Similarity Caching

Flavio Chierichetti*
Sapienza University
Dipartimento di Informatica
Roma, 00198, Italy
chierichetti@di.uniroma1.it

Ravi Kumar, Sergei Vassilvitskii
Yahoo! Research
701 First Avenue
Sunnyvale, CA 94089, USA
ravikumar@yahoo-inc.com,
sergei@yahoo-inc.com

ABSTRACT

We introduce the *similarity caching problem*, a variant of classical caching in which an algorithm can return an element from the cache that is similar, but not necessarily identical, to the query element. We are motivated by buffer management questions in approximate nearest-neighbor applications, especially in the context of caching targeted advertisements on the web. Formally, we assume the queries lie in a metric space, with distance function $d(\cdot, \cdot)$. A query p is considered a cache hit if there is a point q in the cache that is sufficiently close to p, i.e., for a threshold radius r, we have $d(p, q) \leq r$. The goal is then to minimize the number of cache misses, vis-à-vis the optimal algorithm. As with classical caching, we use the competitive ratio to measure the performance of different algorithms.

While similarity caching is a strict generalization of classical caching, we show that unless the algorithm is allowed extra power (either in the size of the cache or the threshold r) over the optimal offline algorithm, the problem is intractable. We then proceed to quantify the hardness as a function of the complexity of the underlying metric space. We show that the problem becomes easier as we proceed from general metric spaces to those of bounded doubling dimension, and to Euclidean metrics. Finally, we investigate several extensions of the problem: dependence of the threshold r on the query and a smoother trade-off between the cache-miss cost and the query-query similarity.

Categories and Subject Descriptors

H.2.m [**Database Management**]: Systems—*Miscellaneous*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Online computation*

General Terms

Algorithms, Theory

Keywords

Caching, nearest-neighbor, buffer management, competitive analysis

1. INTRODUCTION

A caching subsystem is an integral component of any large-scale system, especially one where computing the answer anew — whether it involves reading a page off disk, as in the case of operating systems [21], or concerns buffer management issues in database systems [5, 7, 19, 23], or means recomputing the top results, as in the case of search engines [16] — is expensive. The basic paging problem is to devise a strategy to maintain a set of k pages in fast memory. At each point in time, a request arrives for a page p. If p is in the cache (*cache hit*), then it is served immediately. Otherwise (*cache miss*), the system fetches p from slower storage and serves it (and typically stores it in the cache). The seminal paper of Sleator and Tarjan [22] showed that no deterministic paging algorithm is $o(k)$-competitive and the simple LRU (Least Recently Used) strategy achieves that bound. Although a classical topic, the caching problem continues to receive attention both from practitioners [16], as well as theoreticians. On the theoretical side, there are many variations on this simple framework (cf. [2]), with several long-standing conjectures.

In our work we consider the case when the pages lie in a metric space, and a cache hit occurs if there is a "similar" page in the cache. Our primary motivation comes from caching/buffer management questions in approximate nearest-neighbor applications such as multimedia systems [8] and contextual advertising on the web [17]; we describe the latter in more detail. Today, many web pages, in particular blogs, are supported by advertisement (ad) revenue. When a user navigates to a blog, the on-line service provider (OSP), e.g., Yahoo! or Google, selects the most relevant ad to display to the user based on the user characteristics and the page content. For example, a user from a New York IP address navigating to a page about Bangkok may be shown ads by travel agents advertising cheap flights. Since the OSPs are typically paid only in the event of a click, it is in their best interest to select and show the most relevant ad.

Consider then the problem faced by an OSP — upon arrival of user u on page p, it must select the most relevant ad

*Part of this work was done while the author was visiting Yahoo! Research. The author was partially supported by the ENEA project Cresco, by the Italian Ministry of University and Research under Cofin 2006 grant Web Ram and by the Italian-Israelian FIRB project (RBIN047MH9-000).

PODS'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

to show to the user.[1] The total number of potential candidate ads is in the hundreds of millions, and the computation to select the most relevant ad is typically expensive. Thus, it is prudent to cache the results from previous visits by u to p. In the classical formulation, a cached result would only be valuable if a user with identical characteristics to u landed on a page identical to p. However, if u and v are similar (say, v is from New Jersey, whereas u was from New York), then, when v navigates to the same page p about Bangkok, the same travel agency ads are likely to be among the most relevant, and the OSP should use the cached results to its advantage. Obviously, as the similarity between u and v decreases, the results becomes less relevant, and the OSP needs to be cognizant of the threshold at which the cached result would no longer be useful.

We call the problem of caching in this framework *similarity caching*. Formally there exists a metric space $(\mathcal{X}, d)$ over the queries, and a threshold radius r. When a query p arrives, if there is a stored query q in the cache with $d(p, q) \leq r$, then the cost to the algorithm is 0, as the request can be satisfied using the cache. Otherwise, we have to compute a solution for p, at a cost of 1. Observe that if the distance between any two points in $\mathcal{X}$ is large (e.g., $> 2r$), then there is no power gained by the similarity aspect and we recover the classical caching problem. In this work we are interested in the case when some inter-point distances are less than r.

Mirroring the analysis of classical caching, we strive to develop algorithms with good competitive ratios against an oblivious adversary. While the correlation between practical performance and the competitive ratio is not always high — there are many algorithms that are $O(k)$-competitive for the paging problem, but perform much worse than LRU in practice — it remains the best tool at our disposal for analyzing on-line algorithms, in particular when nothing is known about the distribution of inputs.

1.1 Our results

Let k be the size of the cache used by OPT, and k' be the size used by the algorithm. Similarly, let r be the threshold used by OPT, and r' be the radius used by the algorithm. We begin by showing that if $k' = k$ and $r' = r$, then there can be no competitive polynomial-time similarity caching algorithms, unless P = NP (Theorem 1) . This leads us to study the relaxed version of the problem, where the algorithm is allowed to have a larger cache and/or a larger radius than that of OPT. We will use (a, b)-algorithms to denote those where $k' = a \cdot k$ and $r' = b \cdot r$ for $a, b \geq 1$.

As we vary the generality of the underlying metric space $\mathcal{X}$, we show a transition in the complexity of the problem. For general metric spaces, we show that a relaxation of a factor of 2 on the radius is necessary. On the flip side, there is a $(2, 2)$-algorithm that is 2-competitive (Theorem 3). If we restrict $\mathcal{X}$ to be a metric space of doubling dimension d, we show a $(O(2^d), 1)$-algorithm that is $O(2^d)$-competitive, and give an almost matching lower bound (Theorems 4 and 5). Finally, we show that for Euclidean spaces, we can achieve an $(O(sd), t)$-algorithm that is $\left(\frac{s}{s-1}\right)$-competitive for some value of $t \in (1, 2)$ (Theorem 7).

