Exploit Every Bit: Effective Caching for High-Dimensional Nearest Neighbor Search (Extended Abstract)

Bo Tang[1] Man Lung Yiu[1] Kien A. Hua[2]
[1] The Hong Kong Polytechnic University, [2] University of Central Florida
[1]{csbtang,csmlyiu}@comp.polyu.edu.hk, [2] kienhua@cs.ucf.edu

I. Introduction

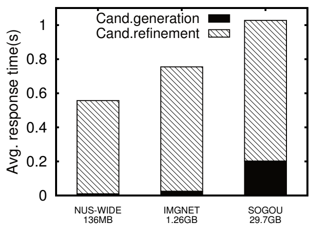
The k nearest neighbor (kNN) search takes a query point q and a point set $\mathcal{P}$ as input, and returns k points of $\mathcal{P}$ that are nearest to q. It has a wide range of applications in multimedia information retrieval, where multimedia objects (e.g., images, audio, video) are modeled as data points with dimensionality in orders of hundreds.

Due to *the curse of dimensionality* in the high dimensional space, the query efficiency of exact indexing methods degenerates to that of linear scan. Recent research focus on finding approximate results for the kNN query. Locality sensitive hashing (LSH) [3] is an attractive approach as it offers c-approximate kNN results at a sub-linear time complexity of the dataset cardinality n. The structure of LSH consists of a collection of hash tables. Each hash bucket contains a list of object identifiers (object IDs) rather than actual points. The actual data points are often stored in another file. At the query time, we process a query q in two phases:

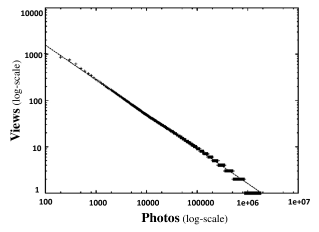
1) *candidate generation*: retrieve a candidate set of object identifiers from hash tables,
2) *candidate refinement*: for all object identifiers in the candidate set, fetch their data points from the data point file in the hard disk, then compute their distances to q and determine the k nearest results.

We mainly consider disk-based LSH [2], which is suitable for very large datasets that cannot fit into the memory. Existing LSH methods require fetching a large set of candidates (typically hundreds or thousands) from the disk and thus incur significant disk I/O costs. Therefore, the candidate refinement phase turns out to be the performance bottleneck in recent LSH methods. To verify this, we execute the state-of-the-art LSH method (C2LSH [2]) on three real high-dimensional datasets stored on the disk. Figure 1(a) depicts the average running time (wall-clock time) per query of C2LSH on these datasets. The candidate refinement phase is the performance bottleneck and it motivates us to optimize the candidate refinement time.

In this paper, we exploit a query log and devise caching techniques to reduce the candidate refinement time of high-dimensional kNN search. Caching can benefit from the temporal locality of queries as observed in typical query logs. Similarly, we expect that the query logs in multimedia retrieval systems exhibit temporal locality. For example, In Flickr, a small fraction of photos receive most of the views (see Figure 1(b)). Although there exist caching techniques [1] for kNN search on distance-based indexing methods, they are not applicable to LSH methods. LSH methods require lookup of objects by object identifiers; however, such lookup operation cannot be supported by the caches in [1].



(a) Running time (wall-clock) of C2LSH



(b) Total number of views per photo, adapted from [6]

In summary, the full version of this paper [5] develops caching methods for accelerating the candidate refinement phase of disk-based LSH. Extensive experimental results on real datasets demonstrate that our proposed caching approach can accelerate the candidate refinement time of kNN search by at least an order of magnitude.

II. Solution Overview

In our caching problem, the main research question is: *how to exploit the limited memory size and query workload to reduce the candidate refinement time*. In order to boost the cache hit ratio, we propose to cache conservative approximations of data points (i.e., representing each point in a few bits). Such conservative representation provides lower and upper distance bounds, which can be used to prune unpromising candidates and detect true kNN results early.

