Linear Elastic Caching via Ski Rental

Ravi Kumar
Google
ravi.k53@gmail.com

Todd Lipcon
Google
tlipcon@google.com

Manish Purohit
Google
mpurohit@google.com

Tamas Sarlos
Google
stamas@google.com

ABSTRACT

In this work we study the Linear Elastic Caching problem, where the goal is to minimize the total cost of a cache inclusive of not just its misses, but also its memory footprint integrated over time. We demonstrate a theoretical connection to the classic ski rental problem and propose a practical algorithm that combines online caching algorithms with ski rental policies. We also introduce a lightweight machine learning-based algorithm for ski rental that is optimized for production workloads and is easy to integrate within existing database systems. Evaluations on both production workloads in Google Spanner and publicly available traces show that the proposed elastic caching approach can significantly reduce the total cache cost compared to traditional fixed-size cache policies.

1 INTRODUCTION

In-memory caching is a cornerstone technique for achieving high performance in database management systems and cloud services. By storing frequently accessed data in main memory, caching reduces costly disk I/O operations and improves latency. However, memory is expensive—for example, Amazon ElastiCache Serverless charges $3 per day for just 1GiB[1]. Given the high cost, provisioning these caches presents a fundamental challenge: substantial human effort might be needed to determine an optimal cache capacity, and statically allocating memory to meet peak demand can lead to significant resource underutilization for bursty workloads.

Traditional caching systems typically assume fixed memory resources and focus on eviction policies to minimize misses. However, cloud environments offer new opportunities for cost optimization. For example, ElastiCache charges only for the *average* size of the maintained cache. Thus, one could minimize this cost by dynamically increasing or decreasing the cache capacity over time in response to the workload.

In this paper, we formulate the *linear elastic caching* problem of designing a policy that minimizes the total cost incurred by the cache: both the direct storage cost of maintaining the cache in RAM, and the direct and indirect costs associated with cache misses.

[1]https://aws.amazon.com/elasticache/pricing/ (as of November, 2024)

 15th Annual Conference on Innovative Data Systems Research (CIDR '25). January 19-22, Amsterdam, The Netherlands

Our work draws an interesting connection between linear elastic caching and the classic ski rental problem. We propose a practical algorithm that integrates online caching algorithms with ski rental policies, determining a time-to-live (TTL) for each cached page based on its access patterns and associated costs. Theoretical analysis shows that separately optimizing the cache eviction policy and the ski rental policy suffices to minimize the total cost.

This work was directly motivated by a desire to reduce cache costs in Spanner [9], a large globally distributed database system at Google. Since data access patterns encountered in production are far from adversarial, we opt for a lightweight machine learning approach for designing an appropriate ski rental policy. We show that incorporating such a learned ski rental algorithm with Spanner's existing cache eviction policy helps to minimize the total cost of cache ownership by over 5%. We also evaluate our approach through extensive experiments on publicly available traces. Results show that linear elastic caching significantly reduces the total cost incurred by the cache compared to fixed-size cache policies.

2 PROBLEM FORMULATION

We refer to entries in the cache as "pages" for simplicity. Unlike traditional caching problems where the cache size is fixed and the goal is to design an eviction policy to minimize cache misses, here we consider the setting where the cache size is variable and the goal is to strike a balance between the cost of cache misses and the cost of maintaining the cache (in RAM bytes × seconds) over time.

For a page p, let $r(p)$ denote the *per-unit time cost* of holding the page in cache, and $B(p)$ denote the cost of incurring a *cache miss* for the page[2]. Typically, $r(p)$ is proportional to the size of the page and scales with RAM prices, while $B(p)$ captures the cost of the latency and/or I/O incurred due to the cache miss; however, these quantities can be arbitrary functions of the page contents or metadata. Let $\mathrm{sz}(p)$ denote the size (in bytes) of page p.

For a fixed universe U of pages and a cache with maximum capacity k, an algorithm receives a sequence σ of page requests online and must decide how long to hold each page in cache with the goal of minimizing the total cost, i.e., the sum of the total cost of all cache misses and the total RAM cost. Formally, if A_t is the set of pages in cache at time t by some algorithm $\mathcal{A}$ for request sequence σ, then the total cache maintenance cost is defined as $\mathrm{MemCost}(\mathcal{A}, \sigma) = \sum_t \sum_{p \in A_t} r(p)$ and the total cost due to cache evictions is defined as $\mathrm{EvictCost}(\mathcal{A}, \sigma) = \sum_t \sum_{p \in A_{t-1} \setminus A_t} B(p)$[3].

[2]We use symbols r and B as they correspond to the cost for *renting* and *buying* in Ski Rental respectively.

[3]As is standard in caching literature, we charge for cache evictions rather than cache misses. The two quantities only differ by a constant since any page p that is brought

The *total cost* of ownership incurred by the algorithm is defined as:

$$\text{TCO}(\mathcal{A}, \sigma) = \text{MemCost}(\mathcal{A}, \sigma) + \text{EvictCost}(\mathcal{A}, \sigma).$$

The goal of the *linear elastic caching problem* is to design an online algorithm $\mathcal{A}$ that minimizes $\text{TCO}(\mathcal{A}, \sigma)$ subject to the constraint that the total size of pages in the cache at any time t is at most k, i.e., $\sum_{p \in A_t} \text{sz}(p) \leq k$.

As is standard in the analysis of online algorithms, we adopt the lens of competitive analysis, which provides a bound on the worst-case performance of the online algorithm. Formally, for any instance $\mathcal{I}$ of a minimization problem, if $\text{cost}(\mathcal{A}, \mathcal{I})$ is the cost incurred by some online algorithm $\mathcal{A}$ and $\text{OPT}(\mathcal{I})$ is the cost incurred by an optimal offline algorithm, then the competitive ratio of $\mathcal{A}$ is defined to be:

$$\text{competitive-ratio}(\mathcal{A}) = \max_{\mathcal{I}} \frac{\text{cost}(\mathcal{A}, \mathcal{I})}{\text{OPT}(\mathcal{I})}.$$

If the competitive ratio of $\mathcal{A}$ equals c, we often use the shorthand that $\mathcal{A}$ is c-competitive.

