Exploit Every Bit: Effective Caching for High-Dimensional Nearest Neighbor Search

Bo Tang, Man Lung Yiu, Kien A. Hua, *Fellow, IEEE*

Abstract—High-dimensional k nearest neighbor (kNN) search has a wide range of applications in multimedia information retrieval. Existing disk-based kNN search methods incur significant I/O costs in the candidate refinement phase.
In this paper, we propose to cache compact approximate representations of data points in main memory in order to reduce the candidate refinement time during kNN search. This problem raises two challenging issues: (i) which is the most effective encoding scheme for data points to support kNN search? and (ii) what is the optimal number of bits for encoding a data point? For (i), we formulate and solve a novel histogram optimization problem that decides the most effective encoding scheme. For (ii), we develop a cost model for automatically tuning the optimal number of bits for encoding points. In addition, our approach is generic and applicable to exact / approximate kNN search methods. Extensive experimental results on real datasets demonstrate that our proposal can accelerate the candidate refinement time of kNN search by at least an order of magnitude.

Index Terms—High dimensional data, similarity search, histogram, caching

1 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 [8], where multimedia objects (e.g., images, audio, video) are modeled as data points with dimensionality in orders of hundreds [13], [29].

Due to *the curse of dimensionality* in the high dimensional space [33], the query efficiency of exact indexing methods degenerates to that of linear scan. Recent research, in both computer vision [1], [5] and database [13], [14], [29] communities, focus on finding approximate results for the kNN query. Locality sensitive hashing (LSH) [7], [16], [17] is an attractive approach as it offers c-approximate kNN results[1] at a sub-linear time complexity of $|n|$. It reduces the dimensionality of data by hashing similar data items into the same hash bucket with high probability. Unlike conventional and cryptographic hash functions, LSH aims to maximize the probability of a "collision" for similar data items. 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) *the candidate generation phase*: retrieve a candidate set of object identifiers from hash tables,

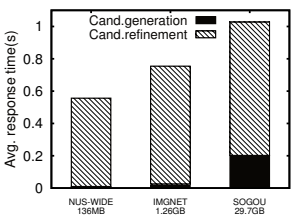


Fig. 1: Running time (wall-clock) of C2LSH

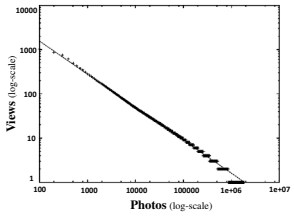


Fig. 2: Total number of views per photo [31]

2) *the candidate refinement phase*: for all object identifiers in the candidate set, fetch their data points from 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 [13], [29], 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 [13]) on three real high-dimensional datasets stored on the disk (used in the experimental study). Figure 1 depicts the average running time (wall-clock time) per query of C2LSH on these datasets. The candidate refinement stage 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 [25]. 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

- *B. Tang and M. L. Yiu are with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong.*
  E-mail: {csbtang, csmlyiu}@comp.polyu.edu.hk
- *K. A. Hua is with the College of Engineering & Computer Science, University of Central Florida.*
  E-mail: kienhua@cs.ucf.edu

1. A point p is called a c-approximate NN of q if $dist(q,p) \leq c \cdot dist(q,p^*)$, where c is the approximation ratio and p^* is the exact NN of q.

of the views (see Figure 2, adopted from [31]). Although there exist caching techniques [11], [27] for kNN search on distance-based indexing methods [6], [20], 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 [11], [27].

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.

1.1 Technical Challenges

(1) *Given the query workload (and the number of bits for encoding an approximated point), which scheme is the most effective for encoding data points?*
(2) *Given a cache size, what is the optimal number of bits for encoding a data point?*

Interestingly, we discover that we can cast challenge (1) as a histogram optimization problem. Traditional histograms have been designed for the selectivity estimation problem [19]; however, they are not effective with respect to our optimization goal. This motivates us to find the most effective histogram for our problem. Our method exploits both the data distribution and the query workload to develop compact approximations (of data points) that lead to tight distance bounds.

For challenge (2), it is non-trivial to find the optimal number of bits for encoding a data point in the cache. If each point occupies too few bits, then the cache hit ratio becomes high but those cached points lead to loose distance bounds. If each point occupies too many bits, then they provide tight distance bounds but the cache hit ratio becomes low. In this paper, we will develop a cost model to find the optimal number of bits for encoding a data point.

1.2 Novelties and Contributions

The novelties of this paper are: (i) we formulate a novel histogram optimization problem for reducing the candidate refinement cost in kNN search, (ii) our proposed solution is generic; it is applicable to not only LSH methods, but also to exact tree-based indexes.

Our technical contributions are summarized as follows.

- we formulate an appropriate histogram metric for our problem, and design an algorithm to construct an optimal histogram with respect to the novel histogram metric for challenge (1) ⟨Section 3⟩;
- we extend our solution for exact tree-based indexes (e.g., iDistance, VP-tree) ⟨Section 3.6⟩;
- we devise a cost model for estimating the performance of our solution and for automatic tuning parameter in our solution, to address challenge (2) ⟨Section 4⟩;
- we demonstrate the superiority of our caching solution on three real datasets, in particular, the largest dataset (SOGOU, 29.7 GB) has a real query log ⟨Section 5⟩.

The remainder of this paper is organized as follows. We formulate our caching problem in Section 2 and present our histogram-based caching method in Section 3, then provide a cost model and the optimal parameter setting in Section 4. We conduct our experimental study in Section 5. We discuss related work in Section 6. Finally, we conclude this paper in Section 7.

2 Definition and Problem Statement

2.1 Definitions

We represent points and the distance both in the point form and the vector form interchangeably.

Definition 1 (Point). *A point p is defined as a d-dimensional tuple $p = (p.1, p.2, ..., p.d)$. Its vector form is defined as: $\overrightarrow{p} = [p.1, p.2, ..., p.d]$.*

Definition 2 (Distance metric). *The Euclidean distance $dist_q(c)$ of a data point c from a query point q is defined as: $dist_q(c) = \sqrt{\sum_{j=1}^{d}(q.j - c.j)^2} = ||\overrightarrow{q} - \overrightarrow{c}||$*

Then we define the kNN search problem as:

Definition 3 (kNN search problem). *Given a query point q and a point set $\mathcal{P}$, the kNN search returns a subset $R \subset P$ of k points such that $dist_q(p) \leq dist_q(p')$ for any $p' \in P - R$.*

In this paper, we just return the identifiers of points in R but not the actual points. This is reasonable for multimedia information retrieval applications.

As discussed in the introduction, we focus on accelerating *disk-based LSH methods* (e.g., C2LSH [13]) without affecting their query results. These methods access (i) a hash-based index $\mathcal{I}$, whose hash buckets store point identifiers, and (ii) a sequential file for the point set $\mathcal{P}$, which supports direct access of data point by identifier[2].

During query processing, we first retrieve a candidate set $C(q)$ from the index $\mathcal{I}$ as follows:

Definition 4 (Candidate set $C(q)$). *Given a query point q, the index $\mathcal{I}$ reports a set of identifiers for candidate points $C(q) = \{id_i\}$.*

Then, we fetch the corresponding data points by identifiers from the file of $\mathcal{P}$.

2.2 Research objective

For typical disk-based LSH methods, the candidate refinement time T_{refine} is usually much longer than candidate generation time T_{gen} (see Figure 1). In this paper, we aim to reduce T_{refine} significantly by caching points in RAM. To boost the cache hit ratio, we propose to cache *compact approximate points* (p'_{id}), which will be elaborated in Section 3.

2. An alternative is to store $\mathcal{P}$ based on the distribution of clusters in data [20]. We will test its effect in experiments.

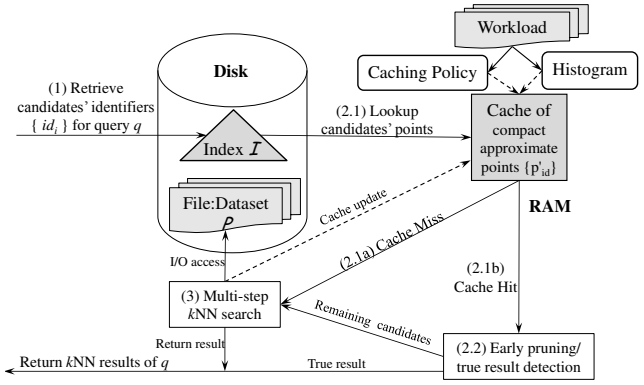


Fig. 3: Framework of caching on a high-dimensional dataset

Figure 3 illustrates the framework of our caching problem for kNN search on a high dimensional dataset. Our kNN search procedure (cf. Section 3) 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 $\mathcal{I}$. In Phase 3, we apply a multi-step kNN search method [22], [26], which will be elaborated in Section 2.3. 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 time T_{refine} is dominated by the I/O cost, we express it 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:

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

where ρ_{hit} is the cache hit ratio, and ρ_{prune} is the ratio of the number of pruned candidates to the number of cache hits.

The scope of our problem: Our goal is to minimize the value of C_{refine} before candidate refinement (and thus minimize T_{refine}). We can reduce C_{refine} by two orthogonal aspects: (i) a caching policy that offers high hit ratio, and (ii) compact approximation of points that provides tight distance bounds for pruning.

The study of (i) caching policies is orthogonal to our problem. In fact, our proposed solution can be applied with existing web caching policies [25], e.g., Least-Recently-Used (LRU) and Highest-frequency-first (HFF). LRU is a dynamic caching policy, whereas HFF is a static caching policy that requires a query workload $\mathcal{WL}$[3] to decide the initial cache content. We will elaborate HFF in Section 4.

Our focus is issue (ii), which will be discussed in Section 3. Formally, we define our problem as:

3. It is usually the historical query log.

Definition 5 (**Caching problem**). *Given a cache size $\mathcal{CS}$, a workload of queries $\mathcal{WL}$, and a point set P, determine the cache content such that it minimizes $\sum_{q \in \mathcal{WL}} C_{refine}(q)$.*

Besides LSH methods, we will discuss how to adapt our proposed solution for tree-based indexes (e.g., R-tree, X-tree, SR-tree) [3] in Section 3.6.