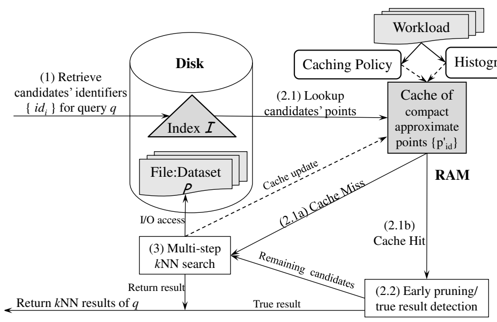

Workload
Caching Policy
Histogram
Disk
(1) Retrieve candidates' identifiers { id_i } for query q
Index I
(2.1) Lookup candidates' points
Cache of compact approximate points {p'_id}
File:Dataset P
Cache update
(2.1a) Cache Miss
RAM
I/O access
(2.1b) Cache Hit
(3) Multi-step kNN search
Remaining candidates
Return result
Return kNN results of q
True result
(2.2) Early pruning/ true result detection

Fig. 1: Our caching framework

First, we illustrate our caching framework in Figure 1. It consists of three phases: (1) candidate generation, (2) candidate reduction, (3) candidate refinement. Both Phases 1 and 3 apply existing work directly and they incur I/O. In Phase 1, we apply an existing index $\mathcal{I}$ (e.g., disk-based LSH). In Phase 3, we apply a multi-step kNN search method [4]. Phase 2 incurs no I/O and it runs our proposed technique to reduce the number of candidates before entering Phase 3.

Since the candidate refinement phase is dominated by the I/O cost, we express the candidate refinement cost as:

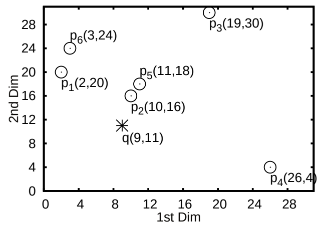
$T_{refine} \approx T_{io} \cdot C_{refine}$, where T_{io} is the disk I/O cost for fetching a data point and C_{refine} is the remaining candidate size for the refinement phase.

In general, C_{refine} is the sum of (i) the number of candidates not in the cache, or (ii) the number of candidates in the cache but they cannot be pruned:

$$\begin{aligned} C_{refine} =& (1-\rho_{hit}) \cdot |C(q)| + \rho_{hit} \cdot (1-\rho_{prune}) \cdot |C(q)| \\ =& (1-\rho_{hit} \cdot \rho_{prune}) \cdot |C(q)| \end{aligned} \tag{1}$$

where ρ_{hit} is the cache hit ratio, ρ_{prune} is the ratio of the number of pruned candidates to the number of cache hits, and $|C(q)|$ is the number of candidates. Our goal is to minimize C_{refine}. This can be done by compact approximation of points that provides tight distance bounds for pruning.

Space-efficient approximations of data points. Normally, it takes 32 bits (i.e., a floating point value) to represent each coordinate in a data point. To save space, we propose to approximate each coordinate by a short code. For example, consider the point $p_1 = (2, 20)$ in Figure 2a. We employ a histogram to map coordinates into codes. According to the example histogram in Figure 2b, the values 2 and 20 are mapped to the codes 00 and 10 respectively. Thus, we can represent p_1 by a bit-string p'_1 : " 00 10 ". Figure 2c shows the cache content for the approximate points p'_1, p'_2, p'_3, p'_4. The approximated points provide lower and upper distance bounds for candidate reduction in Phase (2).