Specifically, for the linear elastic caching problem, for any request sequence σ, let $\text{OPT}(\sigma)$ denote the total cost incurred by the optimal offline algorithm that is aware of the entire sequence ahead of time. Then the competitive ratio of an algorithm $\mathcal{A}$ is defined as:

$$\text{competitive-ratio}(\mathcal{A}) = \max_{\sigma} \frac{\text{TCO}(\mathcal{A}, \sigma)}{\text{OPT}(\sigma)}.$$

3 RELATED WORK

The problem of dynamically changing the cache size in response to the workload has been well-studied in the algorithms research community. Gupta et al. [14] propose the general *elastic caching problem* where the cache maintenance cost can be a general function of the set of pages stored in the cache. In their setting, $f : 2^U \to \mathbb{R}^+$ is an arbitrary non-negative set function and the cache maintenance cost incurred by an algorithm is $\sum_t f(A_t)$ where A_t is the set of pages held in cache at time t. They design an $O(\log n)$-competitive randomized online algorithm when the maintenance cost depends only on the cache size (rather than the identities of the pages in cache) and an n-competitive deterministic algorithm for arbitrary monotone functions where $n = |U|$. We note that the linear elastic caching problem studied in our paper is a special case where $f(A_t) = \sum_{p \in A_t} r(p)$ if the total size of pages in A_t is at most k and $f(A_t) = \infty$ otherwise.

Dynamically adjusting the cache size has also been studied in the systems and database communities [5, 6, 15, 19, 25]. The broad goal there is to explore elastic resource management, when cache and cache maintenance costs become a factor; they observe that dynamic cache instantiation can provide substantial cost reductions. Perhaps the most directly relevant paper is by Carra et al. [6], where they consider horizontal scaling of the in-memory cache by dynamically adding/removing cache instances. They assume that page requests follow a Poisson arrival process and aim to dynamically optimize for the best single TTL that can be applied for all pages. The algorithm maintains a virtual TTL cache [13] and at the end of each epoch sets the number of cache instances based on the size of the virtual cache.

into the cache must also get evicted, except for the pages in the cache at the end of the input instance.

Optimizing storage costs by exploiting tiered storage options in the cloud has also been an active area of study [12, 20–23, 26]. When there are multiple storage tiers with differing storage and access costs, these papers observe that (generalizations of) the ski rental problem can be applied in this setting. Puttaswamy et al. [26] propose a frugal cloud file system by automatically transferring data between hot and cold storage tiers and rederive the optimal deterministic and randomized ski rental algorithms. However, these papers do not provide theoretical guarantees that when the hot storage has limited size, the ski rental policies do not adversely interact with the cache eviction policy. In contrast, we provide a theoretical justification for incorporating ski rental algorithms side-by-side with existing cache eviction policies and show that optimizing for both separately does indeed suffice to minimize the total cost of ownership.

4 CONNECTIONS WITH SKI RENTAL

Our problem description was formulated suggestively to indicate the similarity with the classical ski rental [2] problem. Recall that in the *ski rental problem*, a skier wishes to ski for some unknown number d of days and they can either *rent* a pair of skis at a cost of r per day or *buy* them at a higher cost of B and ski for free from then on. Clearly, if $d \geq \frac{B}{r}$, an optimal solution is to buy the skis on the first day and incur a total cost of B; on the other hand, if $d < \frac{B}{r}$, the optimal solution is to only rent skis and pay a total cost of $d \cdot r < B$. However, d is unknown to an online algorithm and each day when the skier wants to ski, the algorithm needs to decide whether to rent skis for the day or buy (and ski without cost for any future days).

The ski rental problem is one of the prototypical examples of decision making under uncertainty and has been well studied in the classical competitive analysis framework. The goal of competitive analysis is to study the worst-case performance of online algorithms. In particular, consider the *breakeven algorithm* that rents skis for the first $\lfloor \frac{B}{r} \rfloor$ days and then buys on the next day (if any). It can be readily observed via a simple case analysis that the total cost incurred by the breakeven algorithm is never more than twice the cost incurred by the optimal offline solution. Indeed, if $d \leq \lfloor \frac{B}{r} \rfloor$, then the breakeven algorithm simply rents the skis for all d days and incurs the same cost as the optimal solution; on the other hand, if $d > \lfloor \frac{B}{r} \rfloor$, then the breakeven algorithm incurs a total cost of $\lfloor \frac{B}{r} \rfloor \cdot r + B \leq 2B$ while the optimal solution buys on the first day and pays B. The breakeven algorithm is thus 2-competitive and this is best possible for deterministic algorithms. However, randomization helps and Karlin et al. [17] design a randomized algorithm that obtains a competitive ratio of $e/(e-1) \approx 1.58$. It is well known that any deterministic algorithm for ski rental is characterized by how long the algorithm rents skis before finally buying them, while any randomized algorithm is characterized by a distribution over such buy times.

Now, focusing back on the linear elastic caching problem, consider a single request to page p at some time t_1. Suppose the next request to p is at some time $t_2 > t_1$. Then considering this page in isolation, there are two alternatives available: (i) either, we hold the page in cache and incur a cache maintenance cost of $r(p)$ per unit time, or we (ii) evict the page p and incur an eviction cost of $B(p)$.

Since the total length of this time interval $(t_2 - t_1)$ is unknown when the page arrives (at time t_1), the problem faced by the caching algorithm is exactly the ski rental problem with holding the page in cache corresponding to renting and eviction corresponding to buying. We emphasize that this is simply an informal connection between the two problems and not a formal reduction—in linear elastic caching, we cannot treat each page in isolation as the total size of pages in cache at any time cannot exceed the maximum cache capacity k.

5 MAIN RESULTS

Since each page request in isolation leads to an instance of the ski rental problem, a natural strategy for linear elastic caching is to invoke an algorithm for the ski rental problem for each page request. Suppose for a particular page p that is requested at time t, and a ski rental algorithm responds with a buy time of b. Then one can set a TTL (*time-to-live*) of $t + b$ when admitting or accessing page p in the cache. Thus *renting* for b time units followed by buying exactly corresponds to evicting the page from cache when its TTL expires.

However, since such a strategy treats each page request independently, it may violate the maximum cache capacity k. A linear elastic caching algorithm needs to simultaneously solve these per-request ski rental instances while also maintaining that the cache size is always at most the capacity k.

First, we show that one can combine an arbitrary online algorithm for regular caching (with a fixed size cache and a goal of minimizing the total miss cost) with an arbitrary ski rental algorithm, in a black box manner. The result relies on a reduction of linear elastic caching to the *read-write caching problem* [10] that we define next. In an instance of read-write caching, each page request is either a *read* request or a *write* request. On a read request to a page p, if p is not in the cache then the algorithm incurs a *miss* cost. On the other hand, on a write request to a page p, the algorithm incurs a *write* cost if p is in the cache. The goal is to design an algorithm that minimizes the sum of miss costs and write costs.