Note that our solution offers speedup without affecting the quality of query results. If an exact tree-based index is used, the query results remain exact. If an LSH method is used, the quality of its query results is preserved.

2.3 Multi-step kNN Search

We illustrate multi-step kNN search methods [22], [26] in this section. Kriegel et al. [22] present an efficient method that requires both lower and upper distance bounds functions. For example, suppose the candidate set of q is $C(q) = \{p_1, p_2, p_3, p_4\}$. Figure 4 depicts the lower and upper distance bounds of these candidates as intervals. Their exact distances (from q) are shown as gray dots; However, they can be obtained only after fetching the exact points from the disk. First, it calculates the k-th smallest lower distance bound lb_k and the k-th smallest upper distance bound ub_k, among the candidates, the values of lb_2 and ub_2 are shown in Figure 4. Since the upper distance bound of p_1 is less than lb_2, lb_3 and lb_4, p_1 must be a result so we need not fetch p_1 from the disk. The lower distance of p_4 is larger than ub_2, p_4 cannot be a result so we also need not fetch p_4. It suffices to fetch p_2 and p_3 from the disk.

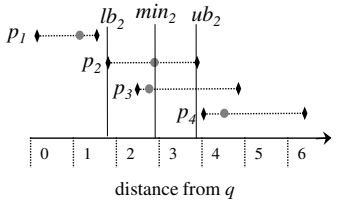


Fig. 4: Multi-step kNN methods, $k = 2$

3 Histogram-based Caching for kNN Search

Histograms, generally, are designed to provide selectivity estimates on a single column attribute of a relational table. In this section, we utilize a histogram to define compact approximate representations of points. Then, we present a kNN search algorithm with our proposed histogram-based cache. Finally, we formulate the novel metric to build an effective histogram for kNN search.

3.1 Histogram and Approximate Points

We first define a histogram as follows:

Definition 6 (Histogram). *A histogram $\mathcal{H}$ is defined as an array of buckets which cover a domain interval (e.g.,*

[0,..,N_{dom}])[4]*. Let B be the number of buckets in $\mathcal{H}$. Each bucket (say, the i-th bucket) stores: (i) an interval $[l_i..u_i]$ of values, and (ii) the total frequency $freq_i$ of values in the interval.*

Figure 5b shows an example histogram. In our problem, we only care about the bucket position i and its interval $[l_i..u_i]$, but not its frequency $freq_i$. Each bucket position can be represented by a binary code. In general, for a histogram with B buckets, the code length is: $\tau = \lceil \log_2(B) \rceil$. We will examine histogram construction methods in Section 3.3 and discuss the tuning of τ in Section 4.

With a histogram $\mathcal{H}$, we can convert an exact point p into an approximate point p' as follows:

Definition 7 (Bucket lookup). *Given a value v, we define the function $\mathcal{H}(v) = i$, such that v is covered by the interval of the bucket i, i.e., $l_i \leq v \leq u_i$.*

Definition 8 (Approximate point p'). *Given a d-dimensional point $p = (p.1, p.2, \cdots, p.d)$, we define an approximate point p' with respect to the global histogram $\mathcal{H}$ as:*

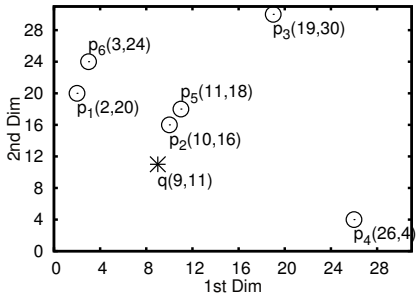
$$p' = (\ \mathcal{H}(p.1), \mathcal{H}(p.2), \cdots, \mathcal{H}(p.d)\ ) .$$

Specifically, we approximate each dimension value by a τ-bit code. In general, if the dataset has different domain sizes for different dimensions, then we may apply normalization to scale each dimension.

For example, consider the point $p_1 = (2, 20)$ in Figure 5a. According to the example histogram in Figure 5b, 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 5c shows the cache content which stores the approximate points p'_1, p'_2, p'_3, p'_4.

To achieve a compact cache, we pack the bit-string encoding of each point into one or multiple consecutive words in memory.[5]

In subsequent discussion, we use a global histogram $\mathcal{H}$ to define all dimensions of an approximate point p'. We will discuss alternative histograms in Section 3.6.



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

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

(b) a histogram $\mathcal{H}$, with $\tau = 2$

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

(c) a cache, with $\mathcal{CS} = 16$ bits

Fig. 5: Example of histogram-based coding

3.2 kNN Search Algorithm

Algorithm 1 elaborates the framework of kNN search with histogram-based caching. The corresponding steps in Figure 3 are also labeled in this algorithm.

First, we retrieve a set $C(q)$ of candidates from the index $\mathcal{I}$ (Line 2), and then check whether they are in the cache. For each candidate c_i found in the cache (Lines 5–6), we compute its lower/upper distance bounds to q as follows.

$$dist_q^+(p') = \sqrt{\sum_{j=1}^{d} \max\{|q.j - p^l.j|, |q.j - p^u.j|\}^2}$$

$$dist_q^-(p') = \sqrt{\sum_{j=1}^{d} \begin{cases} 0 & \text{if } p^l.j \leq q.j \leq p^u.j \\ \min\{|q.j - p^l.j|, |q.j - p^u.j|\}^2 & \text{otherwise} \end{cases}}$$

where $p^l.j = l_{\mathcal{H}(p'.j)}$ and $p^u.j = u_{\mathcal{H}(p'.j)}$. For those candidates missing in the cache, we will fetch their points from the disk (in the final phase).[6]

The next phase (Lines 7–13) focuses on reducing the candidate size (which do not incur disk accesses). Among all candidates $C(q)$, we derive the k-th minimum lower bound distance lb_k, the k-th minimum upper bound distance ub_k (Lines 7–8). First, we prune candidates having $c_i.lb$ larger than ub_k (Lines 10–11), as they cannot be among k nearest neighbors. Second, we identify candidates having $c_i.ub$ less than lb_k (Lines 12–13), as they must be results and moved to the result set $\mathcal{R}$. Obviously, the effectiveness of this phase depends on the tightness of distance bounds (and the histogram $\mathcal{H}$). We will explore this issue in the remaining subsections.

Finally, in the refinement phase, we apply a multi-step kNN search method [22], [26] (which incurs disk I/O), as described in Section 2.3, with the remaining candidate set $C(q)$. This search would fetch data points from $\mathcal{P}$ when necessary. The update of the cache Ψ is optional; it is only required when a dynamic caching policy (e.g., LRU) is used.

Example: Assume $k = 1$ and consider the query q and dataset $\mathcal{P}$ in Figure 5a. We show the running steps of kNN search (Algorithm 1) in Table 1. Suppose that the index $\mathcal{I}$ reports the candidate set for q as: $C(q) =$

4. N_{dom} is the largest dimension value of all points in all dataset

5. With this implementation, an approximate point occupies $\frac{d \cdot \tau}{L_{word}}$ words, where L_{word} is the memory word size (in bits). The value of L_{word} (typically 32, 64) is fixed by the CPU. During kNN search, we can extract these cache items by performing bitwise operations on these words.

6. An optimization is to fetch those points from disk immediately, in order to tighten the bounds lb_k and ub_k to be mentioned soon. However, this optimization is not effective when the hit ratio is low (as few candidates can be pruned) or high (as lb_k and ub_k are tight already).

Algorithm 1 kNN Search (Query q, Result size k)

```
   Disk data: Index I, Dataset file P
   Memory data: Cache Ψ, histogram H
   • Phase 1: candidate generation
1: Result set R := ∅
2: retrieve the candidate set C(q) from I
   • Phase 2: candidate reduction
3: for each c_i ∈ C(q) do                          ▷ part 2.1: cache lookup
4:     c_i.lb := 0; c_i.ub := +∞
5:     if Ψ contains p'_{c_i} then                  ▷ cache hit
6:         c_i.lb := dist_q^-(p'_{c_i}); c_i.ub := dist_q^+(p'_{c_i})
7: lb_k := the k-th minimum of {c_i.lb : c_i ∈ C(q)}
8: ub_k := the k-th minimum of {c_i.ub : c_i ∈ C(q)}
9: for each c_i ∈ C(q) do                          ▷ part 2.2
10:    if c_i.lb > ub_k then                       ▷ early pruning
11:        remove c_i from C(q)
12:    else if c_i.ub < lb_k then                  ▷ true result detection
13:        move c_i from C(q) to R
   • Phase 3: candidate refinement
14: if |R| < k then
15:    Multi-step-kNN( C(q), P, R )                ▷ Ref. [22], [26]
16: return R
```

$\{p_1, p_2, p_3, p_4, p_5, p_6\}$. Since the candidates p_5 and p_6 are missing in the cache, we must access their points from the disk. The cache is used to retrieve approximate points (p'_1, p'_2, p'_3, p'_4) of four candidates, as shown in Figure 5c. Then, we compute their lower/upper distance bounds from q, and obtain the distance threshold $ub_k = dist_q^+(p_2) = \sqrt{\max\{(9-8),(9-15)\}^2 + \max\{(11-16),(11-23)\}^2} = \sqrt{6^2+12^2} = 13.42$. We can prune p_3 and p_4 as their lower bound distances are above 13.42. Finally, we apply multi-step kNN search [22], [26] on the remaining candidates p_1, p_2, which costs at most 2 disk accesses. In summary, this example incurs at most 4 disk accesses: 2 for p_5, p_6, and at most 2 for p_1, p_2.

TABLE 1: kNN search on the cache, $k = 1$

cache (code array)		rectangle	$[lb_i..ub_i]$	pruned ?
$p_1 : \|00\|10\|$		([0..7],[16..23])	[5.39..15]	
$p_2 : \|01\|10\|$	$\Rightarrow$	([8..15],[16..23])	[5.00..**13.42**]	
$p_3 : \|10\|11\|$		([16..23],[24..31])	[14.76..24.41]	yes
$p_4 : \|11\|00\|$		([24..31],[0..7])	[15.52..24.60]	yes

3.3 Histogram Solutions for kNN Algorithm

In relational databases, histograms are used to summarize the data distribution and provide result size estimations for selection queries [19]. However, they have not been used for supporting kNN search. This raises interesting questions:

3.3.1 Are existing histograms effective for kNN search?

(1) Heuristic histograms (e.g., equi-width, equi-depth) [18]. In order to describe them, we use the notations in Definition 6, and denote $F[x]$ as the frequency of value x (in a table column). For equi-width histogram, all buckets have the same width $(u_i - l_i)$; whereas for equi-depth histogram, all buckets have approximately the same sum of frequencies $(\sum_{x=l_i}^{u_i} F[x])$.