(a) a dataset $\mathcal{P}$, with $d = 2$

code i	interval	frequency
00	[0..7]	3
01	[8..15]	2
10	[16..23]	4
11	[24..31]	3

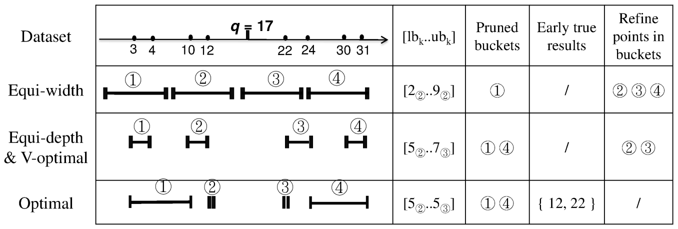
(b) a histogram $\mathcal{H}$, with code length 2

p'_1 : |00|10|
p'_2 : |01|10|
p'_3 : |10|11|
p'_4 : |11|00|

(c) a cache, with 16 bits cache size

Fig. 2: Example of histogram-based coding

Pruning effectiveness of approximated points. Given the code length, the challenge is to find the most effective way for encoding data points. We discover that we can cast this challenge as a histogram optimization problem. For simplicity, we omit the solution of the above problem and refer the interested reader to our full paper [5]. We demonstrate the pruning abilities of different histograms for kNN search in Figure 3. Interestingly, in the optimal histogram (in the last row), the buckets close to q are tight, and other buckets can be loose. It allows us to derive tighter bounds (lb_k, ub_k), prune candidates in buckets 1, 4, and detect the candidates in buckets 2, 3 as true results. Since there are no candidates in the refinement step, it incurs zero I/O cost in this example.



Dataset		[lb$_k$..ub$_k$]	Pruned buckets	Early true results	Refine points in buckets
Equi-width		[$2_{②}..9_{②}$]	①	/	② ③ ④
Equi-depth & V-optimal		[$5_{②}..7_{③}$]	① ④	/	② ③
Optimal		[$5_{②}..5_{③}$]	① ④	{ 12, 22 }	/

Fig. 3: Histograms, with $B = 4$ buckets, on 2NN search

III. Evaluation

We demonstrate the superiority of our caching solution on the real dataset (SOGOU, 29.7 GB) with a real query log. We use C2LSH as the index. Figure 4 plots the average query response time of our best caching method (HC-O), caching exact data points (EXACT) and without cache (i.e., C2LSH) for different cache size $\mathcal{CS}$. First, EXACT performs much better than C2LSH (i.e., without cache). Second, our caching method (HC-O) outperforms C2LSH by up to 5 times. Third, our best method achieves the best performance when the cache size reaches only 1/3 of the data size (i.e., 9G).

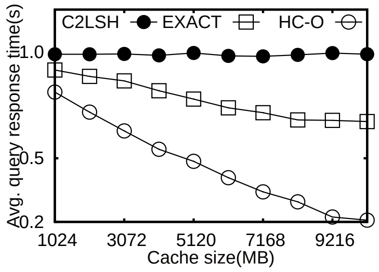


Fig. 4: Average response time (in logscale) vs. cache size

IV. Conclusion

In high-dimensional kNN search, both exact and approximate kNN solutions incur considerable time in the candidate refinement phase. In this paper, we investigate a caching solution to reduce the candidate refinement time. Our caching method HC-O is faster than EXACT caching by at least an order of magnitude, on an approximate index (C2LSH). Our work is also applicable to exact indexes (e.g., iDistance, VP-tree and VA-file). For a comprehensive coverage, see the full version of the paper [5].

Acknowledgement

This work was supported by grant GRF 152201/14E from the Hong Kong RGC.

References

[1] F. Falchi, C. Lucchese, S. Orlando, R. Perego, and F. Rabitti. Caching content-based queries for robust and efficient image retrieval. In *EDBT*, 2009.

[2] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*, 2012.

[3] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, 1998.

[4] T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD*, 1998.

[5] B. Tang, M. L. Yiu, and K. A. Hua. Exploit every bit: Effective caching for high-dimensional nearest neighbor search. *TKDE*, 28(5):1175–1188, 2016.

[6] R. van Zwol. Flickr: Who is looking? In *Web Intelligence*, 2007.