Next, we consider more general trade-off functions between the similarity of the query and the cached point, and the global objective. We first show that if the threshold r is query-dependent, no competitive algorithms exist even on metrics as simple as $\mathbb{R}^1$ (Theorem 9). We then relax the threshold assumption and consider smooth functions that relate total cache-miss cost to the query-query similarity. Let Δ be the ratio between the cost of computing an answer and the minimum non-zero distance between two points in the metric space. For the case of unary cache sizes we give an $O(\log \Delta)$-competitive algorithm, and a nearly tight $\Omega((\log \Delta)/(\log \log \Delta))$ lower bound (Theorems 13, 14). For larger cache sizes, we give an $O(k\Delta)$-competitive algorithm.

1.2 Related work

The similarity caching problem is somewhat related to the so-called Metrical Task Systems (MTS) problem [2]. In the MTS setting, an algorithm can be seen as a pebble in a metric space; upon the arrival of each task, the algorithm decides whether to serve the task in the state it is in, or to move to another state and then serve the task. Each task has a (possibly different) cost for each state; there is also a cost for moving the pebble: the pebble can go from the generic state a to the generic state b for a cost equal to the distance between a and b. The similarity caching problem can be modeled as an MTS by associating each state of the cache with a state of the system. The cost for moving from state a to a different state b is proportional to the set difference between a and b, specifying the number of elements we need to bring into the cache. A unit cost is paid for serving a task (i.e., answering a query) in a state whose associated cache does not contain any suitable query, while, if a suitable query does exist in the cache, the cost is 0. The main limitation of this reduction is that the number of possible states of the cache is, in principle, unbounded (as the space may be infinite). Under the assumptions that the number of points in the space bounded is by n and even assuming the state transition cost is identically 1 (that is, assuming that at each time step the cost of computing the answers of 1 or more different queries is identically 1), we could use the best known randomized algorithm [15] for the uniform MTS achieving a competitiveness of $H_m + o(H_m) = O(\log m)$, where m is the number of states in the system. Given that $m = \binom{n}{k}$, where n is the number of possible queries, this implies an $\Omega(k \log n)$-competitiveness, which would be undesirable in our setting (as n can be very large, possibly infinite).

Perhaps closer to similarity caching is the Metrical Service Systems (MSS) problem [6]: here a server must be moved in a (possibly infinite) metric space. Each request consists of a subset of the space; to serve a request, the server must be moved (if not already there) to any point of the subset. The cost incurred by the algorithm is the total distance traveled. A solution for the MSS problem is also a solution for the similarity caching problem with unitary cache size ($k = 1$).

In [10] the authors considered the k-server uniform MTS problem. The problem is a modification of the MSS: k servers are given (capturing the varying size of the cache) and the cost of moving one of them from one state to another (computing the answer of a single query) is identically 1. It is easy to see how a solution for this problem can be used as a solution for similarity caching. The solution proposed in [10] works when all the request-subsets have the same size w, and the upper bound on its competitiveness is at least $\min(k^w, w^k)$. Clearly, w can be infinite in our setting and

[1] In practice, there are many other constraints to be be considered, e.g., advertiser budgets. We ignore these here.

even in a finite metric space (e.g., the subspace induced by the queries), w could be enormous.

In [11] the authors consider the extension of the k-server problem to situations where the moving cost is a combination of distances in several metric spaces. Although similar in spirit, this approach does not model the structure of the costs that occurs in similarity caching.

The similarity caching problem is closely related to the well studied k-center clustering problem [13], in particular its on-line and streaming [3, 20] variants. In the k-center problem, we are given a set of points and are asked to find k centers that minimize the maximum distance from each point to its nearest center. In the on-line variant, the points arrive incrementally one at a time, with the goal to minimize the competitive ratio. The streaming problem adds a further constraint on the space utilization. There are two major differences between similarity caching and these versions of the k-center problem. First, the total space is severely restricted — similarity caching can only store k points in the cache! Even $O(k \log n)$ space, the holy grail of streaming algorithms, is too high. More importantly, in the world of similarity caching, we are not looking for a static set of k centers. In fact the cache points (centers) must change over time, if the algorithm is to remain competitive. Detecting when these changes occur is far from trivial.

Independent of our work, Falchi et al [8] considered similarity caching in a content-based image retrieval setting (they call it the *metric cache*). Their focus was more on the practical issues and less on the theoretical challenges introduced by caching with similarity. Motivated by our work, Pandey et al. [17] recently studied the feasibility and effect of similarity caching on a contextual ad system. Their focus was once again on the implementation and performance issues surrounding similarity caching. To the best of our knowledge, caching/buffer management issues [5, 7, 19, 23] have not been formally studied in the context of similarity search/approximate nearest-neighbors [12, 14, 1].

2. THE FRAMEWORK

Let $(\mathcal{X}, d)$ be a metric space. Let $P \subseteq \mathcal{X}$ be a given set of points in the metric space. Let $k > 0$ be the *cache size* and let $r \in \mathbb{R}^+$ be the *radius* threshold. The goal is to maintain a set of k points in the cache under the following scenario. At each time-step, a point $p \in P$ arrives. If there is a point q in the cache such that $d(p, q) \leq r$, then we have a *cache hit*, and no cost is incurred. Otherwise we have a *cache miss*, and we can issue a request to bring p into the cache (and possibly evict a member of cache to make space for p). The main evaluation metric is the *competitive ratio*, which is the ratio of the number of requests made by the on-line algorithm and the optimal offline algorithm, OPT.

It is easy to see that the paging/buffer management problem is a special instance of similarity caching: one can simply place all of the points on the vertices of an n-dimensional simplex with side length more than $2r$. Our generalization, however, comes at a price: unlike the classical paging case, it turns out that there are *no* competitive (in the usual sense) polynomial-time on-line algorithms for similarity caching. Thus we must allow the on-line algorithm to have additional power over OPT. This is possible by either relaxing the cache size or the radius or both. We hence focus on the competitive ratio of (a, b)-*algorithms*, which are permitted to use a cache of size $a \cdot k$ and a radius of $b \cdot r$, for $a, b \geq 1$.

3. ALGORITHMS & LOWER BOUNDS

We begin by showcasing the necessity of bi-criteria algorithms for similarity caching. If the algorithm is to compete against OPT using the same cache size k and radius r, then it will necessarily run into an NP-hardness roadblock, *regardless* of its competitive ratio.

3.1 Lower bounds

First, we show the NP-hardness result for arbitrary metric spaces, and then show that a similar, albeit slightly weaker, version holds even in the plane under ℓ_2 distances.