(2) V-optimal histogram [19], which minimizes the average estimation error of selection queries according to the *sum squared error* (SSE) metric [19]:

$$\mathcal{M}_{SSE}(\mathcal{H}) = \sum_{i=1}^{B} \sum_{x=l_i}^{u_i} (F[x] - AVG([l_i, u_i]))^2$$

where $AVG([l_i, u_i]) = \frac{\sum_{x=l_i}^{u_i} F[x]}{u_i - l_i + 1}$ is the average frequency of values in bucket i.

We illustrate the effectiveness of these histograms on kNN search by an example in Figure 6. We will evaluate its effectiveness on real datasets in the experimental section. For ease of illustration, we consider a 1-dimensional dataset $\{3, 4, 10, 12, 22, 24, 30, 31\}$ and each value x in it has frequency $F[x] = 1$. Suppose that $q = 17$ is the only query in the query workload $\mathcal{WL}$. Assume that the cache can hold all approximate points and the code length is $\tau = 2$. Thus, each histogram has $B = 2^\tau = 4$ buckets. Figure 6 depicts the buckets (intervals) of equi-width, equi-depth, and V-optimal histograms. Both equi-depth and V-optimal histograms are the same in this example.

For each histogram, we run the kNN algorithm (in Section 3.2) to find 2NN ($k = 2$) at $q = 17$, and show the running steps in the figure. Histogram buckets are shown in the first column. By using distance bounds (lb_k, ub_k) in the second column, we can prune unpromising candidates and detect true result early in order to reduce the remaining candidate size. In this example, the ideal histogram for our problem has zero remaining candidates (see the last column). On the other hand, equi-width has 6 remaining candidates (in buckets ②,③,④), and equi-depth (and V-optimal) has 4 remaining candidates (in buckets ②,③). These histograms would incur higher cost than the ideal histogram in the candidate refinement phase.

Differences from traditional histograms: Traditional histograms are designed for the selectivity estimation problem. For example, a traditional histogram metric like $\mathcal{M}_{SSE}(\mathcal{H})$ [19]internal minimize the average estimation error over all histogram buckets. It is fine to have a bucket with large width $(u_i - l_i)$, as long as the values within the bucket have similar frequencies. However, for the kNN problem, such a bucket causes loose distance bounds and thus not effective in shrinking the candidate set. Furthermore, $\mathcal{M}_{SSE}(\mathcal{H})$ does not exploit query workload information, which is important for deriving an effective histogram for our problem.

In Section 3.4.1, we will formulate a new histogram metric for building an effective histogram for kNN search. Experiments show that our novel histogram incurs lower I/O cost than existing histograms (e.g., equi-width, equi-depth, V-optimal) on kNN search by at least 50%.

3.3.2 What is the optimal histogram for kNN search?

We demonstrate an example of the optimal histogram for kNN search in the last row in Figure 6. Interestingly, in this histogram, 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.

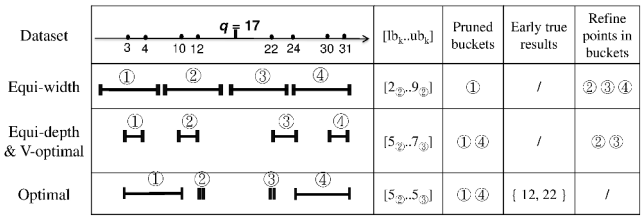


Fig. 6: Effectiveness of histograms, with $B = 4$ buckets, on 2NN search, $\mathcal{WL}$ = { q }

3.4 Effective Histogram Metric

3.4.1 Histogram Metric for kNN Search

This section formulates a histogram metric that captures the effectiveness of a histogram for kNN search. In the following discussion, we associate notations with the superscript q if they depend on q (e.g., $C^q_{refine}, lb^q_k, ub^q_k$).

Our goal is to minimize the remaining candidate size C^q_{refine} (cf. Eqn. 1) before the candidate refinement phase in Algorithm 1. Note that C^q_{refine} is influenced by the distance bounds lb^q_k, ub^q_k (obtained at Lines 7–8), which are derived from the content of $\mathcal{H}$. For each candidate c that hits in the cache Ψ, we can skip it in candidate refinement in two cases:

- (i) if $dist^+_q(c) \le lb^q_k$, then it must be a true result
- (ii) if $dist^-_q(c) \ge ub^q_k$, then it cannot be a result

Otherwise, the candidate requires refinement.

With the above observation, we proceed to define our histogram construction problem as follows.

Definition 9 (Optimal kNN histogram problem)**.** *Let $V = \{v_1, \cdots, v_n\}$ be the set of distinct dimensional values of data points in $\mathcal{P}$.*

Given the code length τ, the query workload $\mathcal{WL}$, and the cache Ψ (i.e., a set of points), the problem is to build a histogram $\mathcal{H}$ with $\mathcal{B} = 2^\tau$ buckets such that (i) it covers all values in V, and (ii) it minimizes the following metric:

$$\mathcal{M}^{\mathcal{WL}}_{kNN}(\mathcal{H}) = \sum_{q\in\mathcal{WL}} \sum_{c\in C(q)\wedge\Psi} refine_{\mathcal{H}}(c) \quad \textit{(M1)}$$

$$\textit{where} \quad refine_{\mathcal{H}}(c) = \begin{cases} 0 & \textit{if } dist^-_q(c) \ge ub^q_k \\ 0 & \textit{if } dist^+_q(c) \le lb^q_k \\ 1 & \textit{otherwise} \end{cases}$$

This optimal histogram construction problem is challenging as the number of combinations of buckets lead to a huge search space $O\binom{n}{\mathcal{B}-1}$, which is $O(n^{\mathcal{B}})$ when $\mathcal{B} \ll n$. Thus, we propose an approximate solution below.

3.4.2 Approximate Histogram Metric

Besides the huge search space $O(n^{\mathcal{B}})$, the above metric does not use histogram bucket intervals explicitly, thus rendering it inconvenient to develop a solution.

To tackle these issues, we propose to approximate the metric (M1), $\mathcal{M}^{\mathcal{WL}}_{kNN}(\mathcal{H})$, in a form that can be efficiently solved. In order to minimize $\mathcal{M}^{\mathcal{WL}}_{kNN}(\mathcal{H})$, we should maximize the number of candidates that can satisfy $refine_{\mathcal{H}(c)} = 0$ for a given query q. Note that there are two cases for $refine_{\mathcal{H}}(c) = 0$: **(Case i)** the lower bound of c is larger than the k-th upper bound (i.e., $dist^-_q(c) \ge ub^q_k$), so c is not a result point and, **(Case ii)** the upper bound of c is smaller than the k-th lower bound (i.e., $dist^+_q(c) \le lb^q_k$), so c must be a result point.

Observe that at most $|C(q)| - k$ candidates can satisfy $dist^-_q(c) \ge ub^q_k$ in **(Case i)**, but at most k candidates can satisfy $dist^+_q(c) \le lb^q_k$ in **(Case ii)**. Since $|C(q)| - k \gg k$, we focus on **(Case i)** and plan to minimize ub^q_k.

Recall that $ub^q_k = k^{th}\min_{c\in C(q)\wedge\Psi} dist^+_q(c)$ is contributed by k points in $C(q)\wedge\Psi$. By denoting these k points as $b^q_1, b^q_2, \cdots, b^q_k$, we have $ub^q_k = \max_{1\le r\le k} dist^+_q(b^q_r)$. We then define the error vector of a candidate in Def. 10.

Definition 10 (**Error vector**)**.** *Given a histogram $\mathcal{H}$ and a candidate c, we define the error vector of c as $\overrightarrow{\epsilon(c)} = [\epsilon(c).1, \epsilon(c).2, ..., \epsilon(c).d]$ where $\epsilon(c).j = u_{\mathcal{H}(c.j)} - l_{\mathcal{H}(c.j)}$.*

By using Lemma 1 (stated below), we derive: $ub^q_k \le \max_{1\le r\le k} ||\epsilon(b^q_r)|| + dist_q(b^q_r)$. Note that each term $dist_q(b^q_r)$ is a constant value (depending on q) that cannot be optimized. As a heuristic, we approximate the minimization of ub^q_k by minimizing the following Metric (M2):

$$\mathcal{M}2^{\mathcal{WL}}_{kNN}(\mathcal{H}) = \sum_{q\in\mathcal{WL}} \sum_{r=1}^{k} ||\overrightarrow{\epsilon(b^q_r)}||^2 \quad \text{(M2)}$$

Next, for convenience, we define a multi-set $\mathcal{QR}$ to store all b^q_r for queries in the workload $\mathcal{WL}$:

$$\mathcal{QR} = \{b^q_r \ : \ q \in \mathcal{WL}, dist(q, b^q_r) \le ub^q_k, r \in [1, k]\} \quad (2)$$

Then, we define $F'[x]$ as the frequency of x in coordinates of candidates in $\mathcal{QR}$, where $x \in [0, N_{dom}]$:

$$F'[x] = \texttt{COUNT}\ \{b^q_r.j = x \ : \ b^q_r \in \mathcal{QR}, j \in [1, d]\} \quad (3)$$

By Lemma 2 (stated below), we express Metric (M2) into the following form using histogram bucket information.

$$\mathcal{M}2^{\mathcal{WL}}_{kNN}(\mathcal{H}) = \sum_{i=1}^{B} \sum_{x=l_i}^{u_i} F'[x] \cdot (u_i - l_i)^2 \quad \text{(M3)}$$

Lemma 1 (**Distance inequality**)**.** *Any candidate c satisfies*

$$dist^+_q(c) - dist_q(c) \le ||\overrightarrow{\epsilon(c)}||$$

Proof: in Appendix A. □

Lemma 2 (**Metric transformation**)**.**

$$\sum_{q\in\mathcal{WL}} \sum_{r=1}^{k} ||\overrightarrow{\epsilon(b^q_r)}||^2 = \sum_{i=1}^{B} \sum_{x=l_i}^{u_i} F'[x] \cdot (u_i - l_i)^2$$

where $u_i - l_i$ denotes the width of bucket i.

