Selective Hashing: Closing the Gap between Radius Search and k-NN Search

Jinyang Gao[‡] H. V. Jagadish[§] Beng Chin Ooi[‡] Sheng Wang[‡]
[‡]School of Computing
National University of Singapore
{jinyang.gao, ooibc, wangsh}@comp.nus.edu.sg
[§]Department of EECS
University of Michigan
jag@umich.edu

ABSTRACT

Locality Sensitive Hashing (LSH) and its variants, are generally believed to be the most effective radius search methods in high-dimensional spaces. However, many applications involve finding the k nearest neighbors (k-NN), where the k-NN distances of different query points may differ greatly and the performance of LSH suffers. We propose a novel indexing scheme called *Selective Hashing*, where a disjoint set of indices are built with different granularities and each point is only stored in the most effective index. Theoretically, we show that k-NN search using selective hashing can achieve the same recall as a fixed radius LSH search, using a radius equal to the distance of the $c_1 k_{th}$ nearest neighbor, with at most c_2 times overhead, where c_1 and c_2 are small constants. Selective hashing is also easy to build and update, and outperforms all the state-of-the-art algorithms such as DSH and IsoHash.

1. INTRODUCTION

Efficient k-nearest neighbor (k-NN) search is essential for a wide range of applications in the areas such as information retrieval, data mining and machine learning. Objects are frequently characterized as feature vectors, and represented as points in a multi-dimensional space. Access methods in multi-dimensional space, including the k-NN problem, have been studied extensively. However, most of them suffer from the so-called curse of dimensionality and demonstrate poor performance when the number of dimensions is high [27]. One technique that shows promise in high-dimensional spaces is locality sensitive hashing (LSH) [10]. LSH is designed for finding points within a fixed radius of the query point, i.e., radius search. For a k-NN problem (e.g., the first page results of search engine), the corresponding radii for different query points may vary by orders of magnitude, depending on how densely the region around the query point is populated. To consider a human analogy: in a crowded city, we may be able to find the k-NN even within our own building, whereas in the middle of the desert, we may have to go many miles even to find the k-NN. In such a case, LSH has to either (i) build index for different radii $R, cR, c^2R, \dots$, which increases the query time and index storage cost, or (ii) use an ad-hoc radius, which voids any quality guarantee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
KDD '15, August 11 - 14, 2015, Sydney, NSW, Australia.
Copyright is held by the owner/author(s). Publication rights licensed to ACM..

Recently, there has been increasing interest in designing learning-based hashing functions [2, 29, 9] to alleviate the limitations of LSH. If the data distributions only have *global* density patterns, good choices of hashing functions may be possible. However, it is not possible to construct global hashing functions capturing diverse *local* patterns. Actually, k-NN distance (i.e., the desired search range) depends on the local density [26] around the query, e.g., a local feature such as a dense group or sparse hull. Since every global function is used for all the data points, when it is learnt to adjust to one local pattern, it will also cause lots of side-effects for other areas. In such cases, no global choice can be good.

We propose the concept of *Selective Hashing* to give a locally optimized index option for every data point. It is a meta-algorithm for k-NN search which works on the top of radius search algorithms such as LSH. Our main innovation is to create multiple LSH indices with different granularities (i.e., radii). Then, every data object is only stored in one selected index, with certain granularity that is especially effective for k-NN searches near it. Using this idea, our method achieves good quality for k-NN search regardless what the distance is, while avoiding having to maintain a huge set of complete indices for various distance radii (and additional query cost to visit multiple indices).

Within the framework of selective hashing, we further study what kind of granularity is most effective for a data point. A main insight here is that selecting the granularity has the equal effect of setting a *guard range* for every data object, i.e., when a query point falls into that range, this data object will be checked. Although k-NN is not a symmetric relation, the inverse relation can be satisfied with high confidence for only a small constant overhead. For example, a 20-NN of a point will have 99% probability to be a reverse 100-NN of that point as well. Thus, for every data object, we estimate its density, decide the *guard range* with certain confidence, and select the index with corresponding granularity.

