Taking two Birds with one k-*NN* Cache

Damiano Carra
University of Verona
damiano.carra@univr.it

Giovanni Neglia
Inria, Université Côte d'Azur
giovanni.neglia@inria.fr

***Abstract*—k-Nearest Neighbors aims at efficiently finding items close to a query in a large collection of objects, and it is used in different applications, from image retrieval to recommendation. These applications achieve high throughput combining two different elements: 1) approximate nearest neighbours searches that reduce the complexity at the cost of providing inexact answers and 2) caches that store the most popular items.**

In this paper we propose to combine the approximate index for the whole catalog with a more precise index for the items stored in the cache. Our experiments on realistic traces show that this approach is doubly advantageous as it 1) improves the quality of the final answer provided to a query, 2) additionally reduces the service latency.

I. Introduction

The availability of vast amount of data today requires diversified ways for sifting through it. Exact search is just one option, while other approaches, such as *similarity search* [1], offer the possibility to explore the data to enrich *user experience*. Many applications, such as image retrieval systems [2], contextual advertising systems [3], and recommendation systems [4] are indeed based on similarity search, which in turn is usually implemented using k-Nearest Neighbor (k-*NN*) [5], [1]. In k-*NN*, the instances are represented as points (called *embeddings*) in a metric space, and a function measures how close a new instance is to the stored ones. Upon a query, k-*NN* provides an answer by retrieving the k closest instances in the dataset and opportunely processing them.

The simplicity and flexibility of k-*NN* algorithms come at the cost of (i) permanently storing the whole set of instances (or at least their embeddings) and (ii) performing expensive searches upon each query. In particular, exact k-*NN* in high-dimensional metric spaces is unfeasible for large databases, as the search practically reduces to a linear scan [6]. The widespread use of k-*NN* has then motivated research on Approximate Nearest Neighbor (k-*ANN*) algorithms [7], [8] and the development of general-purpose k-*ANN* libraries [9], including Facebook FAISS [10] and Microsoft SPTAG [11]. k-*ANN* algorithms trade off accuracy against search speed: they find in a short time k close neighbors, but not necessarily the closest ones. They also trade off accuracy versus memory requirements: they need to store opportune data-structures in a fast memory (RAM or GPU memory) with larger data-structures allowing more precise answers.

While k-*ANN* works on instances' embeddings, when instances are large *multimedia contents*, their retrieval is also time-consuming Instances are stored in a database that usually employs inexpensive, but slow, disks. A common network architectural component to speed up the retrieval is a fast, expensive, smaller storage memory usually referred to as *cache*. Despite its limited size, the cache can take advantage of instances' different popularities (some instances tend to appear more often in the replies). If the caching policy correctly identifies the "hot" instances and stores them in the cache, they can be used to reply quickly [12]. Even a small increase in the fraction of requests satisfied by the cache may turn into significant delay savings [13].

Usually, the cache is oblivious to the goal of the whole system, that is to provide a set of close/similar instances. The cache simply receives a batch of k queries for the instances in the k-*ANN* list, provides those that are stored locally, and updates its local set of instances using a classic caching policy like LRU. We refer to this behaviour as *oblivious cache*. One contribution of this paper is to provide the first analytical model to compute the hit ratio of an oblivious cache, i.e., the fraction of instances served by the cache. Our main contribution is a different way to manage the cache to *improve* the quality of the final answer provided by the system.

Our idea is to conceive the cache as a distinct k-*NN* database with its own k-*ANN* index. We refer to this operation as a k-*NN cache*. As the cache stores a small set of instances in comparison to the catalog, it can employ a more precise index, even if more demanding in terms of memory per instance. The more precise index can identify in the cache some instances closer to the query than those found by the more approximate k-*ANN* search over the whole catalog. The final answer is composed by combining the best answers from the two k-*ANN* searches. The cache can still be updated using a classic policy as LRU, but now taking into account the final answer provided.

Despite seeming a straightforward solution, the works in the literature have not properly considered its implications. For instance, we observe that the choices of the two indexes is not trivial. First, they have different requirements: the main index needs to manage a large, relatively stable, catalog, the cache index to support fast updates to track its highly dynamic set of instances. Second, both indexes should support compatible similarity measures to be able to quickly combine their search results. In designing our solution, we discuss sensible choices for the both indexes.