Proof: in Appendix A. □

3.5 Efficient Solution

We proceed to present an efficient solution for the simplified histogram metric $\mathcal{M}2^{\mathcal{WL}}_{kNN}(\mathcal{H})$(**M3**). First, we represent the inner sum in Eqn. 4 as follows:

$$\Upsilon([l_i, u_i]) = \sum_{x=l_i}^{u_i} F'[x] \cdot (u_i - l_i)^2 \quad (4)$$

Then, we propose an efficient algorithm to construct histogram $\mathcal{H}$ that minimizes the metric $\mathcal{M}2^{\mathcal{WL}}_{kNN}(\mathcal{H})$.

We assume that the value domain is: $1..N_{dom}$.[7] Let $OPT(n, m)$ be the minimum $\mathcal{M}2^{\mathcal{WL}}_{kNN}(\mathcal{H})$ value for the histogram covering the interval $[1..n]$ with at most m buckets. If t is the optimal splitting position for the last bucket, then we have: $OPT(n, m) = OPT(t, m-1)+\Upsilon([t+1, n])$, where $\Upsilon([t + 1, n])$ is the metric value contributed by the last bucket ($[t+1, n]$), and $OPT(t, m-1)$ is the minimum metric value for the histogram covering $[1..t]$ with at most $m - 1$ buckets. By considering all splitting positions, we take $OPT(n, m)$ as the minimum sum as follows:

$$OPT(n, m) = \min_{1 \le t < n} \{OPT(t, m - 1) + \Upsilon([t + 1, n])\} \quad (5)$$

With Eqn. 5, we apply the dynamic programming approach to calculate $OPT(n, m)$ and the split positions, for all $1 \le n \le N_{dom}$ and $1 \le m \le B$. Finally, we obtain the optimal histogram $\mathcal{H}$ through these split positions.

Lemma 3 (**Monotonicity of** Υ)**.** *if* $l_1 \le l_2$, *Then* $\Upsilon([l_1, u_i]) \ge \Upsilon([l_2, u_i])$.

Proof: According to Eqn. 3, then $F'[x] \ge 0$. Since $l_1 \le l_2$, then, $\sum_{x=l_1}^{u_i} F'[x] \ge \sum_{x=l_2}^{u_i} F'[x]$ and $(u_i - l_1)^2 \ge (u_i - l_2)^2$. Consider $\Upsilon([l_i, u_i]) = \sum_{x=l_i}^{u_i} F'[x] \cdot (u_i - l_i)^2$ then we can conclude $\Upsilon([l_1, u_i]) \ge \Upsilon([l_2, u_i])$. □

Through Lemma 3, our algorithm can terminate when $\Upsilon([t+1, n]) \ge OPT(n, m)$. This technique can significantly reduce the running time when n is very large. The details are presented in Algorithm 2.

Time Complexity: This algorithm is only executed once in the offline phase. It has a total of $O(N_{dom} \cdot B)$ calculations for $OPT(n, b)$. In the worst case, each calculation involves $O(N_{dom})$ values of t (Eqn. 5). Thus, its time complexity is $O(N^2_{dom} \cdot B)$. It is independent of the dimensionality d and the data size $|\mathcal{P}|$.

Histogram maintenance: We expect that the distribution of queries in the workload does not change rapidly. Following the practice in search engines [25], we propose to perform updates and rebuild the cache periodically (e.g., daily).

3.6 Extensions

3.6.1 Adaptation for tree-based indexes

The kNN search on tree-based indexes [4], [6], [20] exhibits interleaving steps between candidate generation and candidate refinement. In this section, we discuss how to adapt our proposed solution to speedup kNN search on tree-based indexes.

7. We can extend this method to handle other value domain, e.g., by applying discretization on floating-point values.

Algorithm 2 Build-kNN-Histogram (Bucket number B, Value domain size N_{dom}, Frequency array F')

```
1: let H be an empty histogram
2: OPT :=new 2D array(1..N_dom, 1..B)            ▷ for OPT values
3: pos :=new 2D array (1..N_dom, 1..B)           ▷ for split positions
4: for m from 1 to B do
5:     for n from 1 to N_dom do
6:         if m = 1 then
7:             OPT(n, 1) := Υ([1, n])
8:         else
9:             OPT(n, m) = +∞
10:            for each t from n − 1 to 1 do
11:                if OPT(n, m) > OPT(t, m-1) + Υ([t+1, n]) then
12:                    OPT(n, m) := OPT(t, m − 1) + Υ([t + 1, n])
13:                    pos(n, m) := t
14:                else if Υ([t + 1, n]) ≥ OPT(n, m) then
15:                    break                     ▷ by Lemma 3
16: n := N_dom
17: for m from B to 1 do
18:     if m = 1 then
19:         l := 1, u := n
20:     else
21:         l := pos(n, m) + 1, u := n
22:         n := pos(n, m)
23:     insert the bucket [l..u] to H
24: Return H
```

We illustrate a general tree structure in Figure 7. Each node occupies a disk block. A leaf node stores data points, whereas a non-leaf node stores branching information for its children. Conceptually, we can divide the tree into two parts: (i) the set of non-leaf nodes as the index $\mathcal{I}$, and (ii) the set of leaf nodes as the dataset $\mathcal{P}$. The storage size of $\mathcal{P}$ is generally much larger than that of $\mathcal{I}$. We store the exact $\mathcal{I}$ in memory.

In this scenario, we consider each cache item to be a leaf node (i.e., approximate representations of all points in that node), but not an individual point. We construct the cache as follows. First, we run queries in the query workload $\mathcal{WL}$ and collect the access frequency of each leaf node. Then, we fill the cache with leaf nodes in descending order of access frequency. Finally, with our technique in Section 3.5, we can build the histogram $\mathcal{H}$ and determine the approximate representations of data points (in leaf nodes).

Any tree-based kNN search solution $\mathcal{A}$ (e.g., [4], [20]) can utilize the above cache, with some slight modifications described below. During kNN search, before we fetch a leaf node (by its Block ID), we first lookup it in the cache. If the leaf node is not in the cache, then we load it from the disk. Otherwise, we retrieve the node from the cache, examine its approximate points, and compute lower and upper bounds for that node. In this implementation, our solution provides tight lower and upper bounds for leaf nodes.

In addition, we can further optimize the algorithm as follows. We first compute the lower and upper bound for each point in that node. These bounds can be used to tighten ub_k and prune some unpromising nodes and approximate points. Then, the multi-step kNN search method determines which node should be examined next. As we will illustrate in experiments (cf. Figures 16(a,c) in Section 5.4.5), the

above approximate caching solution performs better than exact caching.

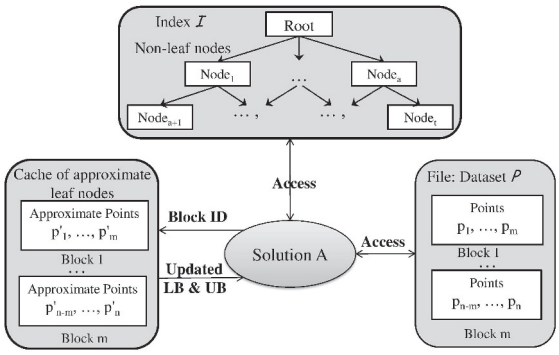


Fig. 7: Tree-based kNN search with our cache

3.6.2 Alternative histogram categories

Besides the global histogram, we may use other histograms to convert a point p into an approximate point p'.

Individual histogram: This approach employs a separate histogram $\mathcal{H}_i$ for each dimension $i = 1..d$. It converts an exact point p to an approximate point p' as follows: $p' = (\ \mathcal{H}_1(p.1), \mathcal{H}_2(p.2), \cdots, \mathcal{H}_d(p.d)\ )$.

We proceed to discuss how to build these d histograms. Observe that our histogram metric M3 is defined by using a frequency array: $F'[x] = \texttt{COUNT}\ \{b_r^q.j = x\ :\ b_r^q \in \mathcal{QR}, j \in [1,d]\}$. We can decompose this array into individual frequency arrays of the form: $F'_j[x] = \texttt{COUNT}\ \{b_r^q.j = x\ :\ b_r^q \in \mathcal{QR}\}$. Then, we can express metric M3 as follows:

$$\sum_{i=1}^{B}\sum_{x=l_i}^{u_i} F'[x]\cdot(u_i-l_i)^2 = \sum_{i=1}^{B}\sum_{x=l_i}^{u_i}(\sum_{j=1}^{d} F'_j[x])\cdot(u_i-l_i)^2$$
$$= \sum_{j=1}^{d}(\sum_{i=1}^{B}\sum_{x=l_i}^{u_i} F'_j[x]\cdot(u_i-l_i)^2)$$

Finally, for each dimension j, we find a histogram $\mathcal{H}_j$ that minimizes $\sum_{i=1}^{B}\sum_{x=l_i}^{u_i} F'_j[x]\cdot(u_i-l_i)^2$ by applying Algorithm 2.

Multi-dimensional histogram: A multi-dimensional histogram $\mathcal{H}_{MD}$ partitions the space into buckets (i.e., bounding rectangles). Given an exact point p, we compute its approximate point as $p' = \mathcal{H}_{MD}(p)$, which denotes the identifier of the bucket enclosing p.

Due to the curse of dimensionality, a multi-dimensional histogram is not effective for approximation (cf. Appendix B). As such, we do not bother to extend our solution for multi-dimensional histogram. Instead, we use an R-tree based multi-dimensional histogram and test its effectiveness in the experimental study.

4 Cost Estimation Model

Section 4.1 estimates the I/O cost of our proposed solution, as a function of the cache size $\mathcal{CS}$ and the code length τ. In Section 4.2, we derive the optimal code length τ (for a given $\mathcal{CS}$) such that it leads to the lowest I/O cost.