THEOREM 1. *There exist metric spaces $(\mathcal{X}, d)$ such that for any $s < \log k$, $\epsilon > 0$, and $c > 0$, no c-competitive $(s, 2 - \epsilon)$-algorithm can run in polynomial time per step, unless P = NP.*

PROOF. Consider the dominating set problem. Given a graph $G = (V, E)$ we want to decide whether there exists a subset of vertices, $V^* \subseteq V$, with $|V^*| = k$, such that for any $u \in V \setminus V^*$ there exists a $v^* \in V^*$ with $(u, v^*) \in E$. Dominating set is equivalent to the set cover problem under L-reductions. Thus, it is hard to approximate to better than $\log |V|$ unless P = NP [18].

Let $s < \log k$ and fix an $\epsilon > 0$. Let $\mathcal{A}$ be an $(s, 2 - \epsilon)$-algorithm for similarity caching. Let the cache have size k, and the request sequence be $v_1, v_2, \ldots, v_n, v_1, v_2, \ldots, v_n, v_1, \ldots$, and a single pass through all the vertices be a *phase*. If a (u, v) edge exists in G, then let the distance between u and v be 1, otherwise let the distance be 2. It is easy to see that this distance function defines a metric.

Notice that the relaxation of $2 - \epsilon$ on the radius is of no help to the algorithm. Furthermore, OPT will have no faults after the first phase, though it may have up to n faults in the first phase. If $\mathcal{A}$ is c-competitive, then after n^2 phases it has failed less than $cn + o(n)$ times. By the pigeonhole principle there exists a phase where $\mathcal{A}$ did not fault at all. The entries in the cache in this phase must form a solution to the dominating set problem. Thus, if $\mathcal{A}$ took polynomial time per step and $s < \log k$, then we can solve the decision version of the dominating set problem in polynomial time. □

While the reduction from the dominating set problem called for an arbitrary metric space, we show that even for simple metrics, like $\mathbb{R}^2$, a relaxation is necessary.

THEOREM 2. *Even if $\mathcal{X} = \mathbb{R}^2$, there exists an absolute constant $1 < r < 2$ such that for any $c > 0$, no c-competitive $(1, r)$-algorithm can run in polynomial time per step, unless P = NP.*

PROOF. The reduction here parallels the dominating set reduction above, but reduces the decision version of the *planar k-center* problem to similarity caching. Since the former is known to be hard to approximate [9], the result follows. □

3.2 Algorithms for general metrics

We now show that in the case of general metrics, the lower bound established above is almost tight, if we insist that the on-line algorithm run in polynomial time per step. We adapt the Flush-When-Full deterministic algorithm to the similarity caching model and show a $(1 + \epsilon, 2)$-algorithm that is $O(1/\epsilon)$-competitive. In particular this gives a 2-competitive $(2, 2)$-algorithm.

THEOREM 3. *For any metric, there exists a deterministic* $(s, 2)$*-algorithm that is* $\left(1 + \frac{1}{s-1}\right)$*-competitive.*

PROOF. Let r and k be, respectively, the radius and the cache size of OPT. Consider the following simple algorithm. Given a point p, if there is a point p_c in the cache such that $d(p_c, p) \leq 2r$, then answer with p_c. Otherwise, store the query point p. If the cache (having size sk) is full, then empty the cache before inserting p.

We say that a phase ends exactly before the arrival of a query that makes the algorithm empty the cache. Consider a generic phase that starts with the ith query p_i and ends with the jth query p_j. A set $S \subseteq \{p_i, \ldots, p_j\}$ of sk queries with pairwise distance greater than $2r$ must have been posed during the phase (that is, the queries cached by our algorithm). Now, consider the set $S' = S \setminus \{p_i\} \cup \{p_{j+1}\}$, $|S'| = sk$. Note that OPT needs to fault at least $sk - k$ times to cover the queries in S' (for, by the triangle inequality, if some query stored by OPT is within distance r from one of these queries, then it would be at distance more than r from all the others). It follows that OPT will fault at least $(s-1)k$ times in the sequence $p_i, \ldots, p_j$. On the other hand, the algorithm will fault sk times during a phase. □

3.3 Metrics with bounded doubling dimension

For the case of general metrics, we showed that a factor 2 relaxation on the radius is inevitable (unless P = NP), lest our algorithm take super-polynomial time to process each query. In the case when the metric $\mathcal{X}$ has a bounded doubling dimension, we can quantify this dependence almost exactly.

Recall that a metric $\mathcal{X}$ has a doubling dimension d if any ball of radius $2r$ can be covered by 2^d balls of radius r. These metrics trivially include the Euclidean metrics, but are more specific than general metric spaces as they contain some structure about the growth of the space.

We start with an algorithm for metrics of bounded doubling dimension.

THEOREM 4. *For a metric of doubling dimension* d*, there exists a deterministic* $(s2^d, 1)$*-algorithm that is* $\left(2^d \frac{s}{s-1}\right)$*-competitive.*

PROOF. The main observation is that, since the doubling dimension of the metric is d, 2^d balls of radius 1 will be enough to cover each ball of radius 2. The caching strategy will be exactly the same as in the proof of Theorem 3, but instead of caching single points, we will cache sets of 2^d of them. Thus every cache miss will require us to bring 2^d points into the cache. □

While the exponential dependence on d is not desirable, we show below that some kind exponential dependence on the doubling dimension is necessary, and hence the result above is almost tight.

THEOREM 5. *For every* $d > 6$*, there exists an* $\epsilon = \epsilon(d) > 0$ *and metrics with doubling dimension at most* d*, where no* $(s, 1+\epsilon)$*-algorithm is better than* $(2^{d-3}/s)$*-competitive. In particular, no* $(2^{d/2}, 1)$*-algorithm is better than* $(2^{d/2-3})$*-competitive.*

PROOF. Let $\epsilon = \epsilon(d) = (2s)^{-2^d/(4s)}$. We would like to show there exists a metric of doubling dimension d such that any $(s, 1+\epsilon)$-algorithm $\mathcal{A}$ is at least $t = 2^d/(8s)$-competitive. Without loss of generality, let the radius of OPT be 1.

We proceed by presenting a metric $\mathcal{X}$, in which the pairwise distance is 1 between $2t$ special pairs of points, and is at least $1 + \epsilon$ otherwise. We will then argue that any algorithm $\mathcal{A}$ as above is no better than t-competitive against an oblivious adversary.

The metric will consist of two sub-metrics, which we dub the "query, "$\mathcal{Q}$ and "cache," $\mathcal{C}$. The queries posed will come from $\mathcal{Q}$, and while one optimal point $c^* \in \mathcal{C}$ will cover them with radius 1, any other point in $\mathcal{Q} \cup \mathcal{C}$ would need a radius of at least $1 + \epsilon$ to be effective.