Theorem 5.1. *Given an α-competitive online algorithm for caching and a 2-competitive deterministic algorithm for ski rental, there is an $(\alpha + 5)$-competitive algorithm for linear elastic caching.*

Proof. We can reduce an instance of linear elastic caching to an instance of read-write caching as follows. Let σ be a sequence of page requests for the linear elastic caching instance. We create a new sequence σ' of read-write page requests as follows: for each page p that arrives at time t in σ, add a read request for page p at time t in σ'. In addition, for each discrete time step, add a write request to every page in the universe. Each page p has a write cost of $r(p)$ and a read cost of $B(p)$. It can be observed that for any caching algorithm $\mathcal{A}$, the read-write cost on sequence σ' equals the $\mathrm{TCO}(\mathcal{A}, \sigma)$. The proof now follows from [10, Theorem 3]. □

While Theorem 5.1 yields almost best-possible theoretical guarantees for linear elastic caching, the proposed algorithm is not ideal to deploy in a production environment since it necessitates maintaining additional bookkeeping. The cache eviction algorithm $\mathcal{A}$ works under the assumption that the cache always has size k and maintains a *ghost cache* with a superset of pages that are actually held in cache. In our next result, we show that when using practical cache eviction policies such as *Least Recently Used* (LRU), simply adding pages to the cache with a TTL that is determined by a ski rental algorithm suffices. We give a formal description in Algorithm 1. Informally, whenever a page request arrives, the ski rental algorithm determines a TTL for the page. The page gets evicted from the cache at its TTL unless it is requested again by that time (the TTL will then be recomputed). If at any point, the requested page is not in the cache and the cache is full, then the cache eviction algorithm determines which page(s) to evict.

Input: Eviction Policy $\mathcal{A}$, Ski Rental Algorithm $\mathcal{B}$
$C \leftarrow \emptyset$;
for *time* $t = 1, 2, \ldots,$ **do**
  `/* Evict any pages whose TTLs have expired */`
  for *page* $p \in C$ **do**
    if $\mathrm{ttl}(p) < t$ **then**
      Evict p from C;
    end
  end
  `/* Process page requested at time t if any */`
  if *page* p_t *is requested at time* t **then**
    Send p_t to $\mathcal{A}$;
    $\mathrm{buytime} \leftarrow \mathcal{B}(p_t)$;
    $\mathrm{ttl}(p_t) \leftarrow t + \mathrm{buytime}$;
  end
end

Algorithm 1: Linear Elastic Caching algorithm $(\mathcal{A} + \mathcal{B})$.

We first introduce some notation to simplify the following discussion. Any page request sequence σ gives rise to many independent instances of the ski rental problem (one per page request). Let $\mathrm{SkiCost}(\mathcal{B}, p, t)$ be the cost incurred by the ski rental algorithm $\mathcal{B}$ on the instance corresponding to a request to page p at time t and let $\mathrm{SkiCost}(\mathcal{B}, \sigma) = \sum_t \mathrm{SkiCost}(\mathcal{B}, p_t, t)$ be the total cost incurred by Algorithm $\mathcal{B}$ on all the induced instances.

Theorem 5.2. *When the cache eviction algorithm $\mathcal{A}$ is LRU, then for any request sequence σ, Algorithm 1 satisfies*

$$\mathrm{TCO}(\mathcal{A} + \mathcal{B}, \sigma) \leq \mathrm{EvictCost}(\mathcal{A}, \sigma) + \mathrm{SkiCost}(\mathcal{B}, \sigma).$$

Proof. Let $\mathrm{TCO}(\mathcal{A} + \mathcal{B}, p_t, t)$ be the total cost incurred by the algorithm on page p_t requested at time t. So by definition we have $\mathrm{TCO}(\mathcal{A} + \mathcal{B}, \sigma) = \sum_t \mathrm{TCO}(\mathcal{A} + \mathcal{B}, p_t, t)$. Let $\mathrm{buytime}(p_t, t) = \mathcal{B}(p_t, t)$ be the time duration for which algorithm $\mathcal{B}$ recommends holding p_t in cache. Let time-in-cache $\mathrm{tic}(p_t, t)$ denote the number of time units that page p_t was actually held in the cache and let d_t be the number of time units until the next request of page p_t in σ. Note that we have $\mathrm{tic}(p_t, t) \leq \mathrm{buytime}(p_t, t)$ since p_t may be evicted earlier by Algorithm $\mathcal{A}$, and also $\mathrm{tic}(p_t, t) \leq d_t$. Finally let $\mathbb{I}_{[p_t \text{ evicted}]}$ be an indicator variable that is 1 iff page p_t is evicted. Then we have:

$$\mathrm{TCO}(\mathcal{A} + \mathcal{B}, p_t, t) = \mathrm{tic}(p_t, t) \cdot r(p_t) + \mathbb{I}_{[p_t \text{ evicted}]} \cdot B(p_t).$$

We now consider three cases depending on whether p_t was evicted by the algorithm $\mathcal{A} + \mathcal{B}$.

Case 1: p_t was not evicted. Then we have $\mathrm{buytime}(p_t, t) \geq \mathrm{tic}(p_t, t) = d_t$ and hence: $\mathrm{TCO}(\mathcal{A} + \mathcal{B}, p_t, t) = d_t \cdot r(p) = \mathrm{SkiCost}(\mathcal{B}, p_t, t)$.

Case 2: p_t was evicted because its TTL expired. Then we have $d_t \geq \text{buytime}(p_t, t) = \text{tic}(p_t, t)$. Hence, we have $\text{TCO}(\mathcal{A} + \mathcal{B}, p_t, t) = B(p_t) + \text{buytime}(p_t, t) \cdot r(p_t) = \text{SkiCost}(\mathcal{B}, p_t, t)$.

Case 3: p_t was evicted by Algorithm $\mathcal{A}$. Once again, we have $\text{tic}(p_t, t) \leq \text{buytime}(p_t, t)$ and hence $\text{SkiCost}(\mathcal{B}, p_t, t) \geq \text{tic}(p_t, t) \cdot r(p_t)$. Using this, $\text{TCO}(\mathcal{A} + \mathcal{B}, p_t, t) = B(p_t) + \text{tic}(p_t, t) \cdot r(p_t) \leq B(p_t) + \text{SkiCost}(\mathcal{B}, p_t, t)$.