Our analysis is based on two assumptions: (i) the distribution of queries follows that of the historical query workload $\mathcal{WL}$, and (ii) the caching policy is HFF (highest-frequency-first).

Specifically, Highest-frequency-first (HFF) [25] is a static caching policy that creates the cache offline and fixes the cache content at runtime. It places the most frequent items into the cache, where the frequency of each cache item p (i.e., candidate) is derived from the query workload $\mathcal{WL}$ as: $freq(p) = |\{q \in \mathcal{WL} : p \in C(q)\}|$.

4.1 I/O Cost Estimation

4.1.1 Cost Estimation model

The I/O cost in the candidate refinement phase is decided by the remaining candidate size C_{refine}, which is proportional to $1 - \rho_{hit}\cdot\rho_{prune}$ (by Eqn. 1).

For ρ_{prune}, we rewrite it as $1 - \rho_{refine}$ where ρ_{refine} is the ratio that a (cache-hit) candidate requires refinement. By using the query workload $\mathcal{WL}$, we estimate ρ_{refine} as the average value $\sum_{q\in\mathcal{WL}} \rho^q_{refine}/|\mathcal{WL}|$, where ρ^q_{refine} is the candidate refinement ratio for a specific query point q.

We aim to estimate the cache hit ratio ρ_{hit} in Section 4.1.2 and the candidate refinement ratio ρ^q_{refine} for a specific query point q in Section 4.1.3.

4.1.2 Estimation of ρ_{hit}

Theorem 1 (Estimation of ρ_{hit}). *Let ρ_{hit} and ρ^*_{hit} be the cache hit ratio in our proposed histogram based cache method (with equi-width histogram) and in the exact cache method, respectively. Let N_{item} and N^*_{item} be the number of cache items in our cache and in the exact cache, respectively. Let $|\mathcal{P}|$ be the dataset cardinality, and L_{value} be the number of bits for representing a data value.We have:*

$$\rho_{hit}\begin{cases} \le \frac{L_{value}}{\tau}\cdot\rho^*_{hit} & (\textit{if } N_{item} < |\mathcal{P}|) \\ = 1 & (\textit{otherwise}) \end{cases}$$

Proof: First, we have: $N_{item}\cdot\tau = {N_{item}}^*\cdot L_{value}$.

The proof for the case $N_{item} \ge |\mathcal{P}|$ is trivial. We thus focus on the case $N_{item} < |\mathcal{P}|$. Let f_i be the query frequency of data point i (according to the query log $\mathcal{WL}$). Without loss of generality, for the HFF caching policy, we arrange the points in the cache in descending frequency order, i.e., $f_1 \ge f_2 \ge \cdots \ge f_n$.

According the definition of hit ratio in HFF, we obtain: $\rho_{hit} = \frac{\sum_{i=1}^{N_{item}} f_i}{\sum_{i=1}^{|\mathcal{P}|} f_i}$ and $\rho^*_{hit} = \frac{\sum_{i=1}^{N^*_{item}} f_i}{\sum_{i=1}^{|\mathcal{P}|} f_i}$

Consider the ratio:

$$\frac{\rho^*_{hit}}{\rho_{hit}} = \frac{\sum_{i=1}^{N^*_{item}} f_i}{\sum_{i=1}^{N_{item}} f_i} = \frac{N^*_{item}\cdot\frac{\sum_{i=1}^{N^*_{item}}(f_i)}{N^*_{item}}}{N_{item}\cdot\frac{\sum_{i=1}^{N_{item}}(f_i)}{N_{item}}}$$
$$\ge \frac{N^*_{item}}{N_{item}}\cdot 1 \quad \text{(by Lemma 4)} \qquad (6)$$
$$= \frac{\tau}{L_{value}}$$

Thus, we obtain: $\rho_{hit} \leq \frac{L_{value}}{\tau} \cdot \rho^*_{hit}$ □

Lemma 4 (Average weight monotone non-increasing).

$$\forall N^*_{item} \leq N_{item}, \quad \frac{\sum_{i=1}^{N^*_{item}}(f_i)}{N^*_{item}} \geq \frac{\sum_{i=1}^{N_{item}}(f_i)}{N_{item}}$$

Proof: Trivial. □

4.1.3 Estimation of ρ^q_{refine} upper bound

Theorem 2 (Estimation of ρ^q_{refine} upper bound). *Given a query point q, let b be its k-th upper bound candidate, let D_{max} be the largest candidate distance from q, and let $g_q(x)$ be the probability density function of candidate distances from q. If $g_q(x)$ follows the uniform distribution, then:*

$$\rho^q_{refine} \leq \min\{\frac{||\overrightarrow{\epsilon(b^q_k)}||}{D_{max}}, 1\}$$

Proof: First, we estimate ρ^q_{refine} as:

$$\rho^q_{refine} = \frac{\int_{dist_q(b^q_k)}^{ub^q_k} g_q(x)dx}{\int_0^{D_{max}} g_q(x)dx} = \frac{dist_q(b^q_k)^+ - dist_q(b^q_k)}{D_{max}}$$

According to $dist^+_q(c) - dist_q(c) \leq ||\overrightarrow{\epsilon(c)}||$ (proved in Appendix A), we have:

$$\rho^q_{refine} \leq \frac{||\overrightarrow{\epsilon(b^q_k)}||}{D_{max}}$$

In addition, since $\rho^q_{refine} \leq 1$, we complete the proof. □

Remark: Although we assume $g_q(x)$ to be uniform distribution in the above equation, our estimation is still quite accurate, as shown in our experimental study.

4.2 Determining the Optimal τ

For any histogram, we can apply the I/O cost estimation equations in Section 4.1 for each τ (from 1 to 32) and then choose the one that gives the lowest estimated I/O cost.

For the equi-width histogram, we provide a closed-form equation to estimate the optimal τ in constant time.

4.2.1 ρ^q_{refine} upper bound, for equi-width

For equi-width histogram, we estimate ρ^q_{refine} by Theorem 3. Since the terms in Theorem 3 are independent of q, we can estimate ρ^q_{refine} for equi-width histogram in constant time.

Theorem 3 (ρ^q_{refine} upper bound, for equi-width).

$$\rho^q_{refine} \leq \min\{\frac{\sqrt{d}}{D_{max}}\overline{w}, 1\}$$

where $\overline{w} = 2^{L_{value}-\tau}$ is the bucket width of equi-width histogram, and $D_{max} = cR$ is calculated by using the (R, c)-guarantee in the LSH scheme [13], [29].

Proof: By Lemma 2, we have: $\rho^q_{refine} \leq \frac{||\overrightarrow{\epsilon(b^q_k)}||}{D_{max}} = \frac{\sqrt{d}}{D_{max}}\overline{w}$. In addition, since ρ^q_{refine} cannot be larger than 1, we complete the proof. □

4.2.2 Determining τ, for equi-width

In this section, we derive the optimal τ for the equi-width histogram. Consider the ratio of $\rho_{hit} \cdot \rho_{prune}$ (for our caching) to ρ^*_{hit} (for exact caching). By Lemma 3, we have:

$$\frac{\rho_{hit} \cdot \rho_{prune}}{\rho^*_{hit}} \approx \frac{\frac{L_{value}}{\tau} \cdot \rho^*_{hit} \cdot (1 - \frac{\sqrt{d}}{D_{max}}\overline{w})}{\rho^*_{hit}}$$
$$= \frac{L_{value}}{\tau} \cdot (1 - \frac{\sqrt{d}}{D_{max}}(2^{L_{value}-\tau}))$$

Observe that L_{value}, d are known for a given dataset, and D_{max} can be calculated by the LSH scheme and the query workload.

To find the optimal τ, we simply iterate τ for each value in the range $[1..L_{value}]$, evaluate the ratio $(\rho_{hit} \cdot \rho_{prune})/\rho^*_{hit}$, and then report the τ value leading to the highest ratio.

5 Experimental Study

In this section, we experimentally evaluate the performance of our proposed solutions and baseline solutions. Section 5.1 introduces the experimental setting. Section 5.2 studies the sensitivity of solutions for different configurations (e.g., ordering of the dataset file, caching policy, categories of histograms). Section 5.3 demonstrates the accuracy of our estimation equations. Section 5.4 compares the performance of solutions with respect to various parameters.

All experiments are conducted on a PC with Intel i7-4770 3.40GHz CPU, 16G RAM, and 64-bit Ubuntu 13.04 operating system. The page (block) size in this system is 4KB(4,096 bytes). All algorithms were implemented in C++, and compiled by g++ 4.7.3 with O3 optimization. All datasets and indexes were stored in hard disk and the OS cache was disabled, as in [36].

5.1 Experimental Setup

Datasets and queries: Table 2 summarizes the real datasets to be used in our experimental study. They store the feature vectors extracted from images.

SOGOU, with raw data size 635 GB, is extracted from web images indexed by Sogou[8] (an image search engine in China). We followed [5] to extract a 960-dimensional GIST descriptor from each image. Sogou also provides the query log (of images) for this dataset.

We also use two datasets from [28]: **NUS-WIDE** (extracted from Flickr images), and **IMGNET** (extracted from an online image database)[9]

Next, we split the query log into: (i) a query workload $\mathcal{WL}$, and (ii) a testing query set $\mathcal{Q}_{test}$. A sufficiently large

8. http://www.sogou.com/labs/dl/p2.html

9. http://staff.itee.uq.edu.au/shenht/UQ_IMH/index.htm. The feature vector of them are 150 dimensions color histogram. However, they do not have real query logs. Following [13], [29], we generate the query log by picking random points from $\mathcal{P}$, and then remove those points from $\mathcal{P}$.

TABLE 2: Dataset information

Dataset	d	# of $\mathcal{P}$	# of $\mathcal{Q}_{test}$	size per point	file size
NUS-WIDE	150	267,415	50	600 bytes	136 MB
IMGNET	150	2,213,937	50	600 bytes	1.26 GB
SOGOU	960	8,304,965	50	3,840 bytes	29.7 GB

$\mathcal{WL}$ is used to populate the cache (in Section 2.1), and to construct the histogram (in our solutions).