We evaluate the proposed k-*NN* cache with realistic datasets and request traces and show that it is able not only to improve the quality of the final answer, but also to increase the hit ratio in comparison of an oblivious cache.

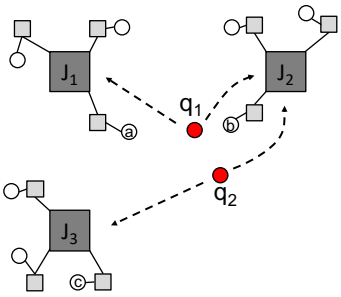

Fig. 1: PQ approach: example.

The paper is organized as follows. In Sect. II we provide background information and discuss related work. In Sect. III we define the problem we study and provide a model for the oblivious cache. Section IV presents our solution, which is evaluated in Sect. V, and Sect. VI concludes the paper.

II. Background and related work

Exact and approximate Nearest Neighbor search. In many applications, similarity is quantified using supervised machine learning techniques that collectively go under the name of distance metric learning [14]. These techniques learn how to map similar objects to vectors in a $d-$dimensional space, where a distance function, e.g., the Euclidean one, quantifies their dissimilarity. A query q_i is a point in the same space and may or may not belong to the collection. The aim of the *k-NN* is to find the k points closest to q_i.

A straightforward solution is to index the collection, e.g., with a tree based data structure, to find the exact k neighbors. Unfortunately, for values of d over 10, such an approach has a computational cost comparable to a full scan of the collection [6]. *Approximate Nearest Neighbor Search* (*k-ANN*) techniques provide k points close to q_i but not necessarily the closest, sometimes with a guaranteed bounded error. Prominent examples are the solutions based on Locality Sensitive Hashing (LSH) [15], Product Quantization (PQ) [10], [16], pivots [17], or graphs [18].

We illustrate the PQ technique for *k-ANN* search as used by Faiss library [10], since we adopt such index for the main catalog. Objects are first grouped together using a coarse *centroid-based* quantization, and then the residual error is quantized with a fine grained quantization. Figure 1 depicts a set of points (white circles) clustered into three groups corresponding to 3 different centroids (big squares labeled J_1, J_2, and J_3). The small squares identify the final quantization of the points. We assume that, for each query, the index first considers the two closest clusters, and then refines the search in the set of objects associated to those clusters. In case of $k = 2$, for query q_1, the two closest clusters are J_1 and J_2, so the search correctly returns elements a and b. For query q_2, the two closest clusters are J_2 and J_3, so the search returns elements b and c, missing that a is closer to q_2 than c.

Serving architecture and cache management. In a large, distributed setting, we can identify different entities. The catalog (set of instances) is stored in a distributed database, which is materialized in different disks. These storage resources are orchestrated by an *Application Server* (AS), which maintains the index of the collection (called the *main index*). The AS is also responsible for processing the queries coming from different clients (see Fig. 2) When a query arrives, the AS performs the *k-ANN* search, using the main index (usually maintained in RAM), and obtains the references to the k objects. The AS then retrieves the objects from the database and serves them back. In order to speed up the process, the AS may be enriched by a *cache*, i.e., a portion of the RAM (or some other fast memory) that stores the "hot" objects, so it can serve them directly, without the need to access the slower database. The answer will in general be composite with some instances retrieved by the databases, other from the cache.

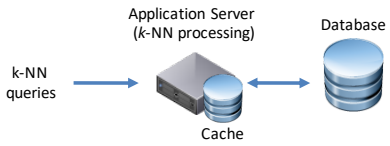

Fig. 2: Reference architecture.

Some works [3], [19] propose instead to introduce also a cache index to support *k-ANN* search on the cache content. Upon a query, if the k instances found in the cache are of acceptable quality (evaluated through some heuristics), they are used to compose the answer *without querying the main index*. The advantage of this solution is to potentially speedup the answer at the cost of serving *more approximate* answers. This idea was proposed almost at the same time in [3], [19] under the name of *similarity caching*. Reference [12] proposed a front-end similarity cache for search engines to replace them during downtime periods. Theoretical studies of similarity caching policies are in [20], [21].

