response, the algorithm chooses state j and pays D_{ij} to switch its state to j from the current state i, and then also pays $c_t(j)$ to service the request. The sequence of service requests is unknown in advance. The goal of the algorithm is to minimize the total cost by deciding a sequence of states to follow. This is a well-studied problem in the online setting [9, 14, 5] and the competitive ratio notion is exactly the same as in the caching case.

As we will later see $d = 2$ states are sufficient in our case. A deterministic 3-competitive algorithm (see, for example, [3]) and randomized 2-competitive algorithms are known for the problem [6].

We assume a generic online algorithm $\mathcal{B}$ for the two-state metrical task system problem (e.g., an algorithm for the ski-rental problem [18]). For notational clarity we will refer to the first state as `in` and the second as `out`. This algorithm is stateful and admits the following primitives:

- $\mathcal{B}.initialize(D)$: this initializes the algorithm with the switching cost matrix $D \in \mathbb{R}^{2\times 2}$; without loss of generality, we can assume that the initial state of the algorithm is `in`.
- $\mathcal{B}.serve(c_{\texttt{in}}, c_{\texttt{out}})$: this serves the request whose costs are $c_{\texttt{in}}$ and $c_{\texttt{out}}$ in states `in` and `out` respectively, and returns the algorithm's new state.

4. AN ONLINE ALGORITHM

In this section we present an online algorithm for the read-write caching problem. We obtain an algorithm by combining appropriate steps of a (generic) classical caching algorithm and a (generic) algorithm for a metrical task system (MTS). The high-level intuition for our combined algorithm presented below is that we can recognize and "weed out" items with high `write` cost residing in the cache of a classical caching algorithm by using an MTS algorithm. Conceptually we run the classical caching algorithm first, and censor its cache with the MTS algorithm in the second step. We first present an algorithm in which the MTS maintains the state for every item; the analysis for this case is considerably simpler. Next we present the more efficient version in which the MTS maintains the state only for the currently cached items.

Algorithm 1 contains the formal description. It uses a "ghost" cache $C_{\mathcal{A}}$, which contains the state of the classical caching algorithm $\mathcal{A}$. The contents of the true cache, C, is determined both by $\mathcal{A}$ and by the decisions of the MTS algorithm. We create an MTS instance for each item in the universe, where the switching cost matrix for an item is given by its respective read and write costs (lines 1–3).

Algorithm 1 Algorithm $\mathcal{D}(\sigma)$ for rw-caching

```
1: for i = 1, 2, ... do
2:     B_i.initialize( 0            c_write(i) )
                     ( c_read(i)    0          )
3: end for
4: C = ∅
5: C_A = ∅
6: for j = 1, 2, ... do
7:     if σ_j = ⟨i, read⟩ then
8:         if A.process(C_A, i) = evict i' then
9:             C = C \ {i'}
10:            C_A = C_A \ {i'}
11:        end if
12:        if i ∉ C then
13:            C = C ∪ {i}
14:            C_A = C_A ∪ {i}
15:        end if
16:        if B_i.serve(0, c_read(i)) = out then
17:            {If we ever get here such that B_i's previous state was
               in, then B_i could be improved by staying in state in.}
18:            C = C \ {i}
19:        end if
20:    else if σ_j = ⟨i, write⟩ and i ∈ C_A then
21:        s = B_i.serve(c_write(i), 0)
22:        if s = out and i ∈ C then
23:            C = C \ {i}
24:        else if s = in and i ∉ C then
25:            {If we ever get here it means that B_i could be improved
               by staying in state out.}
26:            C = C ∪ {i}
27:        end if
28:    end if
29: end for
```

As requests arrive online (line 6), the read-write caching algorithm $\mathcal{D}$ has two cases depending on whether it is a read or a write

request. If the request is a read request for item i, then the algorithm $\mathcal{D}$ consults the classical algorithm $\mathcal{A}$, using the ghost cache $C_{\mathcal{A}}$. If $\mathcal{A}$'s decision is to evict another item i', then item i' is removed from both the ghost cache and the true cache (lines 8–11) and the item i is added to both the caches (lines 12–15). Next, the MTS instance corresponding to item i is invoked with cost vector $(0, \mathrm{c}_{\texttt{read}}(i))$ to see if it is worth storing i in the true cache. If the response of the MTS algorithm is `out`, meaning that MTS is not in favor of storing i in the cache, then i evicted from the true cache C (lines 16–19); note that i is still present in the ghost cache $C_{\mathcal{A}}$.

On the other hand, if the request is a write request for item i and i is present in the ghost cache, then the algorithm $\mathcal{D}$ immediately invokes the MTS instance corresponding to i with cost vector $(\mathrm{c}_{\texttt{write}}(i), 0)$ (line 21). If the response is `out`, meaning MTS thinks it is not worth keeping i in the cache, then i is evicted from the true cache (line 23). Note that as before, i could still be present in the ghost cache. On the other hand, if the response from MTS is `in`, meaning that MTS feels it is worth keeping i in the cache, then i is placed in cache C (lines 24–27).

We next show that Algorithm 1 has a performance guarantee that can be bounded by the guarantees of the classical caching algorithm and MTS algorithm it uses to makes the decisions. As we will see, the online nature of the problem makes it trickier to argue we can get the "best" of both the classical caching and the MTS algorithms.

THEOREM 2. *If $\mathcal{A}$ is an α-competitive caching algorithm and $\mathcal{B}$ is a β-competitive MTS algorithm, then Algorithm 1 is an $(\alpha + \beta)$-competitive algorithm for the rw-caching problem.*

PROOF. Observe that Algorithm 1 maintains the invariant that $i \in C$ if and only if the state of $\mathcal{B}_i$ is `in` and $i \in C_{\mathcal{A}}$. Thus it produces a feasible solution for the rw-caching problem as at most k items are contained in its cache C at any moment. This follows from the facts $C \subseteq C_{\mathcal{A}}$ and $|C_{\mathcal{A}}| \leq k$.

Let $\sigma_{\texttt{read}}$ denote the subsequence of read requests in σ. Similarly let $\sigma_{|i}$ contain the subsequence of read and write requests for item i in σ. Additionally let $\sigma^{\mathcal{A}}_{|i}$ contain the sequence of all `read`s from $\sigma_{|i}$ and those `write`s from $\sigma_{|i}$ when $i \in C_{\mathcal{A}}$. Let $\mathrm{cost}^*_{\text{RO}}(\sigma_{\texttt{read}})$ denote the cost of the optimal offline (read only) caching algorithm when run on the request sequence $\sigma_{\texttt{read}}$ and $\mathrm{cost}^*_{\text{MTS}}(\sigma_{|i})$ be the cost of the optimal offline MTS algorithm when run on $\sigma_{|i}$, the requests for item i. Also let $\mathrm{cost}^*_{\text{RW}}(\sigma)$ denote the cost of the optimal offline rw-caching algorithm when run on the request sequence σ. It clearly holds that

$$\mathrm{cost}^*_{\text{RO}}(\sigma_{\texttt{read}}) \leq \mathrm{cost}^*_{\text{RW}}(\sigma). \tag{1}$$