We begin by describing the query sub-metric, $\mathcal{Q}$. For each $i = 1, \ldots, 2t$, let Q_i consist of $2s$ unique elements, $Q_i = \{q_i^1, \ldots, q_i^{2s}\}$. Then, $\mathcal{Q} = \cup_i Q_i$. The ith query posed by the adversary will consist of one of the elements of Q_i chosen uniformly at random. For any two points $q, q' \in \mathcal{Q}$ we let the distance $d(q, q') = 2$. The doubling dimension of an arbitrary metric is upper-bounded by the logarithm of its size. Thus, the doubling dimension of $\mathcal{Q}$ by itself is bounded by $\log(4st)$.

Obviously, if we knew the sequence of points asked in advance, we would be able to have the perfect cache, and indeed the cache sub-metric, $\mathcal{C}$ contains all of the possible sequences of length $2t$. Informally, each element in $\mathcal{C}$ lies in $Q_1 \times \cdots \times Q_{2t}$. That is,

$$\mathcal{C} = \{(r_1, \ldots, r_{2t}) \mid \forall i,\ r_i \in Q_i\},$$

and $|\mathcal{C}| = (2s)^{2t}$.

Intuitively, a point $c = (r_1, \ldots, r_{2t}) \in \mathcal{C}$ states that the first point in the sequence will be r_1, the next r_2, and so on. Let $\mathcal{C} = \{c_1, \ldots, c_{(2s)^{2t}}\}$. We define the distance between any two elements $c_i, c_j \in \mathcal{C}$ to be $|i-j|\epsilon$. Observe that $\mathcal{C}$ can be isometrically embedded onto the real line and therefore has a doubling dimension of $O(1)$.

Finally, we define the distances between the two sub-metrics. For an element $q \in \mathcal{Q}$ and $c = (r_1, \ldots, r_{2t}) \in \mathcal{C}$, we define the distance as:

$$d(q, c) = \begin{cases} 1 & \text{if } q = r_j \text{ for some } j, \\ 1 + \epsilon & \text{otherwise.} \end{cases}$$

We must verify that the combined metric $\mathcal{X} = \mathcal{Q} \cup \mathcal{C}$ is indeed a metric. There are four interesting triangle inequality conditions. Below $q, q' \in \mathcal{Q}$ and $c, c' \in \mathcal{C}$. Note that

$$\begin{aligned} d(q, q') = 2 &\leq d(q, c) + d(c, q'), \\ d(q, c) \leq 1 + \epsilon &\leq d(q, q') + d(q', c), \\ d(q, c) \leq 1 + \epsilon &\leq d(q, c') + d(c, c'), \\ d(c, c') \leq 1 + (2s)^{2t}\epsilon &\leq d(c, q) + d(q, c'), \end{aligned}$$

where the last inequality holds because $2t = \frac{2^d}{4s}$, and therefore

$$(2^s)^{2t} \cdot \epsilon = (2^s)^{2t} \cdot (2^s)^{-2t} = 1.$$

Finally the doubling dimension of $\mathcal{X}$ is at most $\log(4st)+1 = \log(8st)$ (as, in general, the doubling dimension of the union of k metrics is upperbounded by their maximum doubling dimension plus $\log k$). Thus $t \geq 2^d/(8s)$.

Given the metric space $\mathcal{X}$, the oblivious adversary chooses a sequence $q_1, \ldots, q_{2t}$ by choosing q_i uniformly at random

from Q_i. By construction there exists a single point $c^* \in \mathcal{C}$ that is distance 1 from all points $q_1, \ldots, q_{2t}$. Any algorithm $\mathcal{A}$ though, must commit to points $\{c_1, \ldots, c_s\}$, where each $c_i \in \mathcal{C}$. Once committed, the probability that one of these points will be within a distance of 1 from a randomly selected q_i is at most $1/2$. Therefore, it will fail at least t times in expectation. □

The above lower bound can be strengthened to the case even if OPT is allowed to insert Steiner points (i.e., points that are never queried) in the cache. We omit the details in this version.

3.4 Euclidean metrics

Recall that unless we relax the radius, even in Euclidean spaces, the algorithm cannot run in polynomial time per step, unless P = NP. We further show that even when the algorithm cache size is increased vis-à-vis the optimal, the exponential dependence (at least on the dimension) persists if the radius is not increased.

THEOREM 6. *Fix $d > 0$, and consider the d-dimensional Euclidean space under the ℓ_2 distance measure. Then for any $s > 1$, there exists an input sequence such that no randomized $\left(s, 1 + \Theta\left(\frac{\log s}{d}\right)^{1/6}\right)$-algorithm is better than $\Theta\left(\frac{d}{\log s}\right)^{\frac{1}{3}}$-competitive.*

PROOF. As before, OPT will use a cache size of 1. Let c be a sufficiently large integer, $\kappa = 2c$ and $n = c\kappa^2 \log(2s)$. Finally, let the dimensionality of the space be $d = n\kappa$, and the radius used by OPT, $r^* = \sqrt{n(1 - 1/\kappa)}$.

The adversary will pose κ queries in total. The query i will have a value of 0 in all coordinates, except $ni, ni + 1, \ldots, ni + (n-1)$. These coordinates will be chosen uniformly at random between $\{-1, +1\}$. In other words,

$$q_i = (0, \ldots, 0, \underbrace{\pm 1, \ldots, \pm 1}_{ni, \ldots, ni+n-1}, 0, \ldots, 0),$$

where each sign is chosen uniformly at random.

Let $p^* = \frac{1}{\kappa} \sum_{i=0}^{\kappa-1} q_i$ be the point selected by OPT. Note that p^* is always the same sign as the query point in the relevant dimensions. Thus, for any query q_i,

$$\begin{aligned} d^2(q_i, p^*) &= n\left(1 - \frac{1}{\kappa}\right)^2 + (d - n)\left(\frac{1}{\kappa}\right)^2 \\ &= n\left(1 - \frac{1}{2\kappa} + \frac{1}{\kappa^2}\right) + \frac{n(\kappa - 1)}{\kappa^2} \\ &= n\left(1 - \frac{1}{\kappa}\right) = r^{*2}. \end{aligned}$$

Now consider the choice of the algorithm, and let v be any point present in the cache. We will show that with sufficiently high probability v will not cover q_t. Let Q_t be the set of non-zero coordinates of q_t. For clarity we will denote by v_i the value of v in the ith coordinate.