Other studies proposed to integrate the *k-NN* search of the cache content and of the main catalog in the AS. For example, [22] uses a single index, embedding the information about cached objects in the index itself. The solution speeds up the index search phase, but does not improve the quality of the main index. The authors of [23] propose an early-termination approximate search that exploits the different retrieval times from the hard-disk and the RAM to compose the final answer.

Our solution relies on two indexes (one for the collection and one for the cache content) as in [3], [19], but departs from previous work as the role of the cache is not only to reduce the load on the database, but also to *improve answers' quality*.

III. Oblivious cache

In this section we consider the usual operation of a similarity search system as described above and describe how to manage the cache. When a query arrives, the AS performs a *k-ANN* search over the whole catalog, using the main index, and identifies k objects to retrieve. The most natural way to manage the cache in this setting is to let it operate obliviously to the query process, i.e., as a traditional exact cache driven by

the object request process generated by the AS. Under a Least Recently Used (LRU) policy, for example, the cache maintains an ordered queue, serves the objects stored locally, retrieves the other objects from the database and stores all requested objects at the front of the queue, evicting other objects from the rear of the queue if needed. We refer to this operation as *oblivious cache*.

Caching policies like LRU and its variants have been studied extensively (see for instance [24]), but our *oblivious cache* differs from the usual setting as requests arrive in batch for k-objects and are correlated (two close objects are more likely to appear in the same batch than two far objects). To the best of our knowledge, we present the first model that accurately predicts the hit ratio of a *oblivious cache*.

Analytical model. We propose a model that predicts the hit ratio under some assumptions on the traffic arrival pattern. We assume queries for Q possible values and follow the standard independent reference model (IRM) [25], i.e., queries for the i-th value occur according to a Poisson process with rate λ_i independently from queries for other values. If we normalize the total request rate to 1 (i.e., $\sum_{l=1}^{Q} \lambda_l = 1$), the numerical value of λ_i corresponds also to the probability that a given request is for the i-th value. A query for the i-th value is mapped by the AS to a set $\mathcal{L}_m(i)$ of k objects in a catalog of N objects. Requests for object j at the cache then follow a Poisson process with rate: $\rho_j = \sum_{i|j\in\mathcal{L}_m(i)} \lambda_i$. Note that $\sum_{l=1}^{Q} \rho_l = k$, as the cache receives k requests per time unit.

Given a query, an object j currently in the cache is moved to the front (refreshed) with probability ρ_j, independently from the past. Using Che's approximation for LRU [24], we can assume that, if not refreshed, the content stays in the cache for a deterministic time-interval of duration T_c (called the characteristic time), before being evicted due to the insertion of new objects. Che's approximation allows us to study the cache as a simpler TTL-based cache [26] and it has been mathematically justified in a number of works (see [27] and the references therein). Given a query, content j is in the cache if its previous request arrived less than T_c time earlier. This occurs with probability $o_j = 1 - e^{-\rho_j T_c}$ [24]. The characteristic time can then be computed imposing that the expected cache occupancy is equal to its size C, i.e., $C = \sum_{j=1}^{N} o_j = \sum_{j=1}^{N} 1 - e^{-\rho_j T_c}$, which can be easily solved for T_c by bisection.

The hit rate for object j is equal to its request rate (ρ_j) times the probability to find the object in the cache (o_j). The total hit rate can be computed summing the hit rates. Finally, the hit ratio (h) is equal to the total hit rate divided by the total request rate, i.e.

$$h = \frac{\sum_{j=1}^{N} \rho_j (1 - e^{-\rho_j T_c})}{k}. \tag{1}$$

Figure 3 shows that our model is very precise (see Sect. V for more information about the dataset). The figure focuses on very small cache values for which estimating correctly the hit rate is particularly difficult.

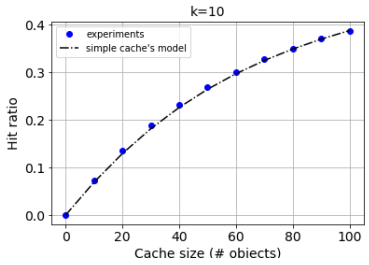

Fig. 3: Comparison of our approximate model (1) with experiments. SIFT1M dataset with equivalent tail $\alpha = 1.2$.