Now consider *any* MTS algorithm $\mathcal{B}'$ processing the sequence $\sigma_{|i}$. If the jth request is `read` and $\mathcal{B}'$'s state is `out` prior to serving the request, then $\mathcal{B}'$ pays exactly $\mathrm{c}_{\texttt{read}}(i)$ in transition and service costs for serving this `read` independent of the state it chooses to transition to. Similarly, if the jth request is `write` and $\mathcal{B}'$'s state is `in`, then $\mathcal{B}'$ pays precisely $\mathrm{c}_{\texttt{write}}(i)$ in transition and service cost for serving this `write` independent of its next state. If $\mathcal{B}'$ is an *optimal* MTS algorithm, then without loss of generality we can assume that if $\mathcal{B}'$ receives a `read` or `write` request in states `in` and `out` respectively then it stays in the same state and pays 0 in total to serve the request. Thus an optimal MTS algorithm processing $\sigma_{|i}$ faces the same cost structure as an rw-caching algorithm with respect to item i. Therefore we also have that

$$\sum_i \mathrm{cost}^*_{\text{MTS}}(\sigma_{|i}) \leq \mathrm{cost}^*_{\text{RW}}(\sigma).$$

From $\mathrm{cost}^*_{\text{MTS}}(\sigma^{\mathcal{A}}_{|i}) \leq \mathrm{cost}^*_{\text{MTS}}(\sigma_{|i})$ it follows that

$$\sum_i \mathrm{cost}^*_{\text{MTS}}(\sigma^{\mathcal{A}}_{|i}) \leq \mathrm{cost}^*_{\text{RW}}(\sigma). \tag{2}$$

Finally we show that

$$\mathrm{cost}_{\mathcal{D}}(\sigma) \leq \mathrm{cost}_{\mathcal{A}}(\sigma_{\texttt{read}}) + \sum_i \mathrm{cost}_{\mathcal{B}}(\sigma^{\mathcal{A}}_{|i}). \tag{3}$$

Consider the case $\sigma_j = \langle i, \texttt{read}\rangle$. If $i \in C$, then this `read` costs 0 for $\mathcal{D}$. If $i \notin C_{\mathcal{A}}$ (and hence $i \notin C$), then this `read` is a cache miss for $\mathcal{A}$, and both $\mathcal{A}$ and $\mathcal{D}$ pay $\mathrm{c}_{\texttt{read}}(i)$. If $i \in C_{\mathcal{A}}$ and $i \notin C$, then it must have been the MTS algorithm $\mathcal{B}_i$ that removed item i from C. Hence $\mathcal{B}_i$ is in state `out` when σ_j arrives and thus $\mathcal{B}_i$ pays $\mathrm{c}_{\texttt{read}}(i)$ in transition and serving cost independent of its next state. Also note that $\sigma_j \in \sigma^{\mathcal{A}}_{|i}$.

Consider the case $\sigma_j = \langle i, \texttt{write}\rangle$. If $i \notin C$, then this `write` has cost 0 for $\mathcal{D}$. If $i \in C$, then $\mathcal{B}_i$ is in state `in` when σ_j arrives and thus $\mathcal{B}_i$ pays $\mathrm{c}_{\texttt{write}}(i)$ in transition and serving cost independent of its next state. Note that $i \in C_{\mathcal{A}}$ holds as well, and thus σ_j belongs to $\sigma^{\mathcal{A}}_{|i}$.

The claim follows from combining inequalities (1), (2), and (3) with the α and β-competitiveness of Algorithms $\mathcal{A}$ and $\mathcal{B}$. □

Note that Theorem 2 holds both for unit and general read costs, with a weighted caching algorithm $\mathcal{A}$ in the latter case. Also note that since the competitive ratio bound is additive, if $\mathcal{A}$ and $\mathcal{B}$ are both optimal, then $\mathcal{D}$ is within factor two of the optimal.

Arbitrary item sizes. While we have assumed unit-sized items, in the generalized caching problem [1] items also have size $s_i \geq 1$, and the caching constraint is $\sum_{i \in C} s_i \leq k$. We consider a simple modification to Algorithm 1, namely, by inserting the following snippet between line 8 and line 9:

---

if $\exists i'' \in C_{\mathcal{A}} \setminus C : \mathrm{c}_{\texttt{read}}(i') \geq \mathrm{c}_{\texttt{read}}(i'')$ and $s_{i'} \leq s_{i''}$ **then**
  $i' = i''$
end if

Theorem 2 continues to hold unchanged in this case by a minor modification of the analysis, charging for the page outs instead of page ins, establishes a bound similar to Theorem 2 for this variation. We omit the details in this version.

4.1 A stateless version

Note that the MTS in Algorithm 1 is stateful, i.e., it maintains the state for every item including the ones not currently in cache. In a *stateless* variant, the MTS in Algorithm 1 would maintain the state for *only* the items in the cache. If an item is evicted, the MTS for it is deleted and is recreated (with the default `in` state) the next time it is cached. This is clearly more space-efficient than the stateful version.

For the stateless version, it is not difficult to see that it is unlikely that we can obtain the same result as Theorem 2. We outline an informal example. Consider a sequence σ where the subsequence $\sigma_{|i}$ for an item i has few read requests separated by large number of write requests. The optimal MTS stays in state `out` and pays $\mathrm{c}_{\texttt{read}} + \mathrm{c}_{\texttt{write}}$ for handling the write costs at most once. An algorithm that is not maintaining state for an item has to revert back to some default state of the MTS every time the item i is read back into the cache. If we are using the work-function MTS ([3]; see Section 6.1) and the reset-state is, say, $(0, 0)$, then every time there is a new read request, the stateless algorithm is paying an additional $\mathrm{c}_{\texttt{read}} + \mathrm{c}_{\texttt{write}}$ cost for handling the write requests following the read. This happens for each read. Hence, in the case $\mathrm{c}_{\texttt{write}} > \mathrm{c}_{\texttt{read}}$, it is clear the competitive ratio can be unbounded.

The above example suggests that to obtain a bounded competitive ratio, it may be useful to assume $c_{\texttt{write}} \leq c_{\texttt{read}}$. We make this assumption and show an algorithm that uses a particular MTS, namely the ski-rental algorithm [9]. This algorithm maintains the MTS state *only* for items in the cache $C_{\mathcal{A}}$ and can be obtained by making the following two simple changes to Algorithm 1: (i) after line 10, we delete $\mathcal{B}_{i'}$, and (ii) we create a instance of $\mathcal{B}_i$ immediately after line 12. We show that this stateless algorithm is still competitive.

THEOREM 3. *If $c_{\texttt{write}} \leq c_{\texttt{read}}$ and $\mathcal{A}$ is an α-competitive algorithm, and $\mathcal{B}$ is the ski-rental algorithm, Algorithm 1 after the above modification is $(\alpha+5)$-competitive.*

PROOF. Let us denote the stateless algorithm as $\mathcal{L}$. When we are using the ski-rental algorithm as $\mathcal{B}$, then effectively, an item present in the cache C (recall $C \subseteq C_{\mathcal{A}}$) is removed by $\mathcal{B}$ after receiving $\lceil c_{\texttt{read}}/c_{\texttt{write}} \rceil$ write requests to it (but its slot is maintained in $C_{\mathcal{A}}$). Without loss of generality, we assume that items are read only on actual read requests, and all algorithms start with an empty cache.