Without loss of generality, we may assume that $|v_i| \leq 2$ for all i. Note that for each non-zero coordinate j of q_t, the square of the difference between v and q_t will be $(1 - |v_j|)^2$ if v_j agrees in sign with the jth coordinate of q_t, and $(1 + |v_j|)^2$ otherwise. Let $R \subset Q_t$ be the set of coordinates where the signs of v and q_t agree, and $\bar{R} = Q_t \setminus R$ be the set where they disagree. Then,

$$\begin{aligned} d^2(v, q_t) &\geq \sum_{j \in R} (1 - |v_j|)^2 + \sum_{j \in \bar{R}} (1 + |v_j|)^2 \\ &= \sum_{j \in R} (1 - 2|v_j| + v_j^2) + \sum_{j \in \bar{R}} (1 + 2|v_j| + v_j^2) \\ &= n(1 + v_j^2) - 2\left(\sum_{j \in R} |v_j| - \sum_{j \in \bar{R}} |v_j|\right). \end{aligned}$$

Let Y denote the random variable $\sum_{j \in R} |v_j| - \sum_{j \in \bar{R}} |v_j|$. Note that if $Y \leq \frac{n}{4\kappa} = \Theta(c\kappa \log(s)) = c'\kappa \log s$, then the squared distance is at least $n(1 - 1/2\kappa)$; that is, the distance is at least $(1 + \Omega(\kappa)^{-1/2})r^* = (1 + \Omega(d/\log s)^{-1/6})r^*$, i.e., v will not cover q_i.

It is easy to see that the expected value of Y is zero and we will use Azuma's inequality to show that a large deviation from the expectation is unlikely. Notice that Y is determined by a sequence of at most n coin flips, and that by changing the outcome of any one of these flips, the value of Y can change by at most 9. Therefore,

$$\begin{aligned} &\Pr[Y - E[Y]] \geq c'\kappa \log(2s)] \\ &\leq \exp\left(-\frac{(c'\kappa \log(2s))^2}{2 \cdot 9^2 n}\right) \\ &\leq \exp\left(-\Theta\left(\frac{c'^2 \kappa^2 \log^2(2s)}{c\kappa^2 \log(2s)}\right)\right) \\ &\leq \exp\left(-\log(2s)\right) \\ &= 1/(2s). \end{aligned}$$

At most s such experiments will be performed for each query (one for each vector in the cache of the algorithm). Thus, by the union bound, the probability that the algorithm gets a cache hit at the arrival of a new query is at most $1/2$. Thus, in expectation, the number of times that the algorithm will fault in a single phase is at least $\kappa/2 = \Theta((d/\log s)^{1/3})$. □

On the positive side, we show that in the Euclidean space, a relaxation of a factor of 2 on the radius is not necessary, and, in low dimensional spaces, one can obtain sub-exponential, competitive algorithms for similarity caching.

THEOREM 7. *Fix, $0 < d < 16$, and consider the d dimensional Euclidean space under the ℓ_2 distance measure. For each d, there exists a constant $t = t(d) \in (1, 2)$ and a $(2sd, t)$-algorithm that is $\left(2d\frac{s}{s-1}\right)$-competitive.*

PROOF. The crux of our proof will boil down to covering a single ball of radius $2r$ by $2d$ balls of radius strictly less than $2r$. Notice that, once we achieve such a construction, we can easily modify the caching scheme described in Theorem 3 to achieve the desired result.

Assume, without loss of generality, that we need to cover a ball of radius $2r$ centered at the origin. Let $\{e_1, \ldots, e_d\}$ be the elementary unit vectors, and consider the set of balls of radius tr centered at $\{\pm r \cdot e_i\}$. Now consider a point p lying in the $2r$-ball centered at the origin. Again, without loss of generality, let p have its largest magnitude in its first coordinate. To maximize the distance from one of the selected points, in particular, from the point $q = (1, 0, \ldots, 0)$, p must have the form $(x, y, y, \ldots, y)$, such that $x \geq y$ and $x^2 + (d-1)y^2 = 4r^2$.

$$\begin{aligned} d^2(p,q) &= (x-r)^2 + (d-1)y^2 \\ &= (x^2 - 2xr + r^2) + (4r^2 - x^2) \\ &= 5r^2 - 2xr. \end{aligned}$$

To maximize the expression above we must minimize x subject to $x \geq y$, in which case we obtain $x = y = 2r/\sqrt{d}$. Plugging this in, we obtain:

$$d(p,q) = \sqrt{5r^2 - \frac{4r^2}{\sqrt{d}}} \quad \text{and} \quad t(d) = \sqrt{5 - \frac{4}{\sqrt{d}}},$$

the latter of which is strictly less than 2 when $d < 16$. □

4. SOME GENERALIZATIONS

In this section we consider several extensions to the similarity caching problem presented above. Recall our motivating application involving approximate nearest-neighbors, viz., showing contextual ads for web pages. By making the threshold parameter r universal, we have made an implicit assumption that all queries are equally important, which need not be the case. It is easy to imagine scenarios where the threshold is query dependent, and the exact value at which we no longer consider two queries similar is different. We begin by tackling this problem, but show that the situation is rather bleak: even in the very simple metric $\mathbb{R}^1$, no constant competitive algorithms exist.

We then turn our attention to the threshold itself. While we have studied the problem with a hard bound on the similarity, one can imagine a smooth function that trades off the utility (to the OSP) with the user-user similarity. If the function is linear in similarity, then the caching algorithm solves a "k-median" version of the problem. However, as we remarked earlier, the problem is harder than the classical k-median, and even for $k = 1$, the situation is non-trivial. We present an almost optimal algorithm, together with matching lower bounds for the case of $k = 1$, and describe a simple algorithm for larger values of k.

4.1 Query-dependent thresholds

So far we have assumed that the threshold r is universal, but suppose that the similarity matters more for some queries than for others, that is different queries have different thresholds. Now, at each step, the algorithm receives both a query q and a radius r_q. An (a,b)-algorithm can answer q with any query at a distance at most $b \cdot r_q$. Obviously, OPT is forced to answer q with a query at a distance at most r_q from it.

Let us define the maximum ratio between radii for the set of queries $\mathcal{Q}$,

$$\rho = \frac{\sup_{q \in \mathcal{Q}} r_q}{\inf_{q \in \mathcal{Q}} r_q}.$$

THEOREM 8. *Fix $\rho < \infty$. If the value of ρ is known, then there exists a deterministic $(s, 1+\rho)$-algorithm that is $\left(1 + \frac{1}{s-1}\right)$-competitive.*

PROOF SKETCH. We just need to modify the algorithm of Theorem 3 so that a query q is answered with something in the cache if their distance is at most $(1+\rho)r_q$. Then, again by the triangle inequality, we show that in each phase OPT has to fault at least $sk - k + 1$ times. □

We now show that, for a competitive algorithm to exist, the ratio ρ has to be bounded.

THEOREM 9. *Even in $\mathbb{R}^1$, for any $a, b > 1$, no (a,b)-algorithm is better than $\left(\frac{1}{4}\log_{4ab}\rho\right)$-competitive.*

PROOF. Let us use the metric $Q_0 = [0,1] \subset \mathbb{R}^1$ with any ℓ distance. Say that OPT has a radius of 1 and a cache that holds 1 element.