Limits of oblivious cache. We illustrate the limits of a *oblivious cache* going back to the example in Fig. 1. Consider now that q_1 and q_2 are two consecutive queries. After serving q_1, the cache will store both contents a and b. Upon a request for content q_2, the AS will look for b and c and retrieve b from the cache, despite the fact that the cache also stores a closer object a. While this is a limit of the approximate index used for the catalog, a more accurate search could be performed among the smaller set of objects stored at the cache, taking opportunistically advantage of closer objects the main index may have failed to identify.

IV. Indexing the Cache Content

In order to improve on a *oblivious cache*, we propose to index the objects in the cache to enable another *accurate k-ANN* search. We denote k-*ANN*$_m$ and k-*ANN*$_c$ the k-*ANN* search in the main index and in the cache index, respectively. The final answer to the query will combine the closest objects from k-*ANN*$_m$ and k-*ANN*$_c$ and will then be more accurate in general. For example, in Fig. 1 k-*ANN*$_c$ could return both elements a and b, improving on k-*ANN*$_m$. Given a query q we denote by $\mathcal{L}_m(q)$ and $\mathcal{L}_c(q)$ the results of k-*ANN*$_m$ and k-*ANN*$_c$ searches, respectively.

Illustrative experiment. We show the potential benefit of using two indexes with the following experiment. We consider consecutive queries for two close objects q_i and p_i and check how many objects from the reply to q_i can be used in the reply to p_i. Note that the objects in q_i's reply would be stored in a LRU cache at the time of p_i. We consider the dataset *SIFT1M* [28], which contains one million objects, and Faiss Polysemous [10] for k-*ANN*$_m$ over the whole dataset. We determine 1,000 pairs of close objects (q_i, p_i) by selecting uniformly at random q_i from SIFT1M and p_i as the closest point to q_i in the dataset.

Table I shows which fraction of objects in q_i's reply (and then in the cache) could be used in the final answer for the set of 1,000 queries. The label "Both" refers to cached objects that are both in $\mathcal{L}_m(q_i)$ and in $\mathcal{L}_m(p_i)$. These are the objects a *oblivious cache* would be able to serve. The label "Usable" refers to objects in $\mathcal{L}_m(q_i)$ that do not appear in $\mathcal{L}_m(p_i)$, but are closer to p_i than some of those in $\mathcal{L}_m(p_i)$. If the cache index is able to correctly identify such objects, they can be included in the final answer improving over the answer

of the main index. Numerical results reveal large potential improvements from using the cache to integrate the answers from the main index: Given m the number of objects appearing in both answers from the main index, the cache can find almost $m/2$ that are closer to the query.

In summary, using the cache as indexed storage may bring benefits in answering k-*ANN* queries. Nevertheless, our example assumes that the cache is able to perform exact k-*NN*, while in practice it would rely on another approximate index. Moreover, requests q_i and p_i may be interspaced with other requests that contribute to evict the objects in $\mathcal{L}_m(q_i)$ from the cache.

Proposed Solution. We assume the AS maintains in RAM both indexes k-*ANN*$_m$ and k-*ANN*$_c$. The portion of the RAM dedicated to the cache can be managed by any in-memory key-value store such as Memcached [29] or Redis [30]. The steps followed by the AS are described in Algorithm 1. We refer to this solution as the k-*ANN cache*.

Algorithm 1: Query processing with the k-*ANN cache* approach

input: $\mathcal{C}$, catalog stored in the database
input: p_m, parameters for the main index
input: p_c, parameters for the cache index
input: k, number of neighbors to look for
input: Q, query sequence

1 index_main ← new Index($\mathcal{C}$, p_m);
2 index_cache ← new Index([], p_c);
3 **foreach** *query* q *in* Q **do**
  `// $\mathcal{L}_m$ and $\mathcal{L}_c$ are built in parallel`
4 $\mathcal{L}_m$ ← index_main(q,k);
5 $\mathcal{L}_c$ ← index_cache(q,k);
6 $\mathcal{L}_{res}$ ← mergesort($\mathcal{L}_m$,$\mathcal{L}_c$, k);
7 $\mathcal{O}_{res}$ ← retrieve($\mathcal{L}_{res}$);
8 build_reply($\mathcal{O}_{res}$);
9 update_cache_index($\mathcal{L}_{res}$);
10 update_cache($\mathcal{O}_{res}$);

When a query arrives, the AS interrogates *in parallel* the two indexes, obtaining the list of references of the objects in the database and in the cache. It then merges the two lists, ordering the references by their distance to the query. Using this new list ($\mathcal{L}_{res}$), the AS retrieves the k nearest objects from the database and from the cache ($\mathcal{O}_{res}$), and sends them to the client. Finally, the answer is used to update the cache and its index. In what follows we consider the cache is updated according to LRU, putting the $\mathcal{O}_{res}$ objects served at the front of the cache and evicting other contents from the rear. The AS will need to keep the cache index and the cache aligned.