In each experiment, we execute the queries in $\mathcal{Q}_{test}$ and measure the average query response time per query. We follow [13], [29] and fix the size of $\mathcal{Q}_{test}$ to 50.

Methods for comparison: We consider three baseline methods: NO-CACHE (not using cache), EXACT (caching exact points), and C-VA (caching the whole VA-file)[10]. For C-VA, we tune the number of bits per point so that the VA-file fits into the cache. According to [32], the encoding scheme of VA-file is the same as Equi-Depth.

Our proposed histogram-caching methods share the prefix HC in their names, and apply the following histograms as stated in Section 3.1.

- *Global histogram:* HC-W (equi-width), HC-D (Equi-Depth), HC-V (V-Optimal), HC-O (our optimal histogram for kNN search).
- *Individual-dimension histogram:* iHC-*.
  It uses d histograms. For each dimension j, it builds a histogram $\mathcal{H}_j$ by the corresponding HC-* method.
- *Multi-dimensional histogram:* mHC-R.
  First, we build an R-tree with 2^τ leaf nodes (by using a corresponding node fanout). Then, we map the MBR of each leaf node to a bucket.

All methods use the same index $\mathcal{I}$ in the same experiment. In most experiments, we employ C2LSH [13] as the index $\mathcal{I}$. We use the C2LSH implementation in [13] and its parameter tuning functions. At the end of Section 5.4, we employ exact kNN search indexes (iDistance and VA-file).

Parameters setting: Unless otherwise stated, we use the following default parameter values. The default result size is $k = 10$. The default cache size $\mathcal{CS}$ is set as 40 MB, 400 MB, 8192 MB for NUS-WIDE, IMGNET, SOGOU, less than 30% of the size, respectively. The default code length, $\tau = 10$, is estimated by using our equations in Section 4. We construct our HC-O histogram by Algorithm 2. By default, we run each method by a single thread in each experiment.

5.2 Effect of Configurations

Besides the above parameters, the configurations in our solutions include: ordering of the dataset file $\mathcal{P}$, caching policy, and the categories of histograms. We proceed to examine the sensitivity of these choices on our solutions. We conduct experiments on the SOGOU dataset with the default parameter setting. The experimental results on NUS-WIDE and IMGNET are similar; we omit them due to space reasons.

10. We use VA-file instead of VA$^+$-file. VA$^+$-file [12] requires Karhunen Loeve Transform (KLT), which is not scalable for huge matrices on our datasets.

5.2.1 Effect of caching policy

The caching policy is an choice in our solution. We compare the HFF and LRU policies as described in Section 2.2. As shown in Figure 8, HFF performs better than LRU. Hence, we set HFF as default caching policy in subsequent experiments.

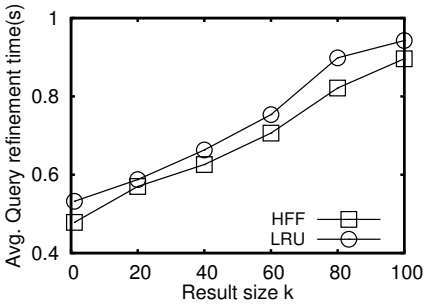


Fig. 8: Effect of caching policy, EXACT caching

5.2.2 Effect of dataset file ordering

We investigate whether the physical ordering of the dataset file $\mathcal{P}$ affects the candidate refinement time T_{refine}. We compare three orderings: (i) the raw ordering in the dataset, (ii) the clustered ordering, which uses the iDistance ordering [20]. (iii) the sorted key ordering, which uses the SK-LSH ordering [35]. In this experiment, we use the EXACT caching method on SOGOU; we obtained similar results for other methods. Figure 9 reports the query refinement time for these orderings. For caching policy HFF, different orderings (Raw, Clustered and Sortedkey) have similar performance. Thus, we use the raw ordering in subsequent experiments.

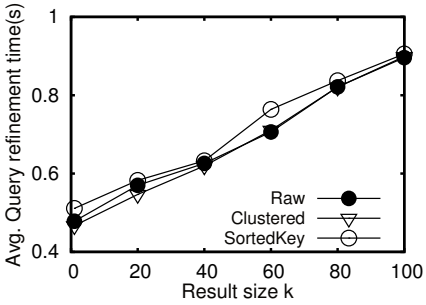


Fig. 9: Effect of dataset file ordering, EXACT caching

5.2.3 Effect of histogram categories

Next, we compare the global histograms (HC-W, HC-D, HC-O) and the individual-dimension histograms (iHC-W, iHC-D, iHC-O), and a multi-dimensional histogram (mHC-R).

Table 3 shows the histogram space (KB), the histogram construction time (s), and the average candidate refinement time T_{refine} during kNN search. Due to the curse of dimensionality, mHC-R is not effective. Global histograms and individual-dimension histograms have similar T_{refine}. However, individual-dimension histogram suffers from high

construction time and occupies more space. For example, it takes 23.8 days to construct iHC-O, but only 35.7 minutes to construct HC-O. Thus, we only use global histograms (HC-W,HC-D, HC-O) in following experiments.

TABLE 3: Effect of histogram categories, on SOGOU

	HC-W	iHC-W	HC-D	iHC-D	HC-O	iHC-O	mHC-R
Space (KB)	8	1,200	8	1,200	8	1,200	1,204
Construction time (s)	0.000	0.004	300	2233	2,140	2.1e6	57.6
Average T_{refine} (s)	0.237	0.230	0.164	0.162	0.123	0.113	0.842

5.2.4 Effect of caching the whole VA-file

We compare methods C-VA and HC-D in Figure 10. While the SOGOU dataset occupies 29.7 GB, we only vary the cache size in the range 1024–6144 MB, which corresponds to 3.4–20 % of the data. At small cache size, C-VA incurs higher time than HC-D because C-VA caches all points but with fewer bits per point. At large cache size, C-VA and HC-D have similar performance since they maintain cache histogram by equi-depth method. Hence, we ignore C-VA in following experiments.

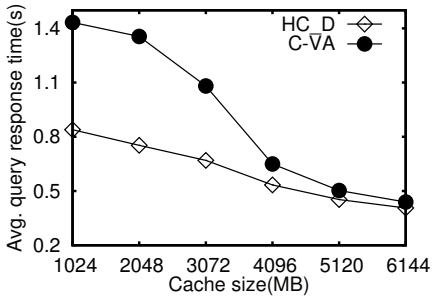


Fig. 10: C-VA and HC-D comparison

5.3 Cost Estimation

Now we test the accuracy of cost estimation model developed in Section 5.3. Figure 12 shows the estimated and the measured I/O cost of the method HC-W, as a function of the code length τ. Observe that the estimated cost is close to the measured cost. Also, the default code length ($\tau = 10$) derived from our cost model is close to the optimal τ measured in the experiment. We also show the optimal τ for each method on each dataset in Table 4.

5.4 Performance Improvement

In this section, we study the performance of our methods with respect to different parameters.

5.4.1 The power of early pruning

Early pruning (including true hit detection) plays an important role in our solution. In this experiment, we study the effectiveness of different histograms for supporting early pruning.

Figure 11 shows the remaining candidate size of the methods with respect to the number of I/O accesses. The performance of mHC-R (R-tree histogram) is bad due to the curse of dimensionality. Observe that HC-O (using our histogram metric) achieves the best performance. On the other hand HC-V (using the SSE histogram metric [19]) does not minimize the I/O cost. In subsequent experiments, we ignore mHC-R due to its bad performance.

Remark: HC-O incurs lower I/O cost than HC-D by 50%.

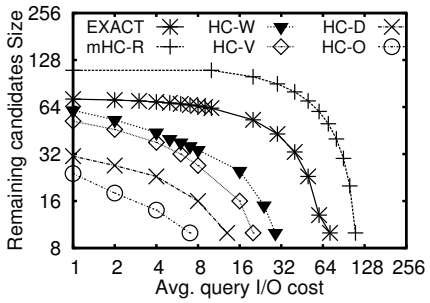


Fig. 11: Remaining candidate size vs query I/O cost, axes in logscale

Table 4 reports the average refinement time of the methods on all datasets with default parameter setting. First, our method HC-O is faster than EXACT by an order of magnitude. Second, although the default code length $\tau = 10$ is not always the optimal (τ^*), our methods still perform much better than EXACT. In subsequent experiments, we ignore HC-V as its performance is unstable; it is worse than HC-W on NUS-WIDE and better than HC-W on IMGNET and SOGOU.

5.4.2 Effect of the cache size $\mathcal{CS}$

Figure 13 plots the average query response time of different methods for different cache size $\mathcal{CS}$. Our caching methods outperform the EXACT caching method. They achieve the best performance when the cache size reaches only 1/3 of the dataset file size. HC-O is the best among all methods. We ignore the baseline methods (NO-CACHE and EXACT) in the following experiments.

5.4.3 Effect of the result size k

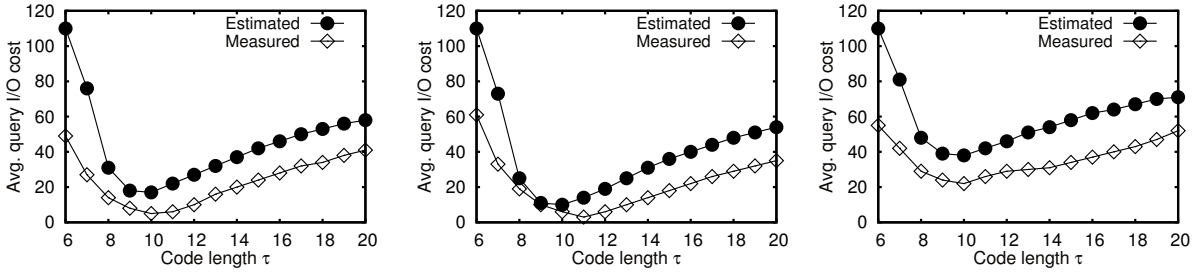
Then we examine the effect of the result size k on our methods. For readability, we plot the average query response time in log scale in Figure 14. The query response time of all methods rises as k increases. HC-O is the best, followed by HC-D, and then HC-W. This result also confirms the effectiveness of our proposed histogram metric (used in HC-O).