Comparing k-NN search with radius search, the only difference is that k-NN search does not provide the radius in the query. Thus, it is a strictly harder problem than the radius search where the radius equals to the distance of k_{th}-NN. As LSH has been refined as a technique, it is now near-optimal for high-dimensional radius search, and the gap between the current solution [1] and the lower bound [18] is almost closed. However, no such good results exist for the k-NN problem and there remains a gap between efficient k-NN search and fixed radius search. Theoretically, we show that k-NN search using selective hashing can achieve the same recall as fixed $c_1 k_{th}$-NN radius LSH search with at most c_2 times overhead, where c_1 and c_2 are small constants. If the density does not change dramatically from one object to its nearest neighbors, we can show that c_1 is less than 5 and c_2 is less than 1.3. Experimentally, the observed overheads is under a factor of two in all the cases.

We propose an efficient algorithm to build the selective hashing index, as well as to handle updates. Unlike most learning-based approaches, selective hashing only needs to maintain the local density for each area, and re-select the radius when necessary. The amortized update cost is independent of the data size, and only depends linearly on the number of indices. We also consider write-intensive applications, for which we propose a lazy updating strategy to re-select the radius when the check-rate (i.e., the proportion of data to be accessed) for a certain object is high in the past queries.

We summarize the contributions of the paper as follows.

• We propose a novel meta-method for indexing, called *Selective Hashing*, which combines multiple index instances together and is especially suitable for k-NN queries. Compared with previous works, selective hashing is the first to optimize the index structure *locally* for every data point. Further, it provides good quality results regardless of the distance of k-NN and does not incur extra storage cost in consuming multiple indices (Section 3).

• We develop an effective index-selection scheme under the selective hashing framework. Our scheme guarantees high recall while achieving check-rate complexity comparable to a fixed radius search. In addition, this index is easy to construct and maintain with a cost that is comparable to the conventional LSH (Section 4).

• Experimental results show that selective hashing is effective for k-NN queries: producing high recall results while incurring low query time (i.e., low check-rate), compared to several leading alternative strategies. It is also small in size and fast to update incrementally (Section 4.4).

2. PRELIMINARIES

In this paper, we consider data objects represented as points in a d-dimensional metric space R^d. We use $\|p, q\|$ to denote the distance between two points p and q. Given a dataset, a point $q \in R^d$ and an integer k, the k-NN of the query q is the k data points that are nearest to q in the dataset. In the high-dimensional case, for the reason that the cost of exact k-NN problem usually grows linearly with the size of dataset, approximate k-nearest neighbor problem is accepted as a compromise.

Locality Sensitive Hashing [10] (LSH) is an efficient approximate algorithm for high-dimensional similarity search. It is efficient and provides a rigorous quality guarantee for finding similar points within a radius r. We denote the sphere centered at point q by $B(q, r)$. Formally, $B(q, r) = \{p | p \in R^d, \|p, q\| \leq r\}$. The general idea of LSH is that objects that are within a given distance will be hashed to the same value with high probability, where the set of hash functions called LSH family:

DEFINITION 1. **(LSH Family, $\mathcal{H}$)** *A family $\mathcal{H} = \{h : R^d \rightarrow U\}$ of functions is called (r, cr, p_1, p_2)-sensitive if $\forall p, q \in R^d$:*

• *if $p \in B(q, r)$, then $Pr_{\mathcal{H}}(h(p) = h(q)) \geq p_1$;*

• *if $p \notin B(q, cr)$, then $Pr_{\mathcal{H}}(h(p) = h(q)) \leq p_2$;*

LSH family for Euclidean distance is accomplished by random Gaussian projection. There are also various versions of LSH family for Jaccard similarity, cosine distance, Hamming distance etc.

LSH usually applies a concatenation of m hashing functions randomly selected from the hashing family to build the hash table(reducing check-rate), and builds l independent hash tables and takes the union of results (raising recall). m and l are parameters of LSH and the best value is related to the data size. In querying phase, all points collide with the query point in any of the hash tables will be considered as candidates. The exact distances between the query and all the candidates will be validated. For k-NN query, the top-k in candidates will be returned and for radius query, those satisfy the radius threshold will be returned. Consider a group of LSH tables as a whole, it provides the following properties:

LEMMA 1. *Given a group of LSH tables denoted as $\mathcal{G}$, $\forall p, q \in R^d$, $\mathcal{G}(p, q)$ is true iff p and q collide in at least one of the hash tables. Then:*

• *if $p \in B(q, r)$, then $Pr(\mathcal{G}(p, q)) \geq P_a$;*

• *if $p \notin B(q, cr)$, then $Pr(\mathcal{G}(p, q)) \leq P_b$;*

• *$P_a = 1 - (1 - p_1^m)^l$, $P_b = 1 - (1 - p_2^m)^l$, $P_b \ll P_a$.*

In subsequent sections, we will consider a group of LSH hash tables $\mathcal{G}$ as a black-boxed index, offering a radius search with some recall guarantee. Taking p_1, p_2 and m as inner parameters optimized based on a certain approximation ratio c, and tuning l to adjust the recall requirement, we have:

LEMMA 2. *To get a recall $P_a = 1 - \delta$, the check-rate P_b for points outside $B(q, cr)$ for $\mathcal{G}$ is bounded by $\log \delta * \phi$, where ϕ is an inner property of $\mathcal{G}$.*

PROOF.
$$\begin{aligned}
&\delta = (1 - p_1^m)^l \\
\Rightarrow &l = \log \delta / \log (1 - p_1^m) \\
\Rightarrow &P_b = 1 - (1 - p_2^m)^l \\
\Rightarrow &P_b \leq l * p_2^m \\
\Rightarrow &P_b \leq \log \delta * p_2^m / \log (1 - p_1^m)
\end{aligned}$$

Here, $\phi = p_2^m / \log (1 - p_1^m)$ is an inner property of $\mathcal{G}$. □

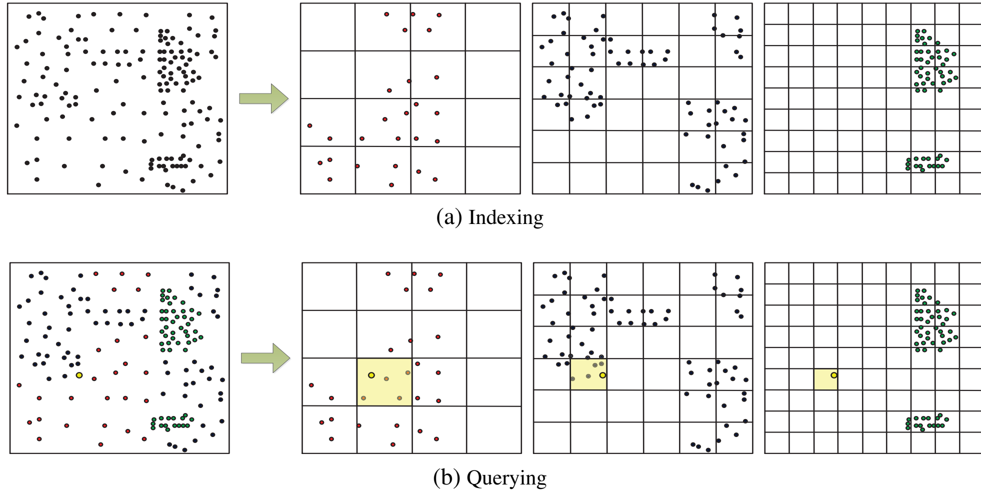
The number of tables (i.e., index size) is also proportional to $\log \delta$ (it is also relevant to the data size in a theoretically optimized solution). Note that achieving a high recall typically does not lead to a significant increase in query complexity (or the number of tables), as it only affects the term $\log \delta$.

To simplify the analysis here we use Euclidean Selective Hashing and Euclidean LSH (i.e., E2LSH) as a running example throughout the rest of discussion. The Euclidean requirement is not compulsory, since we only need a black-boxed radius search index that can offer arbitrary high recall as required. In our subsequent analysis, we also only need the properties of a metric space. Given a radius search algorithm for any metric space with such recall and check-rate guarantee, selective hashing can support k-NN search for such space by using it as a black box.

The theoretical bound of Euclidean LSH can be presented in two ways depending on the problem requirement. If the checking procedure is terminated when sufficient number of points in $B(q, cr)$ is found, then it returns a c-approximate r-NN solution, and its query time is $O(dn^{1/c})$. If all the buckets are fully checked and only those points within $B(q, r)$ are returned, then the results are exact r-NN with a constant recall δ. Usually there is no theoretical bound on the check rate for such an exact solution, since LSH cannot distinguish the points in $B(q, cr)$ - $B(q, r)$ from those in $B(q, r)$. However, in practice, the worst case seldom happens. In this paper, we only focus on the overhead using LSH to solve k-NN problem, and the main result can be achieved by using both versions of theoretical analysis. Since most LSH implementations, such as E2LSH [7], choose to do exact search, and recall is the most common measure for the k-NN problem, we adopt the exact search LSH as the base algorithm for the rest of our analysis.