Combining all three cases above and summing over all page requests, we have:

$$\text{TCO}(\mathcal{A} + \mathcal{B}, \sigma) \leq \text{SkiCost}(\mathcal{B}, \sigma) + \sum_t B(p_t) \cdot \mathbb{I}_{[p_t \text{ evicted by Algorithm } \mathcal{A}]}.$$

We note that the second term on the RHS is the total eviction cost of all pages evicted by Algorithm $\mathcal{A}$ when invoked as part of Algorithm 1. This may not be equal to the cost that would be incurred if we ran Algorithm $\mathcal{A}$ on the request sequence σ directly, i.e., $\text{EvictCost}(\mathcal{A}, \sigma) \neq \sum_t B(p_t) \cdot \mathbb{I}_{[p_t \text{ evicted by Algorithm } \mathcal{A}]}$. However, the following lemma argues that when $\mathcal{A} = \text{LRU}$, the total eviction cost incurred via this procedure does not exceed that of running LRU directly on σ. The theorem thus follows. □

Lemma 5.3. *Let $\mathcal{A} = \text{LRU}$. Then for any request sequence σ,*

$$\sum_t B(p_t) \cdot \mathbb{I}_{[p_t \textit{ evicted by Algorithm } \mathcal{A}]} \leq \text{EvictCost}(\mathcal{A}, \sigma).$$

Proof Sketch. For simplicity consider a single page p that gets evicted from the cache because its TTL expires. Let S be the cache state maintained by LRU before p is evicted and let S' be the state after the eviction. We show that for any future request sequence σ, the number of evictions incurred by LRU starting from state S' is at most that incurred by LRU starting from state S. Let S_t and S'_t denote the states after processing t requests from σ. We observe that for any t, the only page (if any) in $S_t \setminus S'_t$ is p. Further, after page p is requested or after p gets evicted from S_t, we must have $S_t = S'_t$ and hence the two instances incur the same number of evictions from that point onwards. □

We note that Lemma 5.3 and Theorem 5.2 are likely to hold (at least approximately) for many realistic cache eviction policies beyond LRU. The formal proof continues to hold for any eviction policy that maintains pages in a priority queue and the priority of a page only depends on the intrinsic characteristics of the page and its arrival time; in particular, the priority is only (re)-computed when it is requested and is independent of the past requests processed by the algorithm.

Offline Setting. Finally, we also consider the offline version of linear elastic caching where the algorithm is aware of the entire page request sequence a priori. We note that when pages have different sizes, even the special case when the cache maintenance cost is zero is known as the *generalized caching problem* and is known to be NP-complete [16]. When pages have unit sizes but different eviction costs (and zero cache maintenance costs), then the problem is known as weighted paging and admits a polynomial time offline algorithm via a reduction to minimum cost flow [8].

For offline linear elastic caching, we again show a reduction similar to Algorithm 1 and show that given two black box algorithms—one for the offline ski rental problem, and another for the offline caching problem (either weighted paging or generalized caching)—one can implement an offline elastic caching algorithm. At a high level, in the reduction, we subtract the eviction cost of any page by the cost incurred by the ski rental algorithm for that page request.

Theorem 5.4. *Let $\mathcal{A}^*$ be an offline optimal algorithm for weighted paging (or generalized caching if pages have non-uniform sizes) and let $\mathcal{B}^*$ be an offline optimal ski rental algorithm. We define a modified eviction cost $w(p_t) = B(p_t) - \text{SkiCost}(\mathcal{B}^*, p_t)$. For any caching algorithm $\mathcal{A}$, let $\mathcal{A}@\mathcal{B}^*$ be an elastic caching algorithm that runs $\mathcal{A}$ with the modified eviction costs above. Then $\mathcal{A}^*@\mathcal{B}^*$ is an optimal offline algorithm for linear elastic caching.*

Proof Sketch. We note that an optimal offline algorithm for (linear elastic) caching either evicts a page immediately after it is requested or holds it in cache until its next request. The theorem follows by arguing that for any page request p_t, the TCO incurred by $\mathcal{A}@\mathcal{B}$ exactly equals EvictCost incurred by $\mathcal{A}$ and the SkiCost incurred by $\mathcal{B}$ for that page request. If a page p_t is evicted, then the incurred TCO is $B(p_t) = w(p_t) + \text{SkiCost}(\mathcal{B}^*, p_t)$ as desired. On the other hand, if the page is not evicted, then the TCO exactly equals the ski cost incurred by $\mathcal{B}^*$. □

6 DESIGN AND IMPLEMENTATION

We propose implementing linear elastic caching via a TTL cache. Whenever a page is inserted into the cache, it gets assigned a TTL. If no further requests are made to the page, then the page gets evicted at the assigned TTL. If the page does get requested, then the TTL is recomputed. Such a TTL cache is well suited for implementing ski rental based policies since the TTL for a page directly corresponds to the time at which the ski rental algorithm decides to "buy".

The theoretical results above demonstrate that incorporating such TTLs in addition to a classical eviction policy does not deteriorate the performance of the original eviction policy. In addition, the total cost of the resulting elastic caching algorithm is at most the sum of costs experienced by the underlying cache eviction policy and the multiple independent ski rental instances constructed. Consequently, in order to minimize the elastic caching cost it suffices to optimize the ski rental algorithms independently of the cache eviction policy.

7 EVALUATION

7.1 Spanner Case Study

Spanner [9] is a scalable, globally distributed database system at Google. Like most databases, Spanner issues all storage reads through an in-memory page cache. Because Spanner employs a form of log-structured merge trees [3] for its storage layer, the data behind the cache is immutable and cache entries may be safely evicted at any time. This cache represents approximately 45% of Spanner's production memory footprint, which itself represents a substantial fraction of Google's overall fleet memory.

Spanner servers are deployed on Borg [27], Google's cluster management system. Although servers are deployed within containers with a specified memory limit, Borg overcommits memory at the

machine level [4], providing for substantial operational efficiencies within Google's fleet when we are able to reduce usage within our containers. To that end, we have integrated the linear elastic caching algorithm into Spanner, and found that it can substantially reduce Spanner's total cost of ownership (TCO).

7.1.1 Integration. In order to ensure scalability of Spanner's cache, it supports lock-free read accesses, and only enforces mutual exclusion between concurrent mutations (evictions and insertions). It is implemented using a strategy similar to BP-Wrapper [11], in which cache hits are buffered in a lock-free multi-producer single-consumer queue, and the eviction policy later processes batches of accesses while holding an exclusive latch.

This made it quite simple to integrate linear elastic caching into Spanner's cache: cache entries are inserted into a heap keyed by the eviction time computed by the ski rental oracle. When an entry is accessed, the eviction time is recomputed and the entry's position in the heap is adjusted. A background thread periodically evicts any entries that have passed their TTL. Because the TTL calculation and heap maintenance is deferred off the critical path of cache hits, we did not need to implement lock-free priority queues or be concerned about introducing additional contention on the cache implementation.