TABLE I: Fraction of objects common to two k-*ANN*$_m$ searches for close queries q_i and p_i (Both) and fraction of objects in $\mathcal{L}_m(q_i)$ that are not also in $\mathcal{L}_m(p_i)$, but are closer to p_i (Usable). SIFT1M dataset indexed by FAISS Polysemous.

	$k=5$	$k=10$	$k=20$
Both	0.2826	0.2916	0.3082
Usable	0.1358	0.1406	0.1351

Indexes' choice. When we consider the storage memory, we have to distinguish between *(i)* the objects themselves, *(ii)* their representations (feature vectors) in a $d-$dimensional space where we compute distances, and *(iii)* the index, including all its different data structures and the references to the objects (e.g., in case of a hash, the hash table).

For the index of the items stored in the database (k-*ANN*$_m$), we assume that the main catalog is big, and the efficiency is a key aspect. In addition, we need an index that provides not only the references, but also the distances (to be compared with the distances obtained by the k-*ANN* on the cache index). Among the different possibilities, we use an index based on PQ, such as Faiss, since it provides a fast lookup for large datasets, and an approximate distance for each reference in the answer. As future work, we will investigate alternative solutions, such as the ones based on LSH [15], which we do not consider here since they need to retrieve the feature vectors after the references have been found to compute the distances—this may cause significant additional delays as it may not be possible to keep all feature vectors in RAM [10].

As for the index of the cached objects (k-*ANN*$_c$), we need an accurate solution where we can add and remove easily the items. In addition, in order to merge and order answers from two separate indexes (line 6 in Alg. 1), the system should compute and compare easily the distances for both. Given these requirements, we adopt a graph-based approach—Hierarchical Navigable Small World (HNSW) [18]—which is fast and accurate. We modified HNSW adding the `delete` API. Upon deletion deletion of node v, we check, for each neighbor of v, if its number of neighbors fell below a certain threshold (50% of the parameter M, for each level) and, in such a case, we call the `select-neighbors-heuristic` [18] to restore the appropriate amount of neighbors.

V. Experimental results

A. On the datasets and traces

One of the main focus in the k-*ANN* search works has been the design of efficient indexes. There are many datasets, with different number of objects and dimensionality; they usually contain a few thousand representative queries to evaluate the quality of the index. Instead, we aim at evaluating the benefits of a cache in a production scenario, and we need a large *trace* (e.g., millions) of timestamped queries to evaluate the performance of caches with reasonable sizes, accounting for heterogeneous objects' popularities and temporal correlations in the request process. Despite a thorough search, we were not able to find any public repository with both a large dataset and a large number of queries. Therefore, we resorted to create two synthetic evaluation settings from *(i)* the SIF1M dataset [28] and *(ii)* an Amazon trace [31], [32].

For the SIFT1M dataset, we generated a trace according to the independent reference model (IRM) introduced in Sect. III, where objects i is requested with a constant probability λ_i,

which we refer to as *popularity*. In particular, we considered the barycenter of the whole dataset and we assigned a probability to each point i (in our experiments the set of possible queries coincides then with the catalog) according to the distance d_i to the barycenter, *i.e.*, $\lambda_i \propto d_i^{-\beta}$. The parameter β was chosen such that the slope of the tail of the popularity distribution is similar to those found empirically [2]—see Fig. 4, left. Once we built the popularity distributions, we generated a trace with one million requests. We also tested other traces generated through the same procedure, obtaining similar results.