3. SELECTIVE HASHING FRAMEWORK

Selective hashing is a meta-method for indexing. It combines multiple index instances together, e.g., black-boxed LSHs, and produces a more efficient index without extra storage or query cost. Considering that we have multiple index choices for data objects, a traditional approach usually includes all the objects in every index. At query time, there is only one index or a few indices to be consulted. In contrast, selective hashing builds multiple indices, with each object in the dataset being placed in only one index. At query



(a) Indexing

(b) Querying

Figure 1: Overview Intuition of Selective Hashing.

time, all the indices have to be consulted to locate the objects of interest. Thus, selective hashing works in a "select one, query all" manner. It is easy to see that this scheme can provide correct and complete results: each object exists in exactly one of those indices, each of which will be accessed for any single query. The check rate of each object only depends on the index in which it is stored. Each index only contains a disjoint fraction of objects, and hence selective hashing will not bring in any additional cost (especially for hash-based methods with $O(1)$ lookup cost). Moreover, there is almost no extra storage overhead as well. In contrast, traditional methods usually need to build multiple complete index instances with all the data objects stored, which may lead to a huge index size and limit the possibility of combining more indices.

We create multiple indices with different radii $R, cR, c^2R, \ldots$, and adopt selective hashing on the top of these indices. Figure 1 gives an example of our workflow. In the indexing phase, for each data object, we select one index unit to use. The index selection is based on the local density around that data point. Finally, we will have a group of indices, and each data object is stored in only the most suitable one. In Figure 1a we see how each data point is mapped to one of three indices (with c=1.5). Data points in dense regions are stored in the right (small granularity) index, while data points in sparse regions are stored in the left (large granularity) index. In the querying phase, we push down the query to all the indices and collect the results. Then we aggregate the results and return the top k. In Figure 1b all the indices are used to retrieve the nearest neighbors of the given query. The search range in each index is indicated by the shaded cell. The nearest neighbors are found in the left two indices, with no nearby neighbors at all in the right index. In the dense area, most of the data points tend to be in indices with small granularities. When a query falls into such areas, although indices with large granularity are also checked, the data points are only stored and accessed in indices with small granularities. Thus, the actual search range is small accordingly to avoid unnecessary checking but large enough to get accurate results. In the sparse area, however, the query needs a larger search range. Data points are stored in indices with larger granularities to ensure recall. In short, the search range for a query is automatically tuned so that only the potential k-NN candidates will be checked.

From the perspective of data objects (or points), we can also see that every data object has chosen its best local structure for k-NN search. For a k-NN search problem, the best local structure for a query is to only store k-NN while leaving other data objects outside. Note that only query-object relations are considered and queries are pushed to all the indices, so similar objects can choose different radius and be placed in different indices. Thus, the radius selected by each object is determined independently. Although all the hash functions are defined globally, only those with suitable radius are used for each object. Essentially, we use the global hash tables to offer locally optimized index structure. While there are learning-based methods, such as DSH [9], that attempt to optimize hash functions based on the data, they are much harder to tune, since every hash function affects all objects. Now the only problem is that we can only select the index for each data point, not each query. Notice that data p is a k-NN of query q equals with q is a reverse k-NN of p. We need to choose the radius that is similar to the range that contains all the reverse k-NN (i.e., potential queries) of the data point. k-NN is not a symmetric relation, however, we will build a connection between them in the next section.

4. ALGORITHM AND ANALYSIS

4.1 Density Estimation

As mentioned before, our radius selection is based on the local density near each object. Many methods have been proposed for density estimation on multi-dimensional data [21, 26]. We shall first define density to serve the need in radius selection, in a manner inspired by these previous methods. Then we propose a simple density estimation algorithm based on collisions in LSH tables.

4.1.1 Density Definition

DEFINITION 2. **(Data Density Observation $D_o(p, r)$)** *We define the density observation of one point p within radius r as the number of points in the area $B(p, r)$. For a given dataset O, the density observation can be represented as $D_o(p, r) = |\{o | o \in O, o \in B(p, r)\}|$.*

This definition makes all the following analysis independent of the properties of space: all we need is a distance measure. Note also that LSH is considered as a black box with only radius r and recall δ as parameters. Henceforth, we only need to consider the k-NN problem in a metric space.