Let $\sigma_{\texttt{read}}$ denote the subsequence of the input sequence σ consisting of only the read requests. As before, we consider the request sequence $\sigma_{|i}$ for a particular item i. If it was all write requests, then both optimal and $\mathcal{L}$ pay zero. Else, we partition $\sigma_{|i}$ into $\sigma_{|i} = \sigma^1_{|i} \ldots \sigma^k_{|i}$, where for each $j \neq k$, $\sigma^j_{|i}$, consists of a number (possibly zero) of writes and a single read at the end, and $\sigma^k_{|i}$ consists of the last (possibly zero) writes succeeding any read request. There is a natural injective mapping $\tau(i,j)$ from the read requests in $\sigma^j_{|i}$ to the requests in $\sigma_{\texttt{read}}$— denote the request in $\sigma_{\texttt{read}}$ corresponding to $\sigma^j_{|i}$ as $(\sigma_{\texttt{read}})_{\tau(i,j)}$. Let $\text{cost}_{\mathcal{A}}((\sigma_{\texttt{read}})_j)$ denote the cost that $\mathcal{A}$ paid on the specific request $(\sigma_{\texttt{read}})_j$ while processing $\sigma_{\texttt{read}}$.

Recall from the proof of Theorem 2 that for an optimal MTS, for item i, we have

$$\sum_i \text{cost}^*_{\text{MTS}}(\sigma_{|i}) \leq \text{cost}^*_{\text{RW}}(\sigma). \tag{4}$$

Also, $\text{cost}^*_{\text{MTS}}(\sigma_{|i}) = \sum_{j=1}^{k} \text{cost}^*_{\text{MTS}}(\sigma^j_{|i})$.

Note without loss of generality that the optimal MTS does state transitions from `in` to `out` for an item only immediately after a read request, and only in the following two cases: (i) this read is the last read request for the item, or (ii) if ℓ, the number of write requests to this item preceding the next read for it, satisfies $\ell > c_{\texttt{read}}/c_{\texttt{write}}$.

Now, for any $j < k$, if $\mathcal{A}$ did not have i in $C_{\mathcal{A}}$ at the beginning of $\sigma^j_{|i}$, then the read request at the end of $\sigma^j_{|i}$ causes a cache miss. The corresponding request $(\sigma_{\texttt{read}})_{\tau(i,j)}$ must cause a cache miss for $\mathcal{A}$ as well. Hence,

$$\text{cost}_{\mathcal{L}}(\sigma^j_{|i}) \leq \text{cost}_{\mathcal{A}}((\sigma_{\texttt{read}})_{\tau(i,j)}). \tag{5}$$

Else, the algorithm had item i in $C_{\mathcal{A}}$ at the beginning of $\sigma^j_{|i}$ and the MTS $\mathcal{B}_i$ (either newly created at end of $\sigma^{j-1}_{|i}$ or existing) is in state `in` at the beginning of $\sigma^j_{|i}$. This $\mathcal{B}_i$ instance sees only a prefix of writes in the sequence $\sigma^j_{|i}$. This is because item i, once removed from C, is never read back until there is a $\langle i, \texttt{read}\rangle$ request. Suppose $\sigma^j_{|i}$ contains ℓ writes. Note that $\text{cost}^*_{\text{MTS}}(\sigma^j_{|i}) \geq \min(\ell \cdot c_{\texttt{write}}, c_{\texttt{read}})$.

If the item is not evicted from $C_{\mathcal{A}}$ during $\sigma^j_{|i}$, we charge the write costs and the final read cost to the MTS. If $\ell > \lceil \frac{c_{\texttt{read}}}{c_{\texttt{write}}} \rceil$, then

$$\text{cost}_{\mathcal{L}}(\sigma^j_{|i}) = \text{cost}_{\mathcal{B}}(\sigma^j_{|i}) \leq \left\lceil \frac{c_{\texttt{read}}}{c_{\texttt{write}}} \right\rceil c_{\texttt{write}} + c_{\texttt{read}} \leq 3c_{\texttt{read}},$$

while $\text{cost}^*_{\text{MTS}}(\sigma^j_{|i}) \geq c_{\texttt{read}}$. If $\ell \leq \lceil \frac{c_{\texttt{read}}}{c_{\texttt{write}}} \rceil$, then

$$\text{cost}_{\mathcal{L}}(\sigma^j_{|i}) = \ell \cdot c_{\texttt{write}} \leq \text{cost}^*_{\text{MTS}}(\sigma^j_{|i}).$$

Hence in both cases,

$$\text{cost}_{\mathcal{L}}(\sigma^j_{|i}) \leq 3\text{cost}^*_{\text{MTS}}(\sigma^j_{|i}). \tag{6}$$

Next consider the case that the item is actually evicted from $C_{\mathcal{A}}$ before the last read of $\sigma^j_{|i}$. Suppose the item gets evicted from $C_{\mathcal{A}}$ after ℓ'th write, where $\ell' \leq \ell$, then the final read request of $\sigma^j_{|i}$ is handled by $\mathcal{A}$ reading item i into $C_{\mathcal{A}}$. Hence $\mathcal{A}$, on input $\sigma_{\texttt{read}}$, suffers a cache miss for the request $(\sigma_{\texttt{read}})_{\tau(i,j)}$ and pays $\text{cost}_{\mathcal{A}}((\sigma_{\texttt{read}})_{\tau(i,j)})$. We charge the read due to the final request of $\sigma^j_{|i}$ to $\text{cost}_{\mathcal{A}}((\sigma_{\texttt{read}})_{\tau(i,j)})$ and the write costs to the MTS. Thus $\text{cost}_{\mathcal{L}}(\sigma^j_{|i}) = \text{cost}_{\mathcal{B}}(\sigma^j_{|i}) + \text{cost}_{\mathcal{A}}((\sigma_{\texttt{read}})_{\tau(i,j)})$. Arguing similarly as above, when $\ell' > \lceil \frac{c_{\texttt{read}}}{c_{\texttt{write}}} \rceil$, $\text{cost}^*_{\text{MTS}}(\sigma^j_{|i}) \geq c_{\texttt{read}}$ while $\text{cost}_{\mathcal{B}}(\sigma^j_{|i}) = \lceil \frac{c_{\texttt{read}}}{c_{\texttt{write}}} \rceil c_{\texttt{write}} \leq 2c_{\texttt{read}}$. When $\ell' \leq \lceil \frac{c_{\texttt{read}}}{c_{\texttt{write}}} \rceil$, $\text{cost}_{\mathcal{B}}(\sigma^j_{|i}) = \text{cost}^*_{\text{MTS}}(\sigma^j_{|i}) = \ell' \cdot c_{\texttt{write}}$. Hence,

$$\text{cost}_{\mathcal{L}}(\sigma^j_{|i}) \leq 2\text{cost}^*_{\text{MTS}}(\sigma^j_{|i}) + \text{cost}_{\mathcal{A}}((\sigma_{\texttt{read}})_{\tau(i,j)}). \tag{7}$$

Finally, for $\sigma^k_{|i}$, $\mathcal{B}_i$ moves to state `out` after at most $\lceil \frac{c_{\texttt{read}}}{c_{\texttt{write}}} \rceil$ writes. Hence

$$\begin{aligned} \text{cost}_{\mathcal{L}}(\sigma^k_{|i}) = \ \text{cost}_{\mathcal{B}}(\sigma^k_{|i}) \ &\leq \ \left\lceil \frac{c_{\texttt{read}}}{c_{\texttt{write}}} \right\rceil c_{\texttt{write}} \\ &\leq \ 2c_{\texttt{read}} \ \leq \ 2\text{cost}^*_{\text{MTS}}(\sigma_{|i}), \end{aligned} \tag{8}$$

where the last inequality holds since by previous assumption, $\sigma_{|i}$ contains at least one read request.