Informally, we select a bounded subset $Q_1 \subseteq Q_0$ of diameter $(4ab)^{-1}$. The query q_1, with radius $T_1 = (4ab)^{-1}$, will be the leftmost point of Q_1. The ith query, $i \geq 1$, will be the leftmost point of a bounded subset $Q_i \subseteq Q_{i-1}$ of diameter $(4Ks)^{-i}$. Its radius will be equal to the diameter of Q_i.

Formally, to select the subsets, given a bounded $Q_{i-1} = [x,y)$, $i \geq 1$, take the following subsets of it: for each $0 \leq j < 4a$, let the jth subset S_{i-1}^j be equal to

$$S_{i-1}^j = \left[(y-x)\frac{js}{4ab} + x, (y-x)\frac{jb+1}{4ab} + x\right),$$

i.e., the first subset will have its left endpoint in x and it will span for a fraction of $(4ab)^{-1}$ of the diameter of Q_{i-1}; then a fraction of $(b-1)/(4ab)$ of that diameter will be left uncovered; after that the second subset will begin, and so on.

The subset Q_i will be chosen u.a.r. in $\{S_{i-1}^0, \ldots, S_{i-1}^{4a-1}\}$. Without loss of generality, let $Q_i = S_{i-1}^j$; the query q_i will be the left endpoint of Q_i, i.e., $q_i = (y-x)\frac{j}{4ab} + x$.

Given the radius $T_i = (4ab)^{-i}$, the algorithm A, using each single element of its cache, can intersect at most 3 of the subsets $S_{i-1}^1, \ldots, S_{i-1}^{4a-1}$ (as their minimum distance is $(4a)^{-i} - (4ab)^{-i}$). The number of elements of the cache is a, so A will be able to cover at most $3a$ of the $4a$ different S_{i-1}^j sets. Thus, with probability at least $1/4$ it will miss the one chosen by the adversary.

Now, OPT faults a single time. Let the overall number of queries be equal to $4c$, thus $\rho = (4ab)^{4c}$. Note that the last query q_{4c} is able to answer all the queries. Thus OPT would not fault more than once. On the other hand, A will fault c times in expectation. The claim follows. □

The previous lower bound also implies that, for the competitiveness to be independent of ρ, one must have that ab is at least polynomial in ρ.

4.2 Smoother trade-off functions

In the similarity caching model that we have employed, we are asked to maximize the total number of cache hits. However, not all cache hits are created equal — the utility of the similarity cache is greater if all of the cache hits occur with points much closer than the threshold r. In this section we consider the problem when the exact trade-off is linear, but our results generalize to more complex trade-off functions. Assume without loss of generality that the cost of computing the optimal solution is 1, and the cost of using point p to service query q is $d(p,q)$. Then the objective of the algorithm is to minimize the total cost incurred by the caching. We assume that while the metric space itself may be infinite, the minimum distance between any two different points is at least $1/\Delta$.

4.2.1 $k = 1$

We begin by presenting the algorithm for the case when $k = 1$. Note that this case is simple in classical caching;

indeed, there is really no choice for the algorithm to make. In similarity caching, however, the situation is far from trivial. Our algorithm proceeds in phases, with the invariant that the OPT solution pays at least $\Omega(1)$ in every phase, while the algorithm pays at most $O(\log \Delta)$. We further break up each phase into exponentially increasing sub-phases — sub-phase i is of length 2^i. At the end of each sub-phase the algorithm recomputes the optimum cache point. This allows the algorithm to zoom in on the optimum solution inside each phase, trading off the cost of serving queries with a suboptimal cache point and the cost of bringing a new point to the cache every time. The algorithm is presented below.

Algorithm 1 An $O(\log \Delta)$ algorithm for $k = 1$

```
1: Begin new phase with query q
2: b ← 0, r ← 1, C ← q, Q ← {q}.
3: for each new query q do
4:    Q ← Q ∪ {q} (Q is a multiset).
5:    if d(C, q) ≥ 2r then
6:       b ← b + d(C, q) − r
7:    if b ≥ 1/4 then
8:       End current phase. Begin new phase with q
9:    if |Q| ≥ 1/r and r ≥ 1/(2Δ) then
10:      Begin new sub-phase
11:      r ← r/2, C ← Cluster(Q)
12:      if Cost(Q, C) ≥ 1/4 then
13:         End current phase. Begin new phase with q.
```

The function *Cluster*(Q) returns an α-approximation to the optimal cache point for Q. We use a streaming algorithm for k-median to approximately compute *Cluster*(Q) for small constant α, see for example [4]. Note that we *only* change the cache point at the end of each subphase, *not* every time the solution to *Cluster*(Q) changes.

The analysis is centered around the following "zooming in" lemma. Intuitively it says that the distance between any two solutions, each of small cost, decreases inversely with the total number of points seen. In particular, the distance between any two solution of cost less than $1/4$ decreases as $O(1/n)$.

To simplify the notation, for a point set X and solution point z denote by $C(X, z)$ the total cost of serving X using z: $C(X, z) = \sum_{x_i \in X} d(x_i, z)$.

LEMMA 10. *Let $X = \{x_1, \ldots, x_n\}$ be a point set, and z be any point such that $C(X, z) \le t$. Let c^* be another point s.t. $d(X, c^*) \le d(X, z) \le t$. Then $d(c^*, z) \le 4t/n$.*

PROOF. Let $r = 2t/n$ and let $Y \subseteq X$ so that for any $y \in Y$, $d(y, c^*) \le r$. Note that $|Y| \ge n - t/r$. Otherwise, the total cost of servicing points in $X - Y$ is $\sum_{x \in X-Y} d(x, c^*) > (t/r) \cdot r \ge t$, a contradiction to the total cost of c^*. Since $C(Y, z) \le C(X, z) \le 1/(4t)$, by averaging, there is a $y \in Y$ such that $d(y, z) \le 1/(4t|Y|)$. From the definition of Y, we also know $d(c^*, y) \le r$. Applying the triangle inequality

$$d(c^*, z) \le d(c^*, y) + d(y, z) \le r + \frac{1}{4t|Y|} \le r + \frac{2t}{n} = \frac{4t}{n}.$$

□

To prove the competitive ratio of the algorithm we proceed in two parts. We first establish that the algorithm pays at most $O(\log \Delta)$ per phase, and then use use Lemma 10 to show that OPT pays $\Omega(1)$ per phase.

LEMMA 11. *In every phase the algorithm pays at most $O(\log \Delta)$.*

PROOF. We first prove a simpler claim, showing that in every sub-phase the algorithm pays at most $O(1)$. Consider an individual sub-phase and let B be a ball of radius $2r$ centered around the cached point C. We divide the query points into those falling inside the ball B, denoted by, Q^{in} and those falling outside, denoted by Q^{out}. For each of the Q^{in} queries, the cost is at most $2r$. Since each sub-phase consists of at most $1/r$ points, the total cost is at most $O(1)$. Now consider the queries in Q^{out}, each of these queries q, will contribute at least $d(C, q) - r$ to b. Since $d(C, q) \ge 2r$, the algorithm's cost is at most twice the contribution to b. Since the phase ends when b reaches $1/4$, the total cost will be at most $O(1)$.