Further, we consider the generation model behind density observation. Points appearing in a certain area can be formally defined as a spatial Point Process [14]. Although the density of different areas may be different, by looking into two disjoint areas, the counts of points are independent of each other but only depend on their own densities. Thus, this point process can be described as a spatial

Poisson process [14] where the number of points in a certain area (e.g., density observation) follows Poisson distribution. We define the expectation of $D_o(p, r)$ as *actual data density* $D_a(p, r)$.

DEFINITION 3. **(Actual Data Density** $D_a(p, r)$**)**
We consider data objects that are generated from spatial Poisson process. The number of points in a certain area $B(p, r)$ *is a random variable following Poisson process, and its expectation value is defined as the actual density* $D_a(p, r)$. $D_o(p, r)$ *is the observation of* $Poisson(D_a(p, r))$.

The above two definitions emphasize two different aspects of the density concept respectively. From the original physical perspective, density is described as the mass divided by a unit volume, which is an observation. And from the statistical view point, density refers to the probability density function (PDF) of a certain random variable, which is the hidden parameter behind the observation. Therefore, we use two terms, density observation D_o and actual density D_a, to eliminate this ambiguity. For a Poisson distribution $F_{Poisson}(x; \lambda)$, when λ is more than 10, its cumulative distribution function (CDF) can be well approximated by the CDF of Gaussian distribution [14] [1]:

$$CDF_{Poisson(\lambda)}(x) \approx CDF_{Gaussian(\mu=\lambda;\sigma^2=\lambda)}(x) = \Phi(\frac{x - \lambda}{\sqrt{\lambda}})$$

In most of the density estimation works [21, 26], there is one common but indispensable assumption: regardless of the randomness of observation, the actual density should be a continuous and smooth enough function over its domain space. Only with this assumption, can we recover the actual density from a set of isolated points. In our work, we also make a similar assumption about the continuity of actual density — λ-continuity for actual data density. It assumes that for two points within a k-NN distance, the actual density changes at most λ times.

DEFINITION 4. **(λ-Continuity for Actual Data Density)**
Given a dataset O*, the* λ*-continuity for data density is satisfied iff:* $\forall p$*, let* R *be its* k_{th}*-NN distance, i.e.,* $D_a(p, R) = k$*, then* $\forall o \in B(p, R)$ *and* $r \leq R$*,* $D_a(o, r) \leq \lambda D_a(p, r)$.

λ-continuity for data density assumes that the actual density should not change dramatically (i.e., λ times) among the nearest neighbor pairs. For most typical real datasets, even with high skew, λ is no more than 1.5-3. It is interesting that even when outlier clusters exist, λ will still be relatively low (e.g., 3), since the actual density is much smoother than the observation.

4.1.2 Observation Estimation

After having provided a formal definition of density, we are now ready to discuss the estimation algorithm. Specifically, we aim to get the observation $D_o(p, r)$. Indeed, the exact value of $D_o(p, r)$ can be directly counted by enumerating the whole dataset. However, the computation cost is usually not affordable — the complexity of computing the observations for all N data points is $O(N^2)$. In the following analysis, we are only concerned with the lower and upper bounds of such observations. It is therefore sufficient for us to estimate these bounds for $D_o(p, r)$ effectively.

A natural idea here is that the estimation can be directly obtained from LSH tables via collision counting. Collisions indicate how many points are within the radius. Combining the counts from a group of tables, the estimation can be quite accurate.

[1] In this paper, the main bulk of analysis are from the perspective of the CDF of Gaussian. Note that we can also analyze from the CDF of Poisson and similar results can be derived. However, the CDF form of standard Gaussian Φ is more intuitive and commonly used.

THEOREM 1 (LOWER BOUND FOR $D_o(p, r)$). *Given an LSH index* $\mathcal{G}$ *with radius* r/c *and properties* P_a *and* P_b *described in Lemma 1, using* $\#Collision(p, r)$ *to denote the number of points collides with p (i.e.,* $\mathcal{G}(p, q)$ *is true), then with a probability of* ζ*,* $D_o(p, r)$ *will be large than* $LB(p, r)$*, where:*

$$LB(p, r) = \#Collision(p, r) - P_b * N - \Phi^{-1}(1 - \zeta) * \sqrt{P_b * N}$$