7.1.2 Learned Ski Rental Algorithm. In the classical adversarial model of online algorithms, the optimal deterministic policy is to buy at the breakeven time $\frac{B}{r}$. However, we observe that page request patterns are far from adversarial, and we can exploit this observation to learn prediction models. As observed by [18], any (potentially machine-learned) algorithm for ski rental that buys at times in some interval $[\lambda \cdot \frac{B}{r}, \frac{1}{\lambda} \cdot \frac{B}{r}]$ for any constant $\lambda \leq 1$ maintains a bounded worst-case competitive ratio. So, we consider algorithms that set $t = \lambda \cdot \frac{B}{r}$ where λ is a learned function of the page attributes.

7.1.3 Decision Tree Training. Because the ski rental policy must be consulted after every page access (billions of QPS), it is critical that it is extremely inexpensive to evaluate. We also strongly prefer interpretable models so that engineers and operators can easily compare them against their intuition of the problem domain. Given this, we use a shallow decision tree, which can be translated into a small snippet of C++ code and included into Spanner's cache implementation.

In order to train the model, we utilize a cache trace collected from a small sample of production servers. From this trace, we sample a dataset of page requests, each associated with a number of features including (i) *page features* (size, eviction cost, memory cost), (ii) *cache entry state* (e.g., time since prior request, number of cache hits), (iii) *application features* (e.g., Spanner operation category and priority). The *label* for each example instance is the (potentially-infinite) interval until the next request to the page. Categorical features are mapped into real numbers by substituting target means [24].

Unfortunately, we cannot use a standard off-the-shelf decision tree algorithm (e.g., for squared-loss regression) since we seek to directly minimize the ski rental cost, which is discontinuous and non-monotonic. Instead, we use a greedy algorithm to train the decision tree to directly minimize the total ski rental loss over the training set.

Given a set S of ski rental instances, let $\text{SkiCost}_S(\lambda)$ be the total cost over all instances in S when using a given constant value of λ. Let $\lambda^* = \arg\min_\lambda \text{SkiCost}_S(\lambda)$ be the minimizer and $\text{OPT}(S) = \text{SkiCost}_S(\lambda^*)$ be the optimal cost. We note that $\text{SkiCost}_S(\cdot)$ is piecewise linear and hence λ^* can be found by enumerating over the $O(|S|)$ pieces. In practice, we approximately find λ^* by sweeping over a fixed set of candidates in the interval $[0, 2]$.

Each node of the tree corresponds to a set S of training instances. We iterate over all features and all split points to generate a number of candidate partitionings of S. For each partition, say S_1 and S_2, we compute $\text{OPT}(S_1)$ and $\text{OPT}(S_2)$ and then choose the feature and threshold that minimizes the sum. We then recursively apply the same decision tree algorithm within each partition. In practice, we have found that there are minimal improvements past a depth of two or three, while interpretability suffers, so our production implementation only uses a depth of two, discovering optimal splits on the *time in cache* and Spanner-specific *IsBackgroundOperation* features.

7.1.4 Experimental Results. We deployed our learned ski rental policy on a significant fraction of Spanner production servers over several months. Compared to a control group that used a fixed size cache, the elastic caching policy reduces cache usage by 15.5% while only increasing the total cache misses by a relative 5.5%. Because the policy is cost-aware, the increase in cache misses is concentrated on entries with a relatively lower refault cost, and so I/O costs increase by only 0.5%. The TCO, inclusive of the cache RAM, the computation spent to maintain the policy, and the storage I/O beneath, is reduced by approximately 5%. This amounts to a substantial savings considering Spanner's exabyte-scale footprint. Given the positive results in the experimental rollout, we subsequently enabled ski rental eviction in all production Spanner databases at Google.

7.2 Public Traces

Traces and Cache Simulator. We evaluate our ski rental based eviction policies on a number of publicly available cache traces. We utilize a subset of traces collected by [28] from five datasets (Tencent, Alibaba, MSR, Cloudphysics, FIU) containing 10 traces each. We implemented our ski rental based algorithms in libCacheSim [1]. LibCacheSim is designed for high-throughput cache simulations and allows for efficient multi-threaded parallel simulations.

Algorithms. We implement two classical online algorithms for solving the ski rental instances: (i) *Breakeven* is the 2-competitive deterministic algorithm that sets the TTL of a page p to be $\frac{B(p)}{r(p)}$, (ii) *Randomized* is the randomized $e/(e-1) \approx 1.582$-competitive algorithm [17]. Since there are no application-level features available, we do not implement decision trees as the machine learned oracle. Instead, we consider the following very simple learning policy. We split each trace in half and use the first half for training. For each individual page in the training trace, we compute best TTL for the page that minimizes the cost over the training trace. We warm up all caches with one day's worth of requests from the second half of the trace and use the rest for testing and measurements. During the test trace, if we encounter a page that was seen during training, we

set the TTL to be precomputed best TTL for that page. Otherwise, we set the TTL using either the breakeven or randomized policies.

Since different pages occupy different sizes in the cache, we mainly utilize the *Greedy Dual Size Frequency* (GDSF) [7] algorithm as the cache eviction policy. We note that we replicated the results using the new S3FIFO [28] eviction policy and the results remain qualitatively unchanged. In all subsequent results, algorithms labeled as "Breakeven" implement Algorithm 1 using Breakeven as the Ski Rental algorithm and GDSF as the eviction policy, and similarly for the three other elastic caching algorithms. The algorithm labeled "GDSF" is the baseline that uses a fixed cache size and has no ski rental component.

Costs. We note that the cost of a cache eviction relative to the cost of RAM-seconds is an important factor that determines the efficacy of linear elastic caching algorithms. Internal analysis using Spanner suggests a ratio of $\approx 10^9$. This ratio ranges from $\approx 2 \cdot 10^7$ (custom VM RAM vs SSD IOPS on GCP) to $\approx 5 \cdot 10^{10}$ (Amazon S3 Express vs Standard, or Azure Premium vs Hot blob stores) in public clouds. Unless otherwise specified, we present all experimental results using a ratio of 10^9; see Figure 4 for sensitivity analysis.

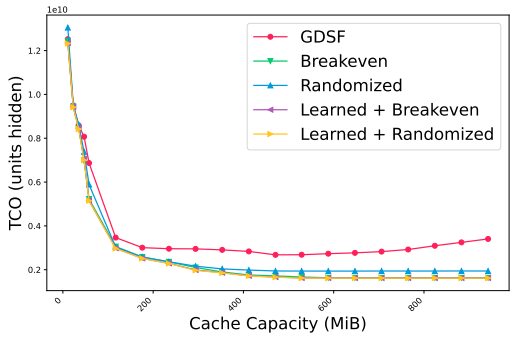

Figure 1: Variation of total cost with cache capacity.