Finally, observe that r is halved at the beginning of each sub-phase. Since the minimum inter-point distance is $1/\Delta$, after $\log \Delta + 1$ sub-phases each query will be either answered exactly with C, or contribute to b, thereby starting a new phase. □

LEMMA 12. *In every phase the optimum solution pays at least $\Omega(1)$.*

PROOF. Observe that whenever OPT brings a new point into the cache, its cost increases by exactly 1, thus we can focus on the case when the optimum solution does not change during a phase. A new phase begins in one of two cases. Either *Cost*$(Q, C) \ge 1/4$, at which point the optimum must have paid at least $1/(4\alpha) = \Omega(1)$, or $b \ge 1/4$. Consider the points in Q, where for each $q \in Q$ at the time of its arrival, $d(C, q) \ge 2r$. Lemma 10 with $t = 1/4$, tells us that any solution with cost no more than $1/4$ must be within r of the algorithm's cache point. Therefore, any optimal solution pays at least $d(C, q) - r$ to cover each point $q \in Q$. Thus when a new phase begins, since $b \ge 1/4$, any optimal solution must have paid at least $b = \Omega(1)$. □

The following theorem then follows directly from the last two lemmas.

THEOREM 13. *Algorithm 1 is $O(\log \Delta)$-competitive.*

The algorithm presented above is almost optimal.

THEOREM 14. *For $k = 1$, no algorithm can have a competitive ratio better than $\Omega\left(\frac{\log \Delta}{\log \log \Delta}\right)$.*

PROOF. Let $f(x)$ be defined as $f(x) = (x \ln^2 x)^{-1}$. Fix a sufficiently large integer n. Our metric will be composed of a subset X of $\mathbb{R}^{n-1}$ with the ℓ_∞ norm. We define X as

$$X = \{(\sigma_2 \cdot f(2), \ldots, \sigma_n \cdot f(n)) \mid (\sigma_2, \ldots, \sigma_n) \in \{-1, 0, 1\}^{n-1}\}.$$

The minimum distance in our metric is $1/\Delta \ge \Omega(f(n))$ and thus $\log \Delta = \Theta(\log n)$.

At the beginning of each phase, the adversary will choose a uniform random "seed" $(\sigma_2, \ldots, \sigma_n)$ in $\{-1, 1\}^{n-1}$; a phase will consist of $n - 1$ queries, $q_2, \ldots, q_n$. Given the random seed, the generic query q_i will be

$$q_i = (\sigma_2 \cdot f(2), \sigma_3 \cdot f(3), \ldots, \sigma_{i-1} \cdot f(i-1), \sigma_i \cdot f(i), 0, \ldots, 0).$$

Consider the following algorithm $\mathcal{A}$. At the beginning of a phase, $\mathcal{A}$ foresees q_n, and store its answer. We upperbound the performance of OPT by that of $\mathcal{A}$.

Since we use the ℓ_∞ distance, $\mathcal{A}$ will pay $f(i+1)$ for the ith query. Therefore, its total payment per phase (denoted T) will be

$$T \leq 1 + \sum_{i=3}^{\infty} f(i) \leq 1 + f(3) + \int_3^{\infty} f(x) = O(1),$$

where the second step holds as $f(x)$ is decreasing for $x > 1$.

Before considering the on-line algorithm $\mathcal{A}$, let us divide the phase into sub-phases. Sub-phase 1 goes from query q_{k_0} to q_{k_1-1}, sub-phase 2 goes from query q_{k_1} to query q_{k_2-1} and so on, where $k_0 = 2$ and $k_{i+1} = k_i + \lceil 1/f(k_i) \rceil = k_i + \lceil k_i \log^2 k_i \rceil$. Note that, for a sufficiently large constant c, we have $k_{i+1} \leq k_i\, c \log^2 k_i$. For $k_i \leq n$, we have

$$k_{i+1} \leq k_i\, c \log^2 n \leq (c \log^2 n)^{i+1} k_0 \leq (2c \log^2 n)^{i+1}.$$

Thus, the number of sub-phases in a phase is seen to be at least $\Omega(\log n / \log\log n)$.

Now consider the generic on-line algorithm $\mathcal{A}$. Fix a sub-phase i, going from $q_{k_{i-1}}$ to q_{k_i-1}. With probability $1/2$ (where the probability is over the random choice of the seed), the point in $\mathcal{A}$'s cache at the beginning of the phase will be at distance at least $f(k_{i-1})$ from each of the queries of the phase. Now, $\mathcal{A}$ may fault during the sub-phase (paying 1) or it may not. In the latter case, $\mathcal{A}$ will pay the total distance between its cache point and the queries. As the length of the ith sub-phase is at least $1/f(k_{i-1})$, the expectation of $\mathcal{A}$'s payment for the sub-phase will be $\Omega(1)$.

There are $\Omega(\log n / \log\log n)$ sub-phases per phase, and each of them costs $\Omega(1)$ in expectation to $\mathcal{A}$. On the other hand, OPT pays $O(1)$ per phase. As $\log n = \Theta(\log \Delta)$, the proof is complete. □

4.2.2 $k > 1$

The case of $k > 1$ is a generalization of the on-line k-median problem (in the same way that the threshold similarity caching problem is a generalization of the on-line k-center problem). It is easy to see that the lower bounds from classical caching apply here as well, so we cannot hope to get a better than a $\Omega(\log k)$-competitive algorithm, or $\Omega(k)$ deterministic competitive algorithm. The simple Flush-When-Full algorithm achieves this ratio when $\Delta = O(1)$; more generally, it leads to an $O(k\Delta)$-competitive ratio. To see this, observe that for any set of $k+1$ non-unique points the total cost paid by OPT is at least $\frac{1}{\Delta}$, whereas the cost for the algorithm will be $O(k)$. Improving upon this bound is non trivial, and finding an $O(\log k + \log \Delta)$-competitive algorithm remains a challenging open problem.

5. DISCUSSION

The notion of similarity caching might strike some as a little strange, after all, the objective of classical caching is to deliver the same exact result. However, the approach of trading off correctness for speed (i.e., performance) is a recurring theme in database applications, particularly involving nearest-neighbor search. Such tradeoffs are most helpful when the computing an approximation gives a large performance boost, while having a negligible impact on the accuracy of the system. Since one of our motivating applications was contextual ads in web search, we adapt and discuss simple experimental results obtained in [17] that demonstrate two key features about similarity caching, namely, it does not hurt accuracy and it can actually improve the performance.