PROOF. Recall that from an LSH group with a radius $r' = r/c$ and approximate ratio c: for all collided points of p, the expected number of points outside $B(p, cr')$ (i.e., $B(p, r)$) is at most $P_b * N$. Besides, the collision of one data point is independent with the others. Thus, the count of such collisions follows a Poisson distribution with a λ (i.e., expectation) less than $P_b * N$. Using Gaussian to approximate Poisson, it follows $Gaussian(P_b * N, P_b * N)$. The PDF of Gaussian (or Poisson) drops exponentially in the tail (3 standard variances or more), we can compute an upper bound $UB(p, r)$ for such a random variable with confidence $1 - \zeta$:

$$\begin{aligned}
&x \sim Poisson(P_b * N) \\
&\Pr(x < UB(p, r)) = 1 - \zeta \\
\Rightarrow &UB(p, r) = CDF_{Poisson(P_b * N)}{}^{-1}(1 - \zeta) \\
\Rightarrow &UB(p, r) = P_b * N + \Phi^{-1}(1 - \zeta) * \sqrt{P_b * N}
\end{aligned}$$

Φ is the CDF of the standard Gaussian. The collisions consist of two parts: one part is from points outside $B(p, r)$ which has an upper bound derived above, the other part is from points within $B(p, r)$. Thus, we have the lower bound for $D_o(p, r)$ denoted as $LB(p, r)$, where $LB(p, r) = \#Collision(p, r) - UB(p, r)$. □

Although this bound is not tight, it tells us that we can obtain a similar estimation in a range that is c times larger since $D_o(p, r/c) * P_a$ points will appear in $\#Collision(p, r)$. Note that in the radius selection, the step size of choosing the next radius is also c times. Therefore at the worst case we select the next larger radius than the most suitable one. In real applications, this estimation works well that the lower bound is about a half of the actual value.

The upper bound for $D_o(p, r)$ can be derived analogously. In what follows, we will directly use the value $D_o(p, r)$ without considering how it is estimated (or simply counted). Giving a more accurate (or even faster) estimation is not the main focus of this paper and we deal it only briefly here.

4.2 Radius Selection

Armed with the power of λ-continuity for data density and density observation of data points, we are ready to solve the core radius selection problem. We first present the selection algorithm that satisfies the quality guarantee, and then analyze the cost of query.

4.2.1 Selection Algorithm

The key problem of selective hashing is to choose an index with suitable radius for each data point. A main insight here is that by selecting a radius for a data point, we are in fact setting a *guard range* for this point, i.e., when a query point falls into that range, this point will be checked. We use the term *target queries* of data p to denote those potential queries which contain p in their k-NN results. Obviously, k-NN is not a symmetric relation and we cannot guarantee that all target queries will fall into the guard range (unless we set the guard range to be the full space). However, the confidence of target queries falling into the guard range could be arbitrary high by the following detailed analysis. Thus, to ensure a recall of $1 - \delta$, we allocate the allowed error rate δ into two parts: (1) $\delta/3$ for cases where LSH fails to get k-NN when target queries fall into guard range; and (2) $2\delta/3$ for cases where the target queries fall outside the guard range. The split parameter (1/3, 2/3) here is optional, since we only aim to build a linear connection

between radius search and k-NN search while the constant does not matter at the theoretical level.

To match the first part, we will build our LSH table groups using $\delta' = \delta/3$. Then we consider how to find the guard range (i.e., radius to be selected) so that the overall probability of target queries falling outside the guard range of their k-NN is bounded by $2\delta/3$.

THEOREM 2 (RADIUS SELECTION). *Given a dataset under λ-continuity assumption, and using ϕ to denote $\Phi^{-1}(1-\delta/3)$, let $\mathcal{B}_k = \lambda k' + \Phi^{-1}(1-\delta/3)\sqrt{\lambda k'}$, where $k' = k + \phi(\sqrt{\phi^2+4k}+\phi)$. Given a data point o, the confidence [2] of target queries falling into r where $D_o(o,r) \geq \mathcal{B}_k$ is larger than or equal to $1-2\delta/3$.*

PROOF. For a k-NN query q, we denote the distance between the query and its k_{th}-NN point as r', and we have $D_o(q,r') = k$. As $D_o(q,r')$ is an observation of $Poisson(D_a(q,r'))$, we know that with a confidence of $1-\delta/3$:

$$D_a(q,r') - \phi\sqrt{D_a(q,r')} \leq k$$
$$\Rightarrow D_a(q,r) \leq k + \phi(\sqrt{\phi^2+4k}+\phi)$$

We denote this upper bound of confidence interval as k'.