Note that each segment $j \in \{1, \ldots, k\}$ lies in exactly one of the above cases of (5)–(8). Hence summing over all j and using the bounds in (5)–(8), we have

$$\begin{aligned} \sum_i \text{cost}_{\mathcal{L}}(\sigma_{|i}) \ &\leq \ \sum_i \sum_{j<k} 3\text{cost}^*_{\text{MTS}}(\sigma^j_{|i}) \\ &\quad + \ \sum_i \sum_{j<k} \text{cost}_{\mathcal{A}}((\sigma_{\texttt{read}})_{\tau(i,j)}) + 2\text{cost}^*_{\text{MTS}}(\sigma_{|i}) \\ &\leq \ 5 \sum_i \text{cost}^*_{\text{MTS}}(\sigma_{|i}) + \text{cost}_{\mathcal{A}}(\sigma_{\texttt{read}}), \end{aligned}$$

where the last inequality holds since $\tau(i,j)$ is injective. Using (4) and since $\text{cost}_{\mathcal{A}}(\sigma_{\texttt{read}}) \leq \alpha \text{cost}^*_{\text{RO}}(\sigma_{\texttt{read}})$, we have the claim that the stateless algorithm is $(\alpha+5)$-competitive. □

5. AN OPTIMAL OFFLINE ALGORITHM

In this section we show the optimal offline algorithm for the rw-caching problem. An offline optimal algorithm is important to understand how well the online algorithm works, especially in practice. In the read-only case, the optimal offline policy is the well-known Belady's algorithm [7]: when an item is requested but is not present in the cache, evict the cache item that will next be used farthest into the future. The proof of optimality of Belady's algorithm is through an exchange argument (see [19]).

At first glance, an analogous algorithm that uses the request patterns to decide whom to evict seems plausible for the rw-caching problem. To explore this thought further, we let $k = 2$ (i.e., cache of size 2) and assume $w = c_{\texttt{write}}(i) < c_{\texttt{read}}(i) = 1$ for every item i. For each item i, let $w(i) \geq 0$ be the number of write requests for i before the next immediate read request for it. For instance, for the sequence $\langle 1, \texttt{read}\rangle$, $\langle 2, \texttt{read}\rangle$, $\langle 3, \texttt{read}\rangle$, $\langle 1, \texttt{write}\rangle$, $\langle 1, \texttt{read}\rangle$, $\langle 2, \texttt{read}\rangle$, at the third request, $w(1) = 1$ and $w(i) = 0$ for every

$i \neq 1$. The following are two natural offline policies that are inspired by Belady's algorithm and can be thought of its two possible generalizations to the read-write case.

(P1) Evict any item i such that $w(i) \cdot \mathrm{c}_{\texttt{write}}(i) > \mathrm{c}_{\texttt{read}}(i)$; if no such item exists, evict the item that will next be read farthest into the future (as in Belady's algorithm).

(P2) Evict item i achieving $\arg\max\{w(i) \cdot \mathrm{c}_{\texttt{write}}(i) + \mathrm{c}_{\texttt{read}}(i)\}$.

However, it turns out neither of these policies dominates the other. Indeed, consider the sequence $\langle 1, \texttt{read}\rangle$, $\langle 2, \texttt{read}\rangle$, $\langle 3, \texttt{read}\rangle$, $\langle 1, \texttt{write}\rangle$, $\langle 1, \texttt{read}\rangle$, $\langle 2, \texttt{read}\rangle$. Focusing on the cost after the third request, it is easy to see that (P1) would evict item 2 and incur a total cost of $w + 1$, whereas (P2) would evict item 1 and incur a total cost of 1. Hence, for this sequence, (P1) is better than (P2). On the other hand, consider the sequence $\langle 1, \texttt{read}\rangle$, $\langle 2, \texttt{read}\rangle$, $\langle 3, \texttt{read}\rangle$, $\langle 1, \texttt{write}\rangle$, $\langle 1, \texttt{read}\rangle$, $\langle 3, \texttt{write}\rangle$, $\langle 3, \texttt{read}\rangle$, $\langle 2, \texttt{read}\rangle$. Again, focusing on the costs after the third request, the policy (P1) would still evict item 2 at the third and fifth requests and would incur a total cost of $2w + 1$, whereas (P2) would evict item 1 at the third request and item 3 at the fifth request and would incur a total cost of 2. Hence, for this sequence, (P2) is better than (P1). This gives some evidence that apparent generalizations of Belady's optimal algorithm to the read-write case may not work.

We now present an optimal offline exact algorithm for rw-caching, which is not based on a Belady-style policy. This algorithm is more holistic and takes a global view of reads and writes. It is based on min-cost flow. (Constructions based on min-cost flow have been used for the offline optimum for approximating join sizes in the sliding window stream setting [13, 34]; to the best of our knowledge, our use in a generalization of Belady's algorithm is new.)

Let $\mathcal{S} = \{\sigma_1, \ldots, \sigma_n\}$ be the request sequence, where each $\sigma_j \in \cup_i \{\langle i, \texttt{read}\rangle, \langle i, \texttt{write}\rangle\}$, and let C denote the current cache. Then any offline algorithm can be specified in the following manner: on each request σ_j, the algorithm first serves the request and then decides to evict a set E_j of items, hence updating the cache as $C \leftarrow C \setminus E_j$.

THEOREM 4. *The offline rw-caching problem is solvable in time $O(\mathrm{poly}(k, n))$, for a cache of size k and a request sequence of length n.*

PROOF. Let $\mathcal{S} = \{\sigma_1, \ldots, \sigma_n\}$ be the request sequence for the rw-caching. Let $\mathcal{T} = \{\tau_1, \ldots, \tau_m\}$ be the subsequence of $\mathcal{S}$ containing all the reads. For two indices i and $j > i$ in $\mathcal{T}$, let w_{ij} denote the number of write requests in $\mathcal{S}$ that arrive between $\{\tau_i, \ldots, \tau_j\}$ (or if $i = n$, then the write requests in S that arrive after τ_n).