5.4.4 Effect of the code length τ

The next experiment investigates the effect of code length τ on our methods. Figure 15 (a),(b),(c) show the average values of $\rho_{hit} \cdot \rho_{prune}$, query I/O cost, and refinement time respectively. Due to the space limit, we only show the experimental results on the largest dataset (SOGOU, 29.7G). Observe that different methods can have different optimal values for τ. For example, the optimal τ for HC-W, HC-D, HC-O are 10, 8, 8, respectively. Again, HC-O is the best and its performance is more robust to τ (e.g., at small τ).

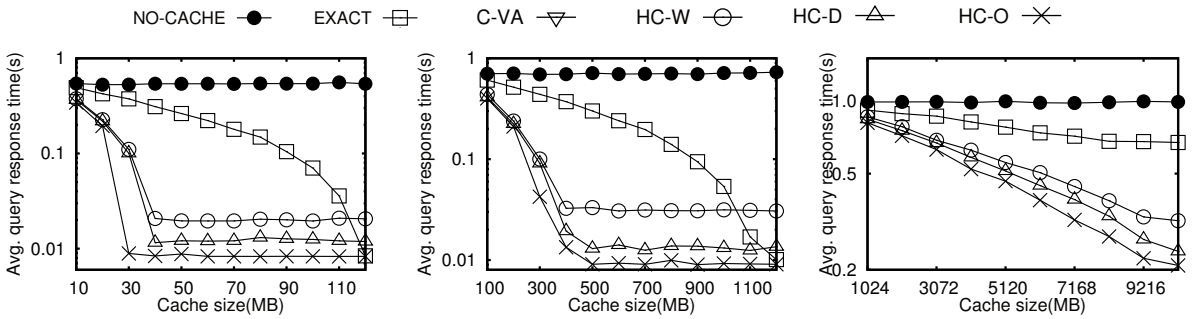
TABLE 4: Avg. refinement time (s) at default $\tau = 10$ and at optimal τ^*

Dataset	EXACT	HC-W			HC-V			HC-D			HC-O		
		Default	Optimal	τ^*	Default	Optimal	τ^*	Default	Optimal	τ^*	Default	Optimal	τ^*
NUS-WIDE	0.3115	0.0451	0.0451	10	0.0555	0.0555	10	0.0110	0.0110	10	0.0087	**0.0087**	10
IMGNET	0.3709	0.0672	0.0495	11	0.0203	0.0182	11	0.0129	0.0112	11	0.0086	**0.0071**	11
SOGOU	0.4803	0.2368	0.2368	10	0.2173	0.1864	8	0.1639	0.0839	8	0.1274	**0.0468**	8



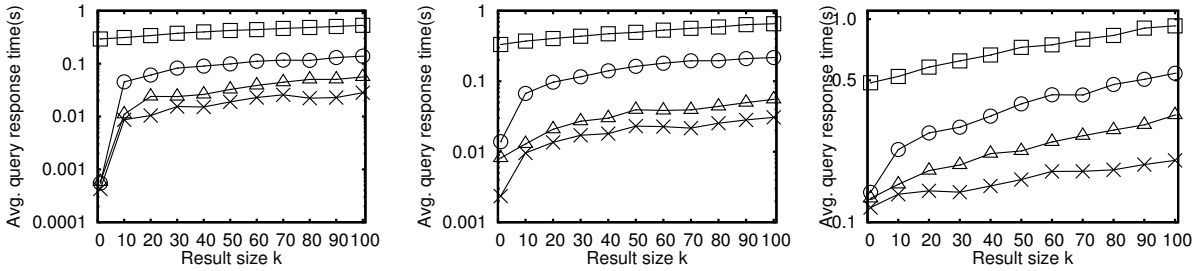
(a) NUS-WIDE (b) IMGNET (c) SOGOU

Fig. 12: The estimated and the measured query I/O cost of HC-W vs. τ, k, $\mathcal{CS}$ at default setting



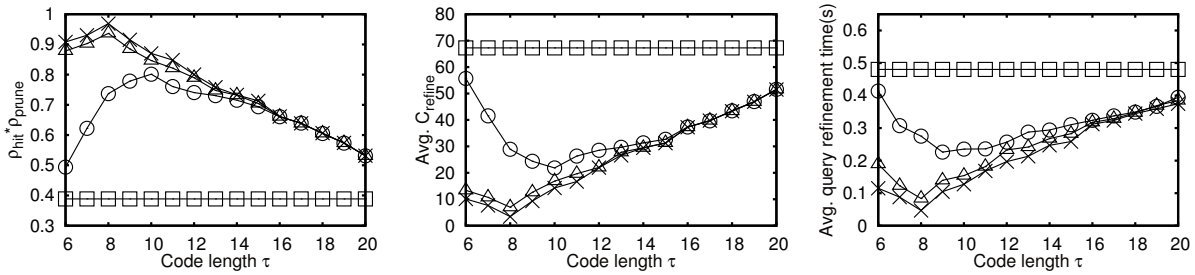
(a) NUS-WIDE (b) IMGNET (c) SOGOU

Fig. 13: Average response time (in logscale) vs. cache size $\mathcal{CS}$, k, τ at default setting



(a) NUS-WIDE (b) IMGNET (c) SOGOU

Fig. 14: Average response time (in logscale) vs. result size k, τ, $\mathcal{CS}$ at default setting

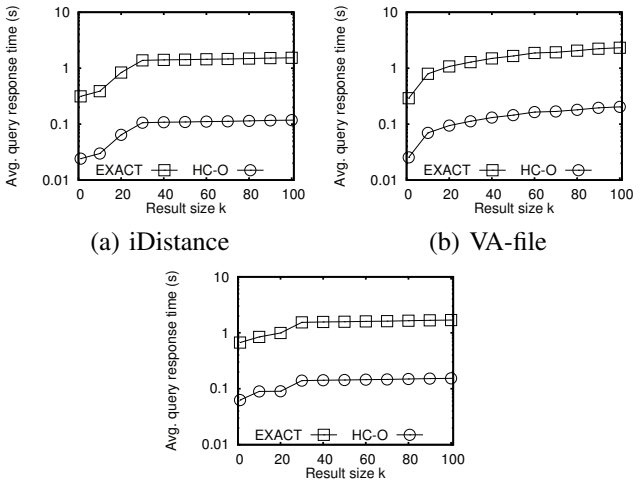


(a) $\rho_{hit} \cdot \rho_{prune}$ (b) I/O cost (c) Refinement time(s)

Fig. 15: Performances vs. code length τ, on SOGOU, k, $\mathcal{CS}$ at default setting

5.4.5 Experiments on exact kNN search indexes

Finally, we compare the performance of HC-O and EXACT on three exact kNN search indexes: iDistance[11], VA-file [32] and VP-tree [4] on IMGNET. Figure 16 shows that the query cost of HC-O is lower than EXACT caching by at least an order of magnitude.



(a) iDistance (b) VA-file

(c) VP-tree

Fig. 16: Exact kNN search indexes, on IMGNET

6 Related Work

Indexing: In the high dimensional exact kNN search, tree-based indexes (e.g., R-tree, X-tree, SR-tree) [3] suffer from *the curse of dimensionality* [33], so their running time for kNN search degenerates to that of linear scan.

We classify existing methods on approximate kNN search into two types: (i) c-approximate methods [7], [10], [13], [17], [23], [24], [29], [35] that provide theoretical result quality guarantees. In particular, C2LSH [13] is the-state-of-the-art disk-based LSH method for computing c-approximate kNN results in sub-linear time, i.e., the result distances are at most c times of the exact result distances, and (ii) other approximate methods [1], [2], [5], [9], [14], [34] that do not guarantee the quality of results.

Our proposed solution can be used on both types of index structures. We provide tighter bounds for pruning unpromising candidates and thus reduce costly I/O access in fetching candidate points. SK-LSH [35] rearrange the data file such that similar points are likely to be placed on the same disk page. This would reduce the I/O cost in the candidate refinement phase. Our work exploits caching techniques to reduce the candidate refinement cost and it is orthogonal to [35].

Vector approximation:

(i) The VA-file [33] and its variant VA$^+$-file [12] provide approximate representations of points in order to speedup linear scan. However, they do not consider the cache and the query workload as in our problem.

11. We use the implementation from:https://code.google.com/p/idistance/

(ii) Vector quantization [21], [30] aims to represent a feature vector by a short code. These codes allow us to derive approximate distances between points efficiently. However, these methods do not guarantee that the approximate distance is always the lower bound distance or the upper bound distance. Thus, they cannot be applied in our kNN search framework.

(iii) Histograms can be used to approximate data values, as we have illustrated in Section 3. In relational databases, histograms are used to capture attribute value distribution and provide selectivity estimation for the query optimizer [18], [19]. The sum squared error (SSE) metric [19] has been designed to formulate the selectively estimation error of a histogram. However, this histogram metric does not necessarily lead to effective candidate pruning in our kNN search problem. In this paper, we propose a suitable histogram metric for kNN search, and construct a corresponding histogram in order to accelerate the refinement phase in kNN search.

7 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) and on exact indexes (iDistance, VP-tree and VA-file).

It is worth noting that our approach is general for any index and achieves promising performance whenever the candidate refinement phase incurs significant time, including other kNN methods [1], [2], [5], [14], [34]. In future, we plan to extend our caching techniques for advanced operations (e.g., kNN join, density-based clustering) on high-dimensional data.

8 Acknowledgement

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

References

[1] V. Athitsos, J. Alon, S. Sclaroff, and G. Kollios. Boostmap: A method for efficient approximate similarity rankings. In *CVPR (2)*, 2004.

[2] V. Athitsos, M. Hadjieleftheriou, G. Kollios, and S. Sclaroff. Query-sensitive embeddings. *ACM Transactions on Database Systems (TODS)*, 32(2):8, 2007.

[3] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.

[4] L. Boytsov and B. Naidan. Learning to prune in metric and non-metric spaces. In *Advances in Neural Information Processing Systems*, pages 1574–1582, 2013.