Now we consider k-NN of q and we want to determine the guard range. First we analyze $D_o(o,r')$ for $o \in B(q,r')$ (i.e., o is a k-NN of q). According to λ-density continuity, we have $D_a(o,r') \leq \lambda D_a(q,r')$. Besides, we also know $D_o(o,r') \sim Poisson(D_a(o,r'))$. Thus with probability $1-\delta/3$ (splitting the error $2\delta/3$ again into two halves):

$$D_o(o,r') \leq D_a(o,r') + \phi\sqrt{D_a(o,r')}$$
$$\leq \lambda D_a(q,r') + \phi\sqrt{\lambda D_a(q,r')}$$

Since $D_a(q,r') \leq k'$ with confidence $1-\delta/3$, the following statement is satisfied with the probability larger than $1-2\delta/3$:

$$D_o(o,r') \leq \lambda k' + \phi\sqrt{\lambda k'}$$

We denote this upper bound of confidence interval for $D_o(o,r')$ as $\mathcal{B}_k$. Up to now, we know that with confidence larger than $1-2\delta/3$, $D_o(o,r)$ will not exceed $\mathcal{B}_k$. Therefore, if we set the guard range of the data object to be r, where $D_o(o,r) \geq \mathcal{B}_k$, with confidence larger than $1-2\delta/3$ we have $r' \leq r$, which means the query falls into its guard range. □

If this guard range assignment strategy is applied to all the data points, for every <query, k-NN> pair, the guard range of k-NN will cover the query with a confidence larger than $1-2\delta/3$. Combined with the failure probability of LSH which is smaller than $1-\delta/3$, the overall recall is larger than $1-\delta$.

In summary, for each data point o, we can select the radius r from candidate set $\{R, cR, c^2R, \dots\}$ where r is the smallest value that satisfies $D_o(o,r) \geq \mathcal{B}_k$. Then for any query, the probability its k-NN to be returned is larger than $1-\delta$. The algorithm for the overall radius selection and index process is described in Algorithm 1.

Here we give a complexity analysis for the indexing cost of Selective Hashing compared with E2LSH. E2LSH involves 3 steps: 1) Get the inner products between data points and projection vectors in hashing family (using F to denote the size of hashing family). $O(FDN)$. 2) Get the integer codes based on the product, bias and projection width. $O(FN)$. 3) Compute the buckets for points based on universal hashing. Insert the points into their corresponding buckets. $O(MLN)$. Hence, the complexity of E2LSH

[2] There is a slight difference between probability and confidence, since the parameter being estimated is not a random variable but a fixed value. The confidence $1-\delta$ here means that with probability $1-\delta$, the confidence interval based on the observation will contain the actual parameter.

Algorithm 1: INDEX

```
  Input: Dataset: List⟨Point⟩ O
  /* smallest radius is R, increment ratio is c,
     and number of indices is SH.size             */
  Output: Index: List⟨LSH⟩ SH
1 foreach p in O do
      /* observation estimation                    */
2     find r where D_o(p, r) = B_k;
      /* insert into only one index.               */
3     integer i = ⌊log_c (r/R)⌋;
4     if i < 0 then i = 0 ;
5     if i ≥ SH.size then i = SH.size − 1;
6     SH[i].insert(p);
7 return SH
```

is $O(FDN + MLN)$. Selective Hashing involves 3 major steps: 1) step 1 in E2LSH. $O(FDN)$. 2) For H radii: follow step 2 and 3 in E2LSH to build LSH, $O(MLN)$; collision counting, $O(LN)$; radius selection by comparing $LB(p,r)$ and threshold B_k, $O(N)$. $O(HMLN)$ in total. 3) follow step 2 and 3 in E2LSH to insert points into the selective hashing index. $O(MLN)$. Hence, the complexity of Selective Hashing is $O(FDN + HMLN)$. Typically, M, L and H are about 20, D is more than 100, F is around 100. Since HML is usually smaller than FD, step 1 is the dominant cost in Selective Hashing, which is consumed in all the hashing indexes. That is why Selective Hashing does not involve H times of cost compared with E2LSH.

4.2.2 Cost Analysis

We first introduce an optimal solution for k-NN using radius search, which cannot be achieved in reality. After that, we discuss the overhead of selective hashing compared to the optimal one.

Using radius search to solve a k-NN problem, the best result it can achieve is to foresee distances of k-NN and to index/query with the k_{th}-NN distance as the radius. Obviously, the k-NN distances of different queries differ greatly, and for different queries the whole dataset needs to be indexed with different radii. Thus, this approach cannot be realized in practice. However, it represents the best we can achieve with the idea of radius search algorithm for k-NN search. Its cost is just the same as conducting a simple radius search. We define this optimal solution as *optimal k-NN*. When the radius search algorithm is LSH, we call it *optimal LSH*.