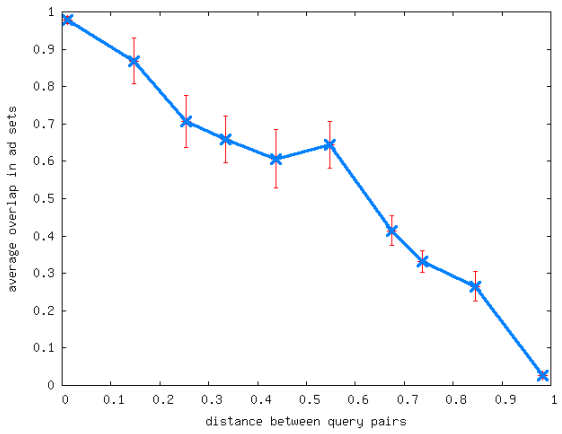

Figure 1: Distance between queries vs Jaccard overlap of the sets of ads chosen for those queries.

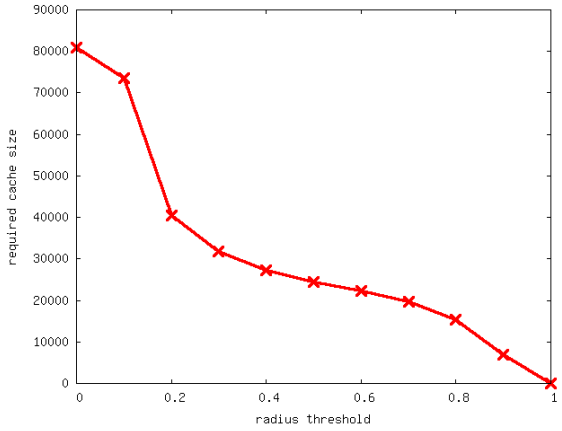

Figure 2: The minimum size of the cache required to "cover" all 100K pages, as a function of the radius threshold.

Recall the contextual advertisement application described earlier, where an OSP receives a request for an advertisement when a user loads a page. Our dataset consists of a time-ordered sequence of eight million such ad requests. Each page request can be thought of as a weighted feature vector that takes all of the relevant information into account; the underlying metric space is weighted Jaccard. The system then returns a set of ads deemed most relevant for the request. We note that obtaining the optimal result would require probing the ad database, and is an expensive operation that contributes a great deal of latency to the time-sensitive system. In Figure 1 we plot the average Jaccard overlap in the top 1000-ad sets vis-à-vis the distance between two queries (i.e., two page requests). We see that at 0.15 distance, ad sets exhibit almost 90% overlap. Thus replying with a cached point at distance within 0.15 will lead us to a near-optimal answer.

While we have shown that nearby queries lead to similar results, it could still be the case that all of the query points are very well separated, and thus similarity caching

would not be effective. We perform the following simple experiment, taking the first 100K requests, we ask what is the minimum size of the cache needed to answer all of these queries with a particular radius threshold. The results are shown in Figure 2. As we expect, the curve is monotonic, as we increase the radius threshold, the size of the cover decreases. However, notice the curve is not linear, and indeed a small increase in the radius threshold yields an almost factor of 2 savings on the size of the optimal cache. Thus similarity caching in this system could improve the performance.

6. CONCLUSIONS

In this paper we introduced the *similarity caching* problem in which a caching algorithm can return an element from the cache that is similar, but not necessarily identical, to the query element. Our main motivation was buffer management questions in approximate nearest-neighbor applications such as retrieval of contextual advertisements. Similarity caching is a strict generalization of classical caching, and we showed that unless the algorithm is allowed extra power over the optimal offline algorithm, the problem is intractable. We then proceeded to quantify the hardness as a function of the complexity of the underlying metric space and showed that the problem becomes easier as we proceed from general metric spaces to those of bounded doubling dimension, and to Euclidean metrics. Finally, we investigated several extensions of the problem, including a smoother trade-off between the cache-miss cost and the query-query similarity. Interesting future work includes obtaining improved algorithms for the Euclidean setting and for $k > 1$ in the smooth trade-off function case.

Acknowledgments

We thank Andrei Broder for suggesting us this problem and Vanja Josifovski, Sandeep Pandey, and Raghu Ramakrishnan for helpful discussions.

7. REFERENCES

[1] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *Proc. 7th International Conference on Database Theory*, pages 217–235, 1999.

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

[3] M. Charikar, C. Chekuri, T. Feder, and R. Motwani. Incremental clustering and dynamic information retrieval. *SIAM Journal on Computing*, 33:1417–1440, 2004.

[4] M. Charikar, L. O'Callaghan, and R. Panigrahy. Better streaming algorithms for clustering problems. In *Proc. 35th Annual ACM Symposium on Theory of Computing*, pages 30–39, 2003.

[5] H. T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. *Algorithmica*, 1(3):311–336, 1986.

[6] M. Chrobak and L. L. Larmore. Metrical service systems: Deterministic strategies. Manuscript.

[7] W. Effelsberg and T. Härder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4):560–595, 1984.

[8] F. Falchi, C. Lucchese, S. Orlando, R. Perego, and F. Rabitti. A Metric Cache for Similarity Search. In *Proc. 6th Workshop on Large-Scale Distributed Systems for Information Retrieval*, 2008.

[9] T. Feder and D. H. Greene. Optimal algorithms for approximate clustering. In *Proc. 20th Annual ACM Symposium on Theory of Computing*, pages 434–444, 1988.

[10] E. Feuerstein. Uniform service system with k-server. In *Proc. 3rd Latin American Symposium on Theoretical Informatics*, pages 23–32, 1998.

[11] M. Flammini and G. Nicosia. On multicriteria online problems. In *Proc. 8th Annual European Symposium on Algorithms*, pages 191–201, 2000.

[12] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. 25th International Conference on Very Large Data Bases*, pages 518–529, 1999.

[13] T. F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[14] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. 30th Annual ACM Symposium on the Theory of Computing*, pages 604–613, 1998.

[15] S. Irani and S. S. Seiden. Randomized algorithms for metrical task systems. *Theoretical Computer Science*, 194(1-2):163–182, 1998.

[16] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *Proc. 12th International Conference on World Wide Web*, pages 19–28, 2003.

[17] S. Pandey, A. Broder, F. Chierichetti, V. Josifovski, R. Kumar, and S. Vassilvitskii. Nearest-neighbor caching for content-match applications. In *Proc. 18th International Conference on World Wide Web*, 2009.

[18] R. Raz and S. Safra. A sub-constant error-probability low-degree test, and sub-constant error-probability PCP characterization of NP. In *Proc. 29th Annual ACM Symposium on Theory of Computing*, pages 475–484, 1997.

[19] G. M. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM Transactions on Database Systems*, 11(4):473–498, 1986.

[20] A. Sharp. Thoughts on the competitive ratio. Manuscript.

[21] A. Silberschatz and P. B. Galvin. *Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[22] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

[23] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.