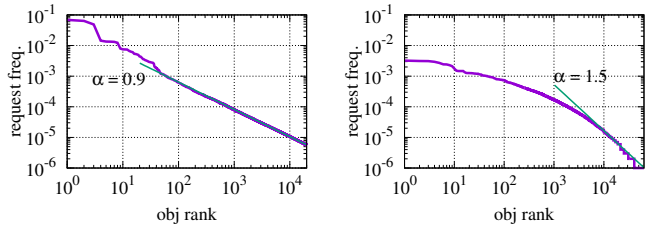

Fig. 4: Popularity distrib.: SIFT1M (left) and Amazon (right).

In order to have a more realistic time correlation in the request trace, we got about 900 thousand timestamped reviews for 63,891 Amazon products in the category "Baby" [31]. Reference [33] proposes a technique to embed the images of these products in a 100-dimensional space, where close points (according to the Euclidean distance) correspond to similar "styles." We have considered the timestamp of each review as timestamp of a corresponding query. While this trace is long enough for our purpose, a centroid-based Faiss index works best with catalogs larger than one million objects [34]. We have then artificially expanded the dataset by multiplying each product embedding with 15 different vectors with components in $\{1, -1\}$ and adding the new points to the dataset. Each of this vector induces an isometric mapping of the original dataset, as it does not change the mutual distances of the embeddings. We checked that in the final catalog, the set of k closest neighbours is not modified by this operation and that we preserve the topological properties. The popularity distribution of the requests is shown in Fig. 4, right.

B. Parameter Settings and Performance metrics

The AS manages the two indexes: k-*ANN*$_m$ and k-*ANN*$_c$. For k-*ANN*$_m$, we use Faiss Polysemous, with 256 centroids. The cache supports a hash table to be able to find which objects in the k-*ANN*$_m$ answer are stored locally, but also an additional index k-*ANN*$_c$ based on HNSW. We set the same k for all queries in a trace. We tested different values of k (5, 10 and 20). Due to space constraints, we report the results only for $k = 10$, but results for the other cases are similar.

As for the performance metrics, we consider the hit ratio, i.e., the portion of the objects that can be retrieved from the cache. In addition, we measure the recall @k, i.e., the number of objects that are in the true k nearest neighbor list.

As for the overall run-time, since both indexes are queried in parallel, each individual index run-time is not affected. The only additional operation w.r.t. a single index is the merge and sort shown in line 6 of Alg. 1, which is very fast since we are merging two ordered sets (k elements each), and it does not affect the overall delay.

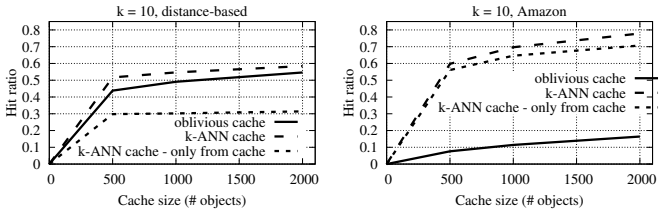

Fig. 5: Hit ratio: SIFT1M (left) and Amazon (right).

C. Results

Hit ratio. Figure 5 shows the hit ratio for different cache sizes for the SIFT1M and the Amazon dataset. We show the two approaches described in the previous section: the *oblivious cache* described in Sect. III (the cache is not indexed and simply returns the objects found by k-*ANN*$_m$), and the k-*ANN cache*, which is our proposed solution. When using the k-*ANN cache*, we keep track of the objects that we find in the cache only (curve labeled "*only from the cache*"), i.e., objects that will be part of the final answer, but have not been returned by Faiss for that specific query. This means that those objects were brought in the cache as answers to other queries, but they are useful, i.e., their distance is smaller than the references indicated by Faiss.

For the SIFT1M dataset, the k-*ANN cache* obtains consistently an increase of 3-4% in the hit ratio compared to *oblivious cache* – note that even a small increase in the hit ratio may turn into significant delay savings [13]. In addition, there is a significant fraction of results that are found only from the cache index, i.e., the cache improves the quality of the answer (see the discussion about the recall in the next paragraphs).

For the Amazon dataset, whose request trace exhibits time correlation, we have qualitatively different results: The hit ratio with a k-*ANN cache* improves by a factor at least 4 in comparison to the *oblivious cache* approach over a wide range of cache values. By analyzing the results, we noted that Faiss returns only few references that are close to the target: this leaves room for objects in the cache to improve over the references proposed by the main index.