The main idea is to construct a min-cost flow[1] instance corresponding to the sequence $\mathcal{S}$. Let K be an integer such that $K > n \cdot \max_\ell(\mathrm{c}_{\texttt{read}}(\ell) + \mathrm{c}_{\texttt{write}}(\ell))$. The set V of nodes in the min-cost flow instance comprises of

- a source s and a sink t;
- nodes $c_1, \ldots, c_k$, one for each cache location;
- nodes $\tau_1, \ldots, \tau_m, \tau'_1, \ldots, \tau'_m$, where the pair τ_i, τ'_i corresponds to the ith read request in $\mathcal{T}$;
- nodes $b_1, \ldots, b_m$.

[1]In the min-cost flow problem, we are given a directed graph with non-negative capacity and cost per unit flow on each edge, a source node, a sink node, and a required amount of flow from the source to the sink. The goal is to route this flow such that the edge capacity constraints are respected, and the total cost of the flow is minimized.

Thus, $|V| = 2 + k + 3m$. The set E contains the following edges, where each edge has unit capacity:

- $(s, c_1), \ldots, (s, c_k)$, $(c_1, t), \ldots, (c_k, t)$, each of cost 0;
- $(\tau'_1, t), \ldots, (\tau'_m, t)$, $(b_1, t), \ldots, (b_m, t)$, each of cost 0;
- for each $i \in [k]$ and for each $j \in [m]$, the edge (c_i, τ_j) if $\tau_j = \langle \ell, \texttt{read}\rangle$, of cost $\mathrm{c}_{\texttt{read}}(\ell)$;
- $(\tau_1, \tau'_1), \ldots, (\tau_m, \tau'_m)$, each of cost $-K$;
- $(\tau'_1, b_1), \ldots, (\tau'_m, b_m)$ each of cost 0;
- for each $i < j$, let $\tau_i = \langle \ell, \texttt{read}\rangle$ and $\tau_j = \langle \ell', \texttt{read}\rangle$
  - if $\ell = \ell'$, then the edge (τ'_i, τ_j) of cost $w_{ij} \cdot \mathrm{c}_{\texttt{write}}(\ell)$ and the edge (b_i, τ'_i) of cost $\mathrm{c}_{\texttt{read}}(\ell)$;
  - if $\ell \neq \ell'$, then the edge (τ'_i, τ_j) of cost $w_{ij} \cdot \mathrm{c}_{\texttt{write}}(\ell) + \mathrm{c}_{\texttt{read}}(\ell')$ and the edge (b_i, τ_j) of cost $\mathrm{c}_{\texttt{read}}(\ell')$.

Note that $|E| = O(m^2 + km)$. An example of the construction

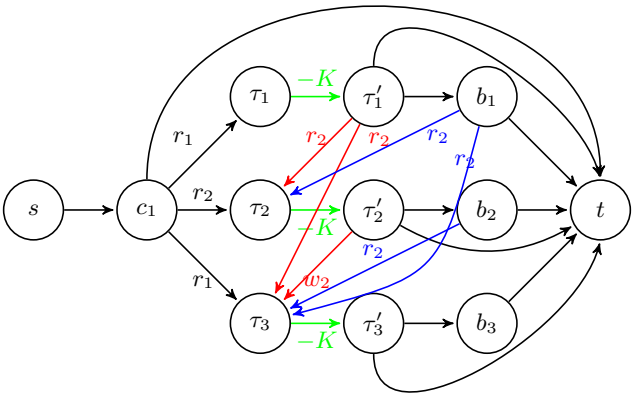


Figure 1: Construction for the request sequence $\langle 1, \texttt{read}\rangle, \langle 2, \texttt{read}\rangle, \langle 2, \texttt{write}\rangle, \langle 2, \texttt{read}\rangle$ and cache size $k = 1$. For $i = 1, 2$, let $r_i = \texttt{read}(i)$ and $w_i = \texttt{write}(i)$. From construction, $\tau_1 = \langle 1, \texttt{read}\rangle$, $\tau_2 = \tau_3 = \langle 2, \texttt{read}\rangle$, and let $K > 4\max\{r_1 + w_1, r_2 + w_2\}$. The capacities are all unit and the non-zero costs are shown on the edges; edges with no labels have zero cost.

is shown in Figure 1. We first provide an informal intuition behind the construction. A flow from τ_i to τ'_i means that the ith request is served. If the flow goes to b_i, then it means the item is evicted after serving. If the flow goes from τ'_i to τ_j, $j > i$, it means that the item read in the ith request is replaced (or reused) for the jth request. Finally, a flow from b_i to τ_j means that the cache location was empty for the duration between the ith and the jth requests. Thus, each flow path corresponds to the decisions made by one cache location.

We now proceed formally. Note that the maximum flow in this construction has value k. Furthermore, we can assume the flow is integral as all capacities are integers. Since all capacities are unit, the flow consists of k edge disjoint paths, each carrying unit flow. Each disjoint path is through one of the c_i's and corresponds to the decision taken on each request by the ith cache location.

Let $\mathcal{F}$ be the set of maximum flows such that each of the nodes τ_i is present in one the k flow paths. Let $\mathcal{C}$ be the set of caching

policies where eviction is done only on receiving read requests. (Note that we perform a set of actions per request. On receiving a read request for an item not in cache, one of these actions must be a read for this item.)

We first claim that there is a one-one relation between flows in $\mathcal{F}$ and caching policies in $\mathcal{C}$. Indeed, for every flow of value k, the unique flow through node c_i can be interpreted as the caching decisions taken by the ith cache location. Suppose this flow goes through the τ-nodes $\tau_{i_1}, \ldots, \tau_{i_t}$, in this order (there are other types of nodes as well in this path). Then, the item in the request τ_{i_1} is the first item read into this cache location. Consider a generic τ_i in this path, and let $\tau_i = \langle \ell, \texttt{read} \rangle$. If the flow goes from τ_i through b_i (or to t), we evict item ℓ right after the request τ_i. If the flow goes from τ_i next to $\tau_j = \langle \ell', \texttt{read} \rangle$ for $j > i$, there are two cases. If $\ell = \ell'$, then we maintain the item ℓ in the cache location for all the requests between τ_i and τ_j. If $\ell' \neq \ell$, then ℓ is evicted when item ℓ' is loaded on request τ_j.

The cost of the flow, other than the $-K$ edges, captures the cost of the caching policy. The read costs for each τ_i are present only once in the graph, and hence charged at most once by the flow. Each write is present multiple times—a write request for item ℓ that arrives after τ_i is present on all outgoing edges $(\tau_i', \tau_j), \forall j > i$. However, since the incoming flow to τ_i' is at most one, at most one of these edges carries a unit flow.

Conversely, given a caching policy, a flow can be constructed using the same correspondence.

Finally, to ensure optimality, note that without loss of generality, the optimal offline policy might as well belong to $\mathcal{C}$, since any eviction of an item ℓ done on a write request could be done earlier, when processing the last read request for ℓ. Also, for the chosen value of K, the min-cost flow will go through all the edges (τ_i, τ_i') and hence will go through every τ node, and hence will belong to $\mathcal{F}$. The policy corresponding to this flow will thus be the optimal.

Thus, the optimal policy can be computed by solving the min-cost flow on the graph, which can be done in time $O(\text{poly}(k, n))$. □

It can be shown (omitted in this version) that in the absence of write requests, solving the min-cost flow on the construction is equivalent to Belady's algorithm for the classical case.