7.2.1 Experimental Results. Figure 1 shows the TCO for a Tencent Block trace for different cache capacities. We note that the x-axis denotes the maximum cache capacity allocated rather than the actual size of the cache. Specifically, for the ski rental based policies, the actual cache used is much smaller (which contributes to the lower TCO). For small cache capacities, the cost of cache misses dominates but for larger caches, Figure 1 shows that the ski rental based algorithms significantly reduce the total cost.

To get an aggregate view of TCO improvement across the different traces, Figure 2 shows the mean percentage reduction in total cost over the different individual traces for each dataset. To compute the TCO improvement for a specific trace, we first run the algorithms with different cache capacities (as in Figure 1) and compare the best TCO obtained by GDSF with that obtained by the ski rental based policies. The figure demonstrates that the ski rental policies and especially the two learning-based policies give significant gains over using fixed cache size policies.

Effect on Miss Rate. We have assumed for the purposes of computing TCO that the cost of cache misses (or evictions) can be precisely quantified. However, a large increase in cache misses can have a significant impact on user perceived latency. Figure 3 shows a plot of the cache miss rate of GDSF for different cache sizes for the Tencent Block trace. The markers for the ski rental policies correspond to the average cache size and miss rate obtained by the corresponding algorithms when using the largest cache capacity. The figure demonstrates that the ski rental based policies incur a significantly lower miss rate for a particular average cache size than the corresponding fixed cache size policy.

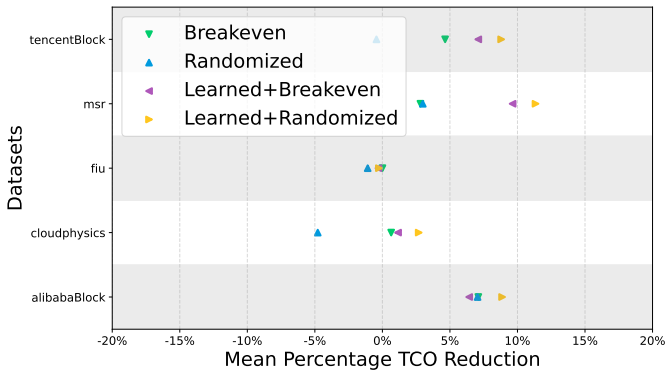

Figure 2: Mean TCO reduction per dataset (positive is better).

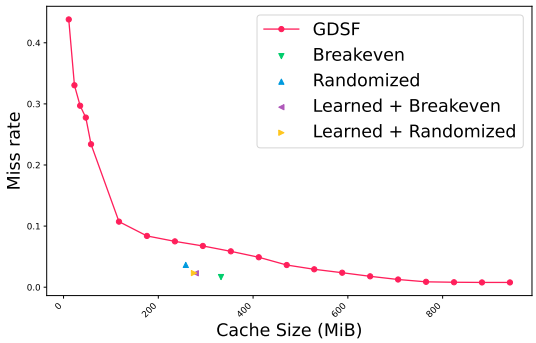

Figure 3: Variation of miss rate with cache size.

Sensitivity to the Cost Ratio. A dominant factor underlying the efficacy of elastic caching policies is the ratio of the cost of a cache eviction to the cost of RAM (per byte-second). Figure 4 shows a plot of TCO improvement for a wide range of different cost ratios for the Tencent Block trace. As the cost of evictions increases, the relative cost of maintaining the cache in RAM decreases and the gains due to elastic caching reduce.

8 CONCLUSIONS

We introduced and studied the linear elastic caching problem, motivated by reducing both the cache miss cost and the cache maintenance cost. Through an interesting connection to the ski rental problem, we developed simple and practical algorithms for linear elastic caching. These algorithms reduce the total cost of ownership on publicly available cache traces. We also propose a lightweight

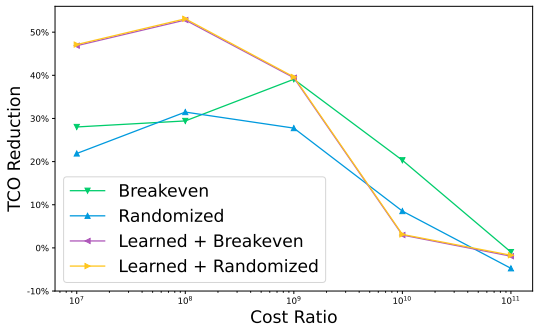

Figure 4: TCO reduction vs Cost Ratio

machine learning approach to design simple ski rental algorithms that are tailored to production workloads. They have also been deployed at scale in Spanner at Google.

Potential future work includes strengthening Theorem 5.2 for other eviction policies beyond LRU. Such a result would also provide a strong justification for incorporating learned ski rental policies along with machine learned caching algorithms.