Recall. Figure 6 shows the recall (fraction of exact k-NNs returned) for the SIFT1M dataset. With the *oblivious cache* (left) the recall is determined by the quality of the k-*ANN* search on the main index through Faiss Polysemous: 42% of the objects returned to the client are in the true k nearest neighbor list. The cache size only affects how many of those objects can be retrieved from the fast cache. On the contrary, k-*ANN cache*'s use of the two indexes is able to boost the recall up to 64% storing only 2000 objects (0.2% of the catalog).

As for the Amazon dataset (Fig. 7), the k-*ANN cache* significantly increases the recall with respect to the *oblivious*

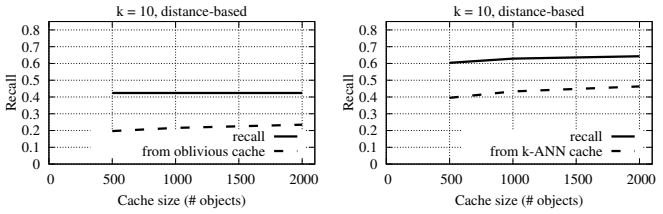

Fig. 6: Recall of the *oblivious cache* scheme (left) and with k*-ANN cache* (right). SIFT1M dataset.

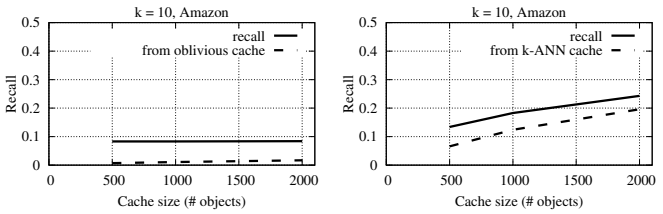

Fig. 7: Recall of the *oblivious cache* scheme (left) and with k*-ANN cache* (right). Amazon trace.

cache, even if not as much as the hit ratio improvement in Fig. 5 would suggest. This means that many objects provided by k*-ANN*$_c$, even if closer than those provided by k*-ANN*$_m$, are still quite far from the query and do not contribute to improve the recall of the system. Overall, the k*-ANN cache* appears promising: for example storing 500 objects (less than 8% of the whole catalog) leads to a 1.5x improvement in terms of accuracy and 7.5x improvement in terms of hit ratio (and then service time reduction).

VI. Conclusion

In this work we showed how, by properly indexing the cache content, we can improve the performance of k*-ANN* searches, not only by reducing the overall *service time*, but by increasing the quality of the answers. This in turn has an impact on the perceived Quality of Experience.

As next step, we plan to investigate how to derive analytical formulas as those in Sect. III also for the k*-NN* cache. The difficulty is that, while in an oblivious cache the instances requested to the cache are determined exogenously by the catalog index, in a k*-NN* cache they are also determined by the cache index and then by the set of instances in the cache.

References

[1] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," *ACM Comput. Surv.*, vol. 33, no. 3, pp. 273–321.

[2] F. Falchi, C. Lucchese, S. Orlando, R. Perego, and F. Rabitti, "Similarity caching in large-scale image retrieval," *Information processing & management*, vol. 48, no. 5, pp. 803–818, 2012.

[3] S. Pandey, A. Broder, F. Chierichetti, V. Josifovski, R. Kumar, and S. Vassilvitskii, "Nearest-neighbor caching for content-match applications," in *Proceedings of the 18th international conference on World wide web*, 2009, pp. 441–450.

[4] D. A. Adeniyi, Z. Wei, and Y. Yongquan, "Automated web usage data mining and recommendation system using k-nearest neighbor (knn) classification method," *Applied Computing and Informatics*, vol. 12, no. 1, pp. 90–108, 2016.

[5] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *Soda*, vol. 93, 1993, pp. 311–21.

[6] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *VLDB*, vol. 98, 1998, pp. 194–205.

[7] M. Aumüller, E. Bernhardsson, and A. Faithfull, "Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," *Information Systems*, vol. 87, p. 101374, 2020.

[8] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin, "Approximate nearest neighbor search on high dimensional data-experiments, analyses, and improvement," *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[9] "Benchmarking nearest neighbors," https://github.com/erikbern/ann-benchmarks, Accessed: 2021-04-16.