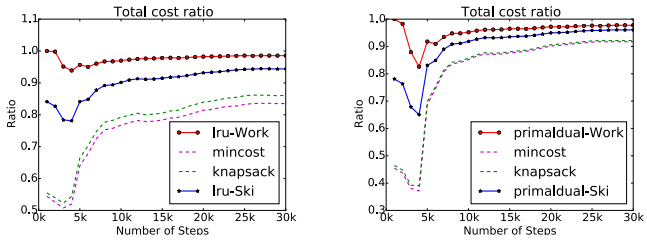


Figure 2: Comparison of total cost ratio (including mincost) for FINANCIAL-SMALL in both the unweighted and weighted case (lower is better).

6. EXPERIMENTS

6.1 Setup

Data. We conduct experiments using two publicly available datasets as well as a synthetic dataset. The FINANCIAL dataset, obtained from the Trace Repository at `traces.cs.umass.edu/`, comes from storage accesses of OLTP applications in financial institutions. The ALEGRA dataset, obtained from `www.cs.sandia.gov/Scalable_IO/SNL_Trace_Data/`, contains I/O kernel trace data. Both FINANCIAL and ALEGRA mention the item sizes (in bytes) along with the read and write requests. We use these sizes as the costs in the weighted version. Since the offline optimal min-cost algorithm could not be made to run on the large datasets, we took a prefix of the first 30k requests from the FINANCIAL dataset to compare the performance of the online algorithms with the offline optimal.

The SYNTHETIC dataset was generated in the following manner. There are three parameters: a power law parameter α, the insertion probability β, and a parameter $r_{\max}$ that specifies the maximum fraction of reads that are allowed. For each item, we first select a fraction in $[0, r_{\max}]$ that decides what fraction of the requests for this item are read requests. At every time step i, we either decide to create a new item with probability β, or choose the jth past item with probability p_j according to the power law distribution with parameter α. These three parameters allow us to trade off locality, the appearance of new items, and the read/write ratio. Intuitively α controls the locality in time, β is the "distance" from stationarity, and $r_{\max}$ controls the fraction of read and writes in the data.

Dataset	# reads	# writes
FINANCIAL	1235k	4099k
FINANCIAL-SMALL	15k	14k
ALEGRA	141k	243k
SYNTHETIC	$\sim$ 50k	$\sim$ 50k

Algorithms and baselines. In the unweighted case, the caching algorithm is always LRU (since it is the most basic algorithm and since it is well known that LRU performs much better in practice than most theoretical caching algorithms). In the weighted case however, obtaining a bounded approximation caching algorithm is a harder problem. We use a deterministic primal-dual algorithm (PrimalDual) based on [5]. Note that this is not the algorithm that gives the best approximation for the weighted classical caching problem. However, we chose this because it is simple, deterministic, and is a generalization of LRU to the weighted case.

We consider the following variants of algorithms for MTS.

(i) AlwaysIn: the MTS is a single-state one that always return `in` to any query. This is our main baseline, since here the algorithm is only running LRU for maintaining a read cache.

(ii) Ski: When the number of states of the MTS is two, it is also known the rent-or-buy problem. The well-known deterministic ski-rental algorithm [3][2] gives a 2-approximation to the rent-or-buy MTS problem [9].

(iii) Work: the MTS is based on a work-function, which gives a 3-approximation to the 2-state MTS problem. In this, the MTS at step i goes to a state X that minimizes the sum of the optimal at $i-1$ steps, and the distance from this $(i-1)$st optimal to X.

In the figures we use the notation $\mathcal{A}$-$\mathcal{B}$ to denote that the caching algorithm is $\mathcal{A}$ and the MTS algorithm is $\mathcal{B}$.

In addition we also implement the following two offline algorithms. The Mincost algorithm, which computes the offline min-cost based solution (Theorem 4), is implemented using a integer linear programming package or-tools (`github.com/google/or-tools`). Since Mincost is computationally infeasible for large datasets, we also use a heuristic approximation based on a Knapsack problem. The knapsack instance is created by defining the weight of any element as the total read cost minus the write cost over the entire stream. The caching algorithm simply stores the top items with respect to this cost.

[2]See `en.wikipedia.org/wiki/Ski_rental_problem` for a simple description.

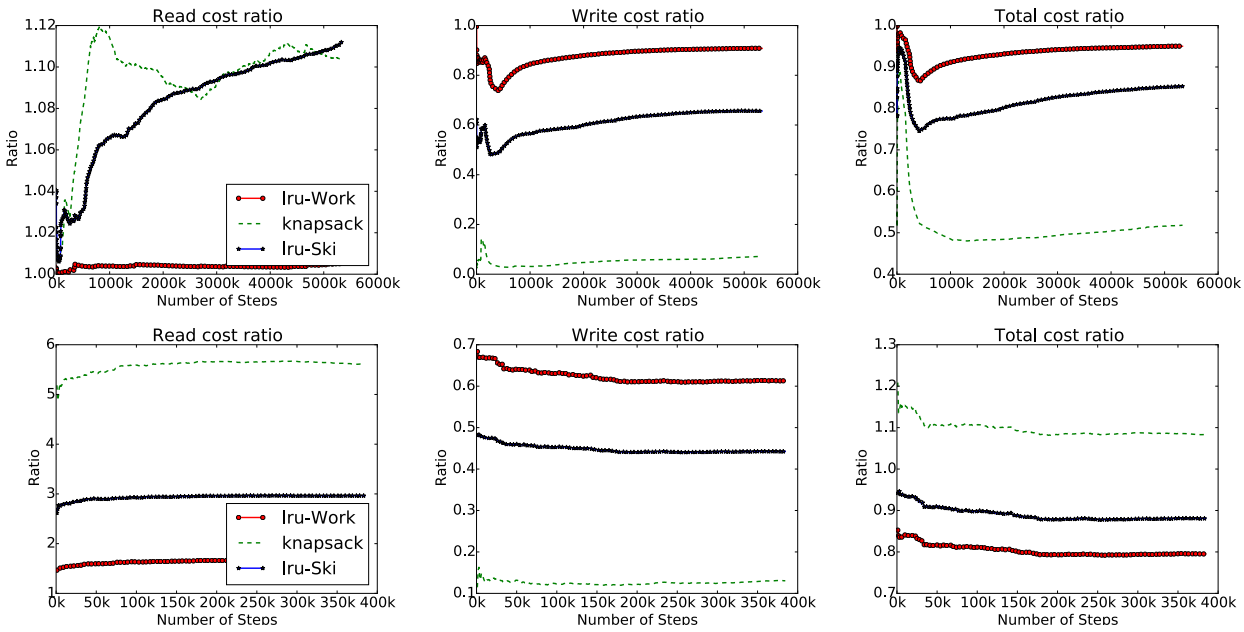


Figure 3: Read-, write-, and total-cost ratios for the FINANCIAL and ALEGRA datasets (lower is better).