REFERENCES

[1] libCacheSim: A high performance cache simulator. https://github.com/1a1a11a/libCacheSim. Accessed: 08-01-2024.
[2] Ski rental problem. https://en.wikipedia.org/wiki/Ski_rental_problem.
[3] David F Bacon, Nathan Bales, Nico Bruno, Brian F Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, et al. Spanner: Becoming a SQL system. In *SIGMOD*, pages 331–343, 2017.
[4] Noman Bashir, Nan Deng, Krzysztof Rzadca, David Irwin, Sree Kodak, and Rohit Jnagal. Take it to the limit: peak prediction-driven resource overcommitment in datacenters. In *EuroSys*, pages 556–573, 2021.
[5] N. Carlsson and D. Eager. Optimized dynamic cache instantiation and accurate LRU approximations under time-varying request volume. *IEEE ToCC*, 11(1), 2023.
[6] Damiano Carra, Giovanni Neglia, and Pietro Michiardi. Elastic provisioning of cloud caches: A cost-aware TTL approach. *IEEE/ACM ToN*, 28(3):1283–1296, 2020.
[7] Ludmila Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories Palo Alto, CA, 1998.
[8] Marek Chrobak, H Karloff, Tom Payne, and Sundar Vishwanathan. New results on server problems. *SIAM Journal on Discrete Mathematics*, 4(2):172–181, 1991.
[9] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM TOCS*, 31(3):1–22, 2013.
[10] Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. Caching with dual costs. In *WWW*, pages 643–652, 2017.
[11] Xiaoning Ding, Song Jiang, and Xiaodong Zhang. Bp-wrapper: A system framework making any replacement algorithms (almost) lock contention free. In *ICDE*, pages 369–380, 2009.
[12] Abdelkarim Erradi and Yaser Mansouri. Online cost optimization algorithms for tiered cloud storage services. *Journal of Systems and Software*, 160:110457, 2020.
[13] Nicaise Choungmo Fofack, Philippe Nain, Giovanni Neglia, and Don Towsley. Performance evaluation of hierarchical TTL-based cache networks. *Comput. Netw.*, 65:212–231, 2014.
[14] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Elastic caching. In *SODA*, pages 143–156, 2019.
[15] Ubaid Ullah Hafeez, Muhammad Wajahat, and Anshul Gandhi. Elmem: Towards an elastic Memcached system. In *ICDCS*, pages 278–289, 2018.
[16] Sandy Irani. Page replacement with multi-size pages and applications to web caching. *Algorithmica*, 33(3), 2002.
[17] Anna R. Karlin, Mark S. Manasse, Lyle A. McGeoch, and Susan Owicki. Competitive randomized algorithms for nonuniform problems. *Algorithmica*, 1994.
[18] Ravi Kumar, Manish Purohit, and Zoya Svitkina. Improving online algorithms via ML predictions. In *NeurIPS*, pages 9684–9693, 2018.
[19] Jeongho Kwak, Georgios Paschos, and George Iosifidis. Elastic femtocaching: Scale, cache, and route. *IEEE Trans. Wireless Comm.*, 20(7):4174–4189, 2021.
[20] Mingyu Liu, Li Pan, and Shijun Liu. To transfer or not: An online cost optimization algorithm for using two-tier storage-as-a-service clouds. *IEEE Access*, 7, 2019.
[21] Mingyu Liu, Li Pan, and Shijun Liu. Keep hot or go cold: A randomized online migration algorithm for cost optimization in STaaS clouds. *IEEE Trans. Netw. Service Mgmt.*, 18(4):4563–4575, 2021.
[22] Mingyu Liu, Li Pan, and Shijun Liu. Cost optimization for cloud storage from user perspectives: Recent advances, taxonomy, and survey. *ACM Computing Surveys*, 55(13):1–37, 2023.
[23] Yaser Mansouri and Abdelkarim Erradi. Cost optimization algorithms for hot and cool tiers cloud storage services. In *CLOUD*, pages 622–629, 2018.
[24] Daniele Micci-Barreca. A preprocessing scheme for high-cardinality categorical attributes in classification and prediction problems. *SIGKDD Explorations*, 2001.
[25] Thanasis Priovolos, Stathis Maroulis, and Vana Kalogeraki. Escape: Elastic caching for big data systems. In *SRDS*, pages 93–9309, 2019.
[26] Krishna PN Puttaswamy, Thyaga Nandagopal, and Murali Kodialam. Frugal storage for cloud file systems. In *EuroSys*, pages 71–84, 2012.
[27] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, pages 1–17, 2015.
[28] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. FIFO queues are all you need for cache eviction. In *SOSP*, pages 130–149, 2023.

A DEFERRED PROOFS

Lemma 5.3. *Let $\mathcal{A}$ = LRU. Then for any request sequence σ,*

$$\sum_t B(p_t) \cdot \mathbb{I}_{[p_t \text{ evicted by Algorithm } \mathcal{A}]} \leq \text{EvictCost}(\mathcal{A}, \sigma).$$

Proof. For simplicity, we assume that all pages have unit size; the proof extends seamlessly even for general page sizes. We recall that LRU maintains the pages held in cache in a priority queue with the most recently requested page at the head of the queue and the least recently requested page at the tail. Upon a cache miss, the newly requested page enters the queue at the head and the least recently used page at the tail gets evicted from the queue (and the cache). On a cache hit, the requested page moves from its initial position in the queue to the head. At any point during execution of the algorithm, we call this queue as the "state" of the algorithm.

The lemma states that additional evictions caused due to TTL eviction in Algorithm 1 do not increase the total cost of evictions incurred by the cache eviction policy $\mathcal{A}$. To prove this, consider a single page p that gets evicted from the cache because its TTL expires; the proof then follows by successively arguing about each page evicted due to TTL expiry. Let S be the cache state maintained (as a priority queue) by LRU before p is evicted and let S' be the state after the eviction. For any state T and request sequence σ, let $\text{LRUCost}(T, \sigma)$ be the number of evictions incurred by LRU on request sequence σ when starting from state T. Our goal is to show that $\text{LRUCost}(S', \sigma) \leq \text{LRUCost}(S, \sigma)$ for any request sequence σ.

Consider running two parallel instances of LRU on request sequence σ starting from states S and S'; we call these algorithms ALG and ALG′ respectively. We abuse notation slightly and use S_t (and S'_t) to denote the states after t pages from σ are processed. Further, let $\text{set}(S)$ denote the set of pages maintained in the cache in state S, i.e., sans priority, (and similarly for S'). We now claim that the following invariant is maintained. See Figure 5 for an illustration.

Invariant A.1. *If $S_t \neq S'_t$ then we must have $\text{set}(S_t) \setminus \text{set}(S'_t) = \{p\}$ and further either*

(1) $\text{set}(S'_t) \subset \text{set}(S_t)$, *or*
(2) $\text{set}(S'_t) \setminus \text{set}(S_t) = \{\text{tail}(S'_t)\}$, *where* $\text{tail}(S)$ *denotes the least recently used page in S.*

We first observe that this invariant is sufficient to show that $\text{LRUCost}(S', \sigma) \leq \text{LRUCost}(S, \sigma)$. Clearly, once $S_t = S'_t$, then from

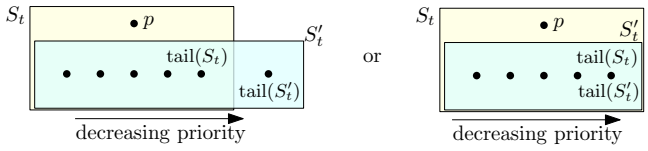

Figure 5: Illustration of Invariant A.1. The pages in the cache (apart from page $p \in S_t$) are arranged in priority order from left (highest priority) to right (lowest priority).

that point onward both algorithms perform identical actions and incur the same cost. So suppose not and consider a request to some page $x \notin S'_t$ and $x \neq p$ and suppose that ALG evicts page y. Then by the invariant that $\text{set}(S_{t+1}) \setminus \text{set}(S'_{t+1}) = \{p\}$, we must have page y has already been evicted by ALG$'$. On the other hand, if $x = p$, then we must have $p \in S'_{t+1}$ and hence by the invariant $S_{t+1} = S'_{t+1}$. Thus, the claim follows.

Finally, we prove the invariant by induction. For the base case, by definition, we have $\text{set}(S_0) \setminus \text{set}(S'_0) = \{p\}$ and $\text{set}(S'_0) \subset \text{set}(S_0)$ as desired. Let x be the $(t+1)$th page. Since $S_t = S'_t \implies S_{t+1} = S'_{t+1}$, here we assume that $S_t \neq S'_t$. We now have a few cases.