[10] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *IEEE Transactions on Big Data*, 2019.

[11] Q. Chen, H. Wang, M. Li, G. Ren, S. Li, J. Zhu, J. Li, C. Liu, L. Zhang, and J. Wang, *SPTAG: A library for fast approximate nearest neighbor search*, 2018. [Online]. Available: https://github.com/Microsoft/SPTAG

[12] B. B. Cambazoglu, I. S. Altingovde, R. Ozcan, and Ö. Ulusoy, "Cache-based query processing for search engines," *ACM Transactions on the Web*, vol. 6, no. 4, pp. 1–24, 2012.

[13] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Dynacache: Dynamic cloud caching," in *Proc. of the USENIX HotCloud*, 2015.

[14] A. Bellet, A. Habrard, and M. Sebban, *Metric learning*. Morgan & Claypool Publishers, 2015, vol. 9, no. 1.

[15] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *IEEE FOCS*, 2006, pp. 459–468.

[16] A. Babenko and V. Lempitsky, "The inverted multi-index," *IEEE Trans. on Pattern An. and Machine Intell.*, vol. 37, no. 6, pp. 1247–1260, 2014.

[17] B. Naidan, L. Boytsov, and E. Nyberg, "Permutation search methods are efficient, yet faster search is possible," *arXiv:1506.03163*, 2015.

[18] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Trans. on Pattern An. and Machine Intell.*, 2018.

[19] F. Falchi, C. Lucchese, S. Orlando, R. Perego, and F. Rabitti, "A metric cache for similarity search," in *Proc. of the 2008 ACM workshop on Large-Scale distrib. systems for inform. retrieval*, 2008, pp. 43–50.

[20] F. Chierichetti, R. Kumar, and S. Vassilvitskii, "Similarity caching," in *Proc. of the ACM PODS*, 2009, pp. 127–136.

[21] M. Garetto, E. Leonardi, and G. Neglia, "Similarity caching: Theory and algorithms," in *Proc. of IEEE INFOCOM*, 2020.

[22] N. R. Brisaboa, A. Cerdeira-Pena, V. Gil-Costa, M. Marin, and O. Pedreira, "Efficient similarity search by combining indexing and caching strategies," in *SOFSEM*. Springer, 2015, pp. 486–497.

[23] F. Nalepa, M. Batko, and P. Zezula, "Combining cache and priority queue to enhance evaluation of similarity search queries," in *Proc. of the ICNC-FSKD*. IEEE, 2018, pp. 956–963.

[24] H. Che, Y. Tung, and Z. Wang, "Hierarchical web caching systems: Modeling, design and experimental results," *IEEE journal on Selected Areas in Communications*, vol. 20, no. 7, pp. 1305–1314, 2002.

[25] E. G. Coffman and P. J. Denning, *Operating Systems Theory*. Prentice Hall Professional Technical Reference, 1973.

[26] N. C. Fofack, P. Nain, G. Neglia, and D. Towsley, "Performance evaluation of hierarchical TTL-based cache networks," *Computer Networks*, vol. 65, pp. 212–231, 2014.

[27] B. Jiang, P. Nain, and D. Towsley, "On the convergence of the TTL approximation for an LRU cache under independent stationary request processes," *ACM TOMPECS*, vol. 3, no. 4, pp. 1–31, 2018.

[28] "Datasets for approximate nearest neighbor search," http://corpus-texmex.irisa.fr/, Accessed: 2021-04-16.

[29] Memcached, https://memcached.org/, Accessed: 2021-04-16.

[30] Redis, https://redis.io/, Accessed: 2021-04-16.

[31] "Amazon trace," https://sim-cache.gitlabpages.inria.fr/similarity-caching-traces/, Accessed: 2021-04-16.

[32] A. Sabnis, T. S. Salem, G. Neglia, M. Garetto, E. Leonardi, and R. K. Sitaraman, "Grades: Gradient descent for similarity caching," in *IEEE INFOCOM*, 2021.

[33] J. McAuley, C. Targett, Q. Shi, and A. Van Den Hengel, "Image-based recommendations on styles and substitutes," in *Proc. of the ACM SIGIR*, 2015, pp. 43–52.

[34] "Faiss: Guidelines to choose an index," https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index, Accessed: 2021-04-16.