Cache size. In all the real data experiments, the size of the cache was fixed at 10% of the number of distinct items. For SYNTHETIC, the size of the cache was fixed at 1000, while the number of distinct items in the generated data was always more than 15k. In all the plots, we start the plot from the first request, since the warm-up phase is anyway insignificant compared to the entire sequence length.

Measures. We study the following measures: the total read cost, total write cost, as well the total cost (read + write). We measure each of these costs per 1000 requests. These costs are then normalized by the cost incurred by the baseline algorithm AlwaysIn at the same request count. We refer to these normalized statistics as the *read-cost ratio*, *write-cost ratio*, and *total-cost ratio*. Similarly, we will also look at the *number of cached-items ratio* which is defined similarly by normalizing the number of cached items at every timestep by the corresponding quantity for LRU-AlwaysIn. We study these costs in both the unweighted and weighted cases. In the unweighted case, the read cost of each item is the same and so is the write cost. For the unweighted case we further study the cases where the write cost is the same as read cost, as well as twice the read cost; we call this multiplier $c_{\texttt{write}}$. In the weighted case the size of the items as obtained from the data is used as the cost of an individual read.

6.2 Results

Comparison to offline. In Figure 2 we compare the total-cost ratio of the various online algorithms, as well as the two offline variants of Knapsack and Mincost, both in the unweighted and weighted model (with $c_{\texttt{write}} = 1$). For this small dataset, even the performance of the offline optimal Mincost is at least 80% of AlwaysIn. As the two plots show, the Knapsack heuristic is a good approximation to the Mincost solution, at least in this dataset. In the subsequent plots, since we cannot calculate Mincost on the large datasets, we only show Knapsack as the offline comparison.

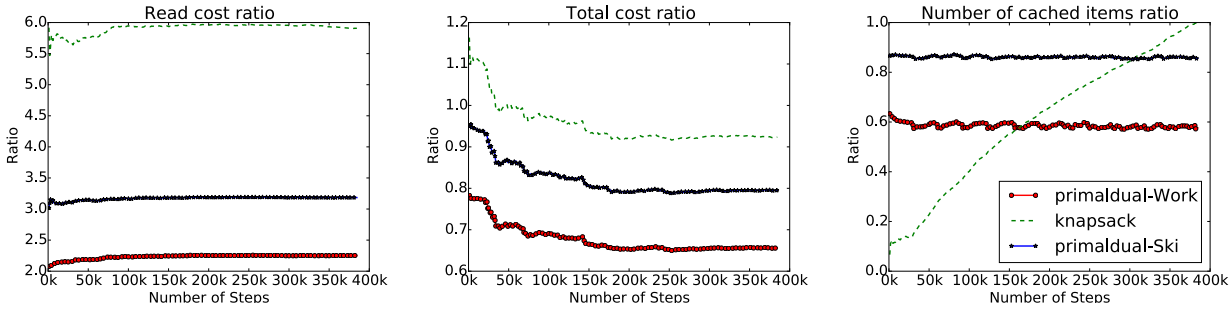
Performance of various online algorithms. Next we show compare the various online algorithms. Figure 3 shows the read-cost ratio, the write-cost ratio as well as the total-cost ratio of all the online algorithms, and Knapsack, on the two large datasets FINANCIAL and ALEGRA in the unweighted model. Note that the read-cost ratio of AlwaysIn is obviously much better than the others, since it essentially focuses on only the reads. However, in terms of the write cost, and thus, the total-cost, both Ski and Work improve upon the baseline significantly. However, it does not seem possible to identify one of the MTS as a strict winner in terms of the total-cost ratio, since their relative orders are different in the two datasets. However, both of them give at least a 10% (20% for best) improvement over AlwaysIn. Note that this is in the $c_{\texttt{write}} = 1$ case, as the $c_{\texttt{write}}$ increases, the gap between the MTS algorithms and AlwaysIn will surely increase.

Also note that it is fairly obvious that Knapsack is not a good stand-in for the optimal in the ALEGRA dataset.

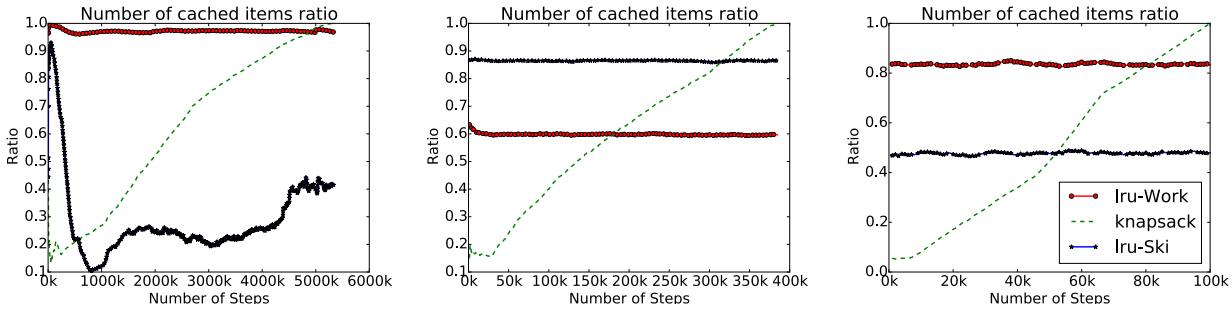
In Figure 4(a) we also investigate the performances of the weighted model, where each item has its own weight. Again, both the MTS algorithms have a cost that is at most 80% of the baseline (70% for Work). Simultaneously, we see that both the MTS algorithms actually end up storing much less items than given the cache size.

Number of cached items. Figure 4(b) shows the number of cached items by each algorithm as the requests arrive in FINANCIAL, ALEGRA, and SYNTHETIC datasets. In each of these datasets, it is instructive to note that the proposed MTS algorithms actually caches less items than the maximum number allowed. This is in fact a big plus, combined with the previous observation that the total cost of these algorithms is also less than the AlwaysIn baseline. The 'best' MTS caches only 40-60% of the number cached by AlwaysIn.

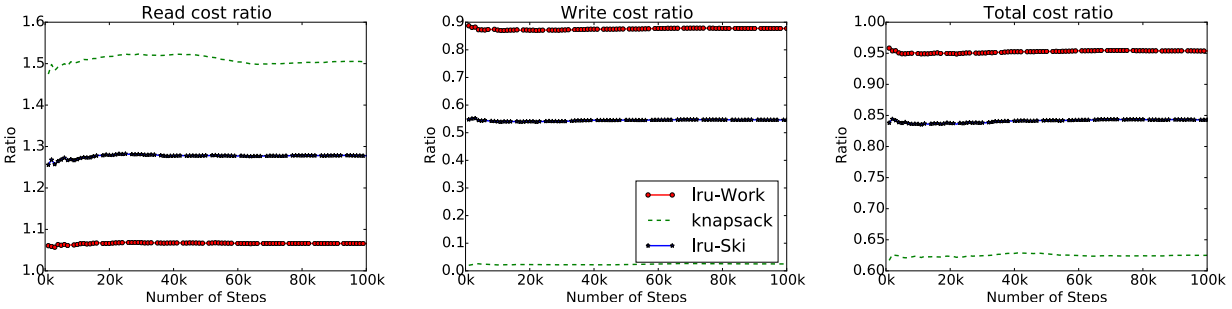
Statefulness vs stateless. We next investigate the comparative performance of the stateless (i.e., MTS maintained only for items in cache) and the stateful versions. In all datasets, the performances are extremely close, there being less than 0.04% difference in the maximum total cost for the largest dataset FINANCIAL (total costs 351690 vs 351802 for the 5-millionth step). Given this observa-



(a) Read-cost ratio, total-cost ratio, and number of cached item comparisons for the weighted model for ALEGRA.