Case 1: $x \in \text{set}(S'_t) \cap \text{set}(S_t)$. Since neither algorithm incurs a cache miss, we have $\text{set}(S'_{t+1}) = \text{set}(S'_t)$ and $\text{set}(S_{t+1}) = \text{set}(S_t)$ and the invariant is maintained.

Case 2: $x \notin S'_t$ and $x \neq p$. In this case, ALG incurs a cache miss and evicts $\text{tail}(S_t)$ so that $\text{set}(S_{t+1}) = \text{set}(S_t) \cup \{x\} \setminus \text{tail}(S_t)$. Depending on whether or not S'_t has an additional page, we have two slightly different subcases. The following figure illustrates the states after processing of page x in both subcases.

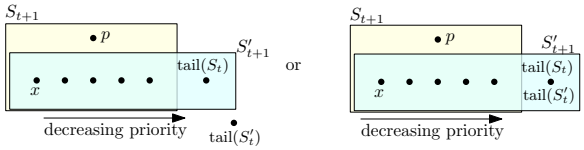

Figure 6: Illustration of how Invariant A.1 is maintained when a page $x \notin S'_t$ and $x \neq p$ is requested.

In the first subcase suppose that $\text{set}(S'_t) \setminus \text{set}(S_t) = \{\text{tail}(S'_t)\}$. Then ALG$'$ needs to evict $\text{tail}(S'_t)$ and fetch x so that $\text{set}(S'_{t+1}) = \text{set}(S'_t) \cup \{x\} \setminus \text{tail}(S'_t)$. Also since pages maintain their priority order, we now have $\text{tail}(S'_{t+1}) = \text{tail}(S_t)$ and thus maintain that $\text{set}(S'_{t+1}) \setminus \text{set}(S_{t+1}) = \{\text{tail}(S'_{t+1})\}$ as desired. This subcase is illustrated in left side of Figure 6.

For the second subcase, suppose that $\text{set}(S'_t) \subset \text{set}(S_t)$, then $|\text{set}(S'_t)| < k$ and ALG$'$ does not evict any page and $\text{set}(S'_{t+1}) = \text{set}(S'_t) \cup \{x\}$. So we maintain that $\text{set}(S_{t+1}) \setminus \text{set}(S'_{t+1}) = \{p\}$ and also have $\text{set}(S'_{t+1}) \setminus \text{set}(S_{t+1}) = \text{tail}(S_t) = \text{tail}(S'_t)$, where the last equality holds since the priority ordering of pages depends solely on their arrival order. This subcase is illustrated in right side of Figure 6.

Case 3: $x = p$. Since $p \in \text{set}(S_t)$, ALG does not incur a cache miss and we have $\text{set}(S_{t+1}) = \text{set}(S_t)$. On the other hand, ALG$'$ does incur a cache miss. Suppose we have $\text{set}(S'_t) \subset \text{set}(S_t)$, then $|\text{set}(S'_t)| < k$ and ALG$'$ does not evict any page and $\text{set}(S'_{t+1}) = \text{set}(S'_t) \cup \{p\} = \text{set}(S_{t+1})$. On the other hand if $\text{set}(S'_t) \setminus \text{set}(S_t) = \{\text{tail}(S'_t)\}$, then ALG$'$ evicts page $\text{tail}(S'_t)$. Then we have $\text{set}(S'_{t+1}) = \text{set}(S'_t) \cup \{p\} \setminus \{\text{tail}(S'_t)\} = \text{set}(S_{t+1})$. Since the priority ordering of pages depends solely on their arrival order, $\text{set}(S'_{t+1}) = set(S_{t+1})$ implies that $S'_{t+1} = S_{t+1}$ as desired. □

Theorem 5.4. *Let $\mathcal{A}^*$ be an offline optimal algorithm for weighted paging (or generalized caching if pages have non-uniform sizes) and let $\mathcal{B}^*$ be an offline optimal ski rental algorithm. We define a modified eviction cost $w(p_t) = B(p_t) - \text{SkiCost}(\mathcal{B}^*, p_t)$. For any caching algorithm $\mathcal{A}$, let $\mathcal{A}@\mathcal{B}^*$ be an elastic caching algorithm that runs $\mathcal{A}$ with the modified eviction costs above. Then $\mathcal{A}^*@\mathcal{B}^*$ is an optimal offline algorithm for linear elastic caching.*

Proof. We assume without loss of generality that $\mathcal{A}^*$ is *eager*, i.e., (i) it evicts any page p_t immediately after time t or does not evict it until the page is requested again, and (ii) it immediately evicts any page p with eviction cost 0. Note that it is easy to ensure that any offline optimal caching algorithm is eager.

We first note that for an optimal offline elastic caching algorithm $\mathcal{E}^*$, there exists an offline caching algorithm $\mathcal{A}$ such that $\mathcal{E}^* = \mathcal{A}@\mathcal{B}^*$. Indeed, we can consider $\mathcal{A}$ to be the offline algorithm that takes exactly the same decisions as $\mathcal{E}^*$.

To complete the proof, it is now sufficient to show that for any offline caching algorithm $\mathcal{A}$, if $\text{EvictCost}(\mathcal{A}, \sigma, w)$ is the total eviction cost incurred by $\mathcal{A}$ with the modified weights w, then:

$$\text{TCO}(\mathcal{A}@\mathcal{B}^*, \sigma) = \text{EvictCost}(\mathcal{A}, \sigma, w) + \text{SkiCost}(\mathcal{B}^*, \sigma).$$

Consider any page p_t requested at time t. Since $\mathcal{A}$ is eager, either p_t gets evicted immediately at the end of time t or stays in cache until it is requested again. Let d_t be the number of time units until the next request to page p_t in σ. So we have two cases:
Case 1: p_t is evicted. We have $\text{TCO}(\mathcal{A}@\mathcal{B}^*, p_t, t) = B(p_t) = w(p_t) + \text{SkiCost}(\mathcal{B}^*, p_t, t)$.
Case 2: p_t is not evicted. Since p_t gets evicted whenever $w(p_t) = 0$, we must have $\text{SkiCost}(\mathcal{B}^*, p_t, t) = d_t \cdot r(t)$. Since p_t is not evicted, we have $\text{EvictCost}(\mathcal{A}, p_t, t, w) = 0$, and $\text{TCO}(\mathcal{A}@\mathcal{B}^*, p_t, t) = d_t \cdot r(t)$. The proof follows from summing over all page requests. □