[5] J. Brandt. Transform coding for fast approximate nearest neighbor search in high dimensions. In *CVPR*, 2010.

[6] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, 1997.

[7] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, pages 253–262, 2004.

[8] R. Datta, D. Joshi, J. Li, and J. Z. Wang. Image retrieval: Ideas, influences, and trends of the new age. *ACM Comput. Surv.*, 40(2), 2008.

[9] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, pages 577–586. ACM, 2011.

[10] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li. Modeling lsh for performance tuning. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 669–678. ACM, 2008.

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

[12] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi. Vector approximation based indexing for non-uniform high dimensional data sets. In *CIKM*, 2000.

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

[14] J. Gao, H. V. Jagadish, W. Lu, and B. C. Ooi. Dsh: data sensitive hashing for high-dimensional k-nnsearch. In *SIGMOD*, 2014.

[15] D. J. H. Garling. The cauchy-schwarz master class: An introduction to the art of mathematical inequalities by j. michael steele. *The American Mathematical Monthly*, 112(6):575–579, 2005.

[16] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

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

[18] Y. E. Ioannidis. The history of histograms (abridged). In *VLDB*, 2003.

[19] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *VLDB*, 1998.

[20] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: An adaptive b^+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.

[21] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(1):117–128, 2011.

[22] H.-P. Kriegel, P. Kröger, P. Kunath, and M. Renz. Generalizing the optimality of multi-step k -nearest neighbor query processing. In *SSTD*, 2007.

[23] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *ICCV*, pages 2130–2137. IEEE, 2009.

[24] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *VLDB*, 2007.

[25] E. P. Markatos. On caching search engine query results. *Computer Communications*, 24(2):137–143, 2001.

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

[27] T. Skopal, J. Lokoc, and B. Bustos. D-cache: Universal distance cache for metric access methods. *IEEE Trans. Knowl. Data Eng.*, 24(5):868–881, 2012.

[28] J. Song, Y. Yang, Y. Yang, Z. Huang, and H. T. Shen. Inter-media hashing for large-scale retrieval from heterogeneous data sources. In *SIGMOD*, 2013.

[29] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*, 2009.

[30] A. Torralba, R. Fergus, and Y. Weiss. Small codes and large image databases for recognition. In *CVPR*, pages 1–8. IEEE, 2008.

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

[32] R. Weber and S. Blott. An approximation based data structure for similarity search. Technical report, Citeseer, 1997.

[33] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, 1998.

[34] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *NIPS*, 2008.

[35] L. Yingfan, C. Jiangtao, H. Zi, L. Hui, and S. Hengtao. Sk-lsh : An efficient index structure for approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 7(9):745–756, 2014.

[36] Y. Zhang and J. Yang. Optimizing i/o for big array analytics. *Proceedings of the VLDB Endowment*, 5(8):764–775, 2012.

Bo Tang received the bachelor's degree in 2012 from Sichuan University, China. He is currently a PhD student in The Hong Kong Polytechnic University, Hong Kong, under the supervision of Dr. Man Lung Yiu. His research interests include similarity search on high dimensional dataset and data exploration on multidimensional dataset.

Man Lung Yiu received the bachelor's degree in computer engineering and the PhD degree in computer science from the University of Hong Kong in 2002 and 2006, respectively. Prior to his current post, he worked at Aalborg University for three years starting in the Fall of 2006. He is now an associate professor in the Department of Computing, The Hong Kong Polytechnic University. His research focuses on the management of complex data, in particular query processing topics on spatiotemporal data and multidimensional data.

Kien A. Hua is a Professor of Computer Science, and Director of the Data Systems Lab at the University of Central Florida. He served as the Associate Dean for Research of the College of Engineering and Computer Science at UCF. Prior to joining the university, he was a lead architect at IBM Mid-Hudson Lab, where he helped develop a highly parallel computer system, the precursor to the highly successful commercial parallel computer known as SP2. Dr.Hua received his B.S. in Computer Science, and M.S. and Ph.D. in Electrical Engineering, all from the University of Illinois at Urbana-Champaign. His diverse expertise includes network and wireless communications, image/video computing, sensor networks, medical imaging, databases, mobile computing, and intelligent transportation systems. He has published widely, with over 10 papers recognized as best/top papers at conferences and a journal. Many of his research have had significant impact on society. His Chaining technique started the peer-to-peer video streaming paradigm. His Skyscraper Broadcasting, Patching, and Zigzag techniques have been heavily cited in the literature, and have inspired many commercial systems in use today. Dr. Hua has served as a Conference Chair, an Associate Chair, and a Technical Program Committee Member of numerous international conferences, as well as on the editorial boards of a number of professional journals. Dr. Hua is a Fellow of IEEE.

APPENDIX

A. PROOF OF LEMMAS

In the following analysis, we employ vectors to express distance computations in a concise manner.

Definition 11 (Distance on vectors). *Let q be a query point and c be a candidate point. The Euclidean distance is $dist_q(c)$, and the upper distance bound $dist_q^+(c)$ can be expressed in terms of the dot product of vectors:*

$$dist_q^+(c) = ||(\overrightarrow{q} - \overrightarrow{c^u})||$$

where $\overrightarrow{c^u} = [c^u.1, c^u.2, ..., c^u.d]$ are defined by a histogram $\mathcal{H}$ as follows:

$$c^u.j = \begin{cases} l_{\mathcal{H}(c.j)} & \text{if } |q.j - l_{\mathcal{H}(c.j)}| > |q.j - u_{\mathcal{H}(c.j)}| \\ u_{\mathcal{H}(c.j)} & \text{otherwise} \end{cases}$$

Proof of Lemma 1: Let $\overrightarrow{A} = \overrightarrow{c} - \overrightarrow{q}$ and $\overrightarrow{\Delta_c} = [\Delta_c.1, \Delta_c.2, ...\Delta_c.d]$. Calculate $\Delta_c.j$ as:

$$\Delta_c.j = \begin{cases} \epsilon(c).j & \text{if } c^u.j > c^l.j \\ -\epsilon(c).j & \text{otherwise} \end{cases}$$

Since $||\overrightarrow{\Delta_c}|| = ||\overrightarrow{\epsilon(c)}||$ and $dist_q(c) = ||\overrightarrow{A}||$ always holds, we have:

$$\begin{aligned} &dist_q^+(c) = ||(c^u - \overrightarrow{q})|| \leq ||(\overrightarrow{\Delta_c} + \overrightarrow{c} - \overrightarrow{q})||^2)^{\frac{1}{2}} \\ &= (\overrightarrow{\Delta_c} \cdot \overrightarrow{\Delta_c} + 2\overrightarrow{\Delta_c} \cdot (\overrightarrow{c} - \overrightarrow{q}) + (\overrightarrow{c} - \overrightarrow{q}) \cdot (\overrightarrow{c} - \overrightarrow{q}))^{\frac{1}{2}} \\ &= (||\overrightarrow{\Delta_c}||^2 + 2\overrightarrow{\Delta_c} \cdot \overrightarrow{A} + ||\overrightarrow{A}||^2)^{\frac{1}{2}} \\ &\leq (||\overrightarrow{\Delta_c}||^2 + 2||\overrightarrow{\Delta_c}|| \cdot ||\overrightarrow{A}|| + ||\overrightarrow{A}||^2)^{\frac{1}{2}} \text{(by Cauchy inequality [15])} \\ &= ||\overrightarrow{\Delta_c}|| + ||\overrightarrow{A}|| = ||\overrightarrow{\epsilon(c)}|| + dist_q(c) \end{aligned}$$

Then, we have: $dist_q^+(c) - dist_q(c) \leq ||\overrightarrow{\epsilon(c)}||$.

Proof of Lemma 2:

$$\begin{aligned} &\sum_{q \in \mathcal{WL}} \sum_{r=1}^{k} ||\overrightarrow{\epsilon(b_r^q)}||^2 = \sum_{b_r^q \in \mathcal{QR}} ||\overrightarrow{\epsilon(b_r^q)}||^2 \text{ by Eqn.2} \\ &= \sum_{b_r^q \in \mathcal{QR}} \sum_{j=1}^{d} (\epsilon(b_r^q).j)^2 \quad \text{by Def. 10} \\ &= \sum_{b_r^q \in \mathcal{QR}} \sum_{j=1}^{d} (u_{\mathcal{H}(\epsilon(b_r^q).j)} - l_{\mathcal{H}(\epsilon(b_r^q).j)})^2 \\ &= \sum_{x=1}^{N_{dom}} F'[x] \cdot (u_{\mathcal{H}(x)} - l_{\mathcal{H}(x)})^2 \quad \text{by Eqn.3} \\ &= \sum_{i=1}^{B} \sum_{x=l_i}^{u_i} F'[x] \cdot (u_i - l_i)^2 \quad \text{group by bucket id} \end{aligned}$$

B. GLOBAL VS. MULTI-DIMENSIONAL HISTOGRAM

Observe that each approximate point p' is associated with a bounding rectangle. Let w_{br} be the average width of dimensions of a bounding rectangle. We proceed to compare the value of w_{br} for the global histogram and the multi-dimensional histogram. In the following discussion, we assume that data points fall in the space $[0, 1]^d$ with uniform distribution.

We take an equi-width histogram as a simple global histogram. The number of buckets is 2^τ, where τ is the code length. Then we derive: $w_{br} = \frac{1}{2^\tau}$. Note that this value is independent of the dimensionality d.

Assume that a multi-dimensional histogram partitions the d-dimensional space such that each rectangle contains at least 2 points (out of n points in the dataset). The average volume of each rectangle is at least $\frac{2}{n}$. Thus, we derive $w_{br} \geq (\frac{2}{n})^{\frac{1}{d}}$. Unfortunately, this value rises rapidly with the dimensionality d.

As an example, we set the data size $n = 1000000$, the dimensionality $d = 100$, and the code length $\tau = 8$. For the global histogram (equi-width), we have: $w_{br} = \frac{1}{2^8} = 0.0039$. For the multi-dimensional histogram, we have: $w_{br} \geq (\frac{2}{1000000})^{\frac{1}{100}} = 0.877$. This example demonstrates that the global histogram achieves a much smaller w_{br} than the multi-dimensional histogram.