(b) Ratio of the number of cached items in the unweighted case for FINANCIAL, ALEGRA, and SYNTHETIC.



(c) Read-, write-, and total-cost ratio in the unweighted case for SYNTHETIC for dataset with at most 0.5 fraction reads.

Figure 4: Performance of algorithms (lower is better).

tion, we advocate the stateless version in practice even though it has a slightly worse competitive ratio.

SYNTHETIC **dataset.** Using the SYNTHETIC dataset, we study the effect of varying the parameters α, $r_{\max}$, and β. For lack of space we only show plots with $\alpha = 2.0$ and $\beta = 0.3$. We choose $r_{\max} \in \{0.1, 0.5\}$ to investigate the effect of varying proportions of read and write. Figures 4(c) show the performance when the fraction of reads is at most 0.5 (the performance when the fraction of reads is at most 0.1 is qualitatively similar and omitted). As expected, as the fraction of writes increases in the input, the gain by the proposed algorithms also increases, from 0.85 for at most 50% reads to 0.6 when the data has at most 10% reads.

7. CONCLUSIONS

In this paper we introduced and studied the problem of read-write or dual cost caching. Our work is motivated by distributed cache settings where items can be updated often, and there is a cost to keeping certain items in the cache owing to the overhead of invalidating them often. Our formulation is simple and builds upon the formulation used in traditional caching. Our online algorithm takes this one step further by building upon online algorithms for traditional caching, and using them in conjunction with algorithms for MTS. We believe this composition and the analysis of its performance is novel and could have other applications. Also, given the simplicity of our algorithm and its use of existing algorithms as black-boxes, it should be easy to adopt it in practice. To fully understand the performance of this online algorithm, we also develop an optimal offline algorithm, which is a generalization of Belady's well-known algorithm for the read-only case.

While we have solved the basic case of read-write caching, there are several other interesting research directions to be explored. Understanding the performance of the online algorithm for stylized workloads would be useful from a theoretical angle. An offline optimal that is not based on min-cost flow would be valuable in practice. It will also be interesting to introduce a notion of time into our framework so that existing policies for cache expirations can be folded into the framework in a principled manner.

8. REFERENCES

[1] A. Adamaszek, A. Czumaj, M. Englert, and H. Räcke. An $O(\log k)$-competitive algorithm for generalized caching. In *SODA*, 2012.

[2] S. Alici, I. S. Altingovde, R. Ozcan, B. B. Cambazoglu, and O. Ulusoy. Timestamp-based result cache invalidation for Web search engines. In *SIGIR*, pages 973–982, 2011.

[3] Y. Azar, U. Feige, and S. Nath. On the work function algorithm for two state task systems. Technical report, MSR-TR-2007-20, Microsoft, 2007.

[4] X. Bai and F. P. Junqueira. Online result cache invalidation for real-time Web search. In *SIGIR*, pages 641–650, 2012.

[5] N. Bansal, N. Buchbinder, and J. S. Naor. A primal-dual randomized algorithm for weighted paging. *JACM*, 59(4):19, 2012.

[6] W. Bein, L. Larmore, and J. Noga. Uniform metrical task systems with a limited number of states. *IPL*, 104(4):123–128, 2007.

[7] L. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[8] R. Blanco, E. Bortnikov, F. Junqueira, R. Lempel, L. Telloli, and H. Zaragoza. Caching search engine results over incremental indices. In *WWW*, pages 1065–1066, 2010.

[9] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[10] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, pages 826–835. IEEE, 2007.

[11] B. B. Cambazoglu, F. P. Junqueira, V. Plachouras, S. Banachowski, B. Cui, and S. Lim. A refreshing perspective of search engine caching. In *WWW*, pages 181–190, 2010.

[12] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[13] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD*, pages 40–51, 2003.

[14] A. Fiat and M. Mendel. Better algorithms for unfair metrical task systems and applications. *SICOMP*, 32(6):1403–1422, 2003.

[15] G. Francès, X. Bai, B. Cambazoglu, and R. Baeza-Yates. Improving the efficiency of multi-site Web search engines. In *WSDM*, pages 3–12, 2014.

[16] K. Hose, D. Klan, and K.-U. Sattler. Online tuning of aggregation tables for olap. In *ICDE*, pages 1679–1686, 2009.

[17] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *SOSP*, pages 167–181, 2013.

[18] A. Karlin, M. Manasse, L. McGeoch, and S. Owicki. Competitive randomized algorithms for nonuniform problems. *Algorithmica*, 11(6):542–571, 1994.

[19] J. Kleinberg and E. Tardos. *Algorithm Design*. Pearson, 2005.

[20] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. *PVLDB*, 1(1):514–525, 2008.

[21] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *WWW*, pages 19–28, 2003.

[22] R. Lempel and F. Silvestri. Web search result caching and prefetching. In *Encyclopedia of Database Systems*, pages 3501–3506. Springer, 2009.

[23] X. Long and T. Suel. Three-level caching for efficient query processing in large Web search engines. In *WWW*, pages 257–266, 2005.

[24] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: measuring and understanding consistency at Facebook. In *SOSP*, pages 295–310, 2015.

[25] T. Malik, X. Wang, R. Burns, D. Dash, and A. Ailamaki. Automated physical design in database caches. In *ICDEW*, pages 27–34, 2008.

[26] S. Nath and A. Kansal. Flashdb: Dynamic self-tuning database for nand flash. In *IPSN*, pages 410–419, 2007.

[27] S. Pandey, A. Z. Broder, F. Chierichetti, V. Josifovski, R. Kumar, and S. Vassilvitskii. Nearest-neighbor caching for content-match applications. In *WWW*, pages 441–450, 2009.

[28] S. Paul and Z. Fei. Distributed caching with centralized control. *Computer Communications*, 24(2):256–268, 2001.

[29] M. Rabinovich and O. Spatscheck. *Web Caching and Replication*. Pearson, 2001.

[30] C. Ravishankar and J. Goodman. Cache implementation for multiple microprocessors. *Computer Communications*, pages 346–350, 1983.

[31] M. Satyanarayanan. A survey of distributed file systems. *Annual Review of Computer Science*, 4(1):73–104, 1990.

[32] K. Schnaitter and N. Polyzotis. A benchmark for online index selection. In *ICDE*, pages 1701–1708, 2009.

[33] K. Schnaitter and N. Polyzotis. Semi-automatic index tuning: Keeping dbas in the loop. *PVLDB*, 5(5):478–489, 2012.

[34] J. Xie, J. Yang, and Y. Chen. On joining and caching stochastic streams. In *SIGMOD*, pages 359–370, 2005.

[35] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *WWW*, pages 387–396, 2008.