Now we analyze the overhead of selective hashing compared to the optimal k-NN (e.g., optimal LSH). We start by comparing the number of returned points in a radius, and the additional cost of checking outside points will be discussed later. For optimal k-NN, the number of points in the radius is exactly k, since the radius is set as k_{th}-NN distance. For selective hashing, some more points will also be checked. The exact number equals to the count of data points whose guard ranges cover the query. According to Section 4.2.1, we know that all the reverse $\mathcal{B}_k$-NN points have a guard range which covers the query point. Thus, all the reverse $\mathcal{B}_k$-NN of this query need to be checked.

Unfortunately, we cannot tell how many reverse $\mathcal{B}_k$-NN a given query has. Intuitively, the average should be $\mathcal{B}_k$, but the actual number depends on the specific query. An adversary could deliberately choose points with large number of reverse $\mathcal{B}_k$-NN. However, if the query distribution is expected to be "well-behaved", this is an upper bound for the expectation number. Similar to data distribution previously described, we define the query density as Q_a, and assume λ'-continuity for query density, which limits the rate of change for Q_a/D_a.

DEFINITION 5. **(λ'-Continuity for Query Density)** *Using $\mathcal{S}$ to denote the whole space, D_a to denote the data distribution, and Q_a to denote the query distribution, where D_a and Q_a are the density function over the whole space, giving some area A, the expectation of the number of data points can be expressed as the integration of D_a on A: $\int_{p\in A} D_a(p)\,\mathrm{d}p$.* [3] *And it is analogous for Q_a. Using $D_a^{-1}(p,k)$ to denote the inverse function for $D_a(p,r)$, which returns a radius r such that $D_a(p,r)=k$, the λ'-continuity for query density is satisfied iff:*
$\forall p \in \mathcal{S}, \forall o \in B(p, D_a^{-1}(p,\mathcal{B}_k)),\ Q_a(o)/D_a(o) \leq \lambda Q_a(p)/D_a(p)$.

Note that we do not require the query distribution to follow the data distribution, but the ratio should be a continuous function. Under this assumption, we have the following theorem.

THEOREM 3 (EXPECTATION OF REVERSE $\mathcal{B}_k$-NN). *When the query distribution follows λ'-continuity for query density, the expectation of reverse $\mathcal{B}_k$-NN is at most $\lambda'\mathcal{B}_k$ per query.*

PROOF. For any point $p \in \mathcal{S}$ (e.g., R^d for a multi-dimensional space), its guard range is $B(p, D_a^{-1}(p,k))$, and its expectation number of reverse $\mathcal{B}_k$-NN can be defined as $E(p)$:

$$E(p) = \int_{q\in B(p,D_a^{-1}(p,k))} Q_a(q)\,\mathrm{d}q$$

Using $t(p)$ to denote the ratio $Q_a(p)/D_a(p)$, due to the λ'-continuity for query density, we have $Q_a(q) \leq \lambda' t(p) D_a(q)$ for any $q \in B(p, D_a^{-1}(p,k))$. On the other hand, $D_a(p, D_a^{-1}(p,\mathcal{B}_k))$ is just $\mathcal{B}_k$. Consequently, we have an upper bound for $E(p)$:

$$\begin{aligned} E(p) &\leq \int_{q\in B(p,D_a^{-1}(p,k))} \lambda' t(p) D_a(q)\,\mathrm{d}q \\ &= \lambda' t(p) \int_{q\in B(p,D_a^{-1}(p,k))} D_a(q)\,\mathrm{d}q \\ &= \lambda' \mathcal{B}_k t(p) \end{aligned}$$

Thus, the expectation of $<$object, reverse $\mathcal{B}_k$-NN query$>$ pairs for the whole space is $\int_{p\in\mathcal{S}} D_a(p)E(p)\,\mathrm{d}p$. And on the other hand, we know the total number of query is the integration of Q_a on $\mathcal{S}$, i.e., $\int_{p\in\mathcal{S}} Q_a(p)\,\mathrm{d}p$. Therefore, the upper bound of the expectation of reverse $\mathcal{B}_k$-NN per query is:

$$\begin{aligned} \frac{\int_{p\in\mathcal{S}} D_a(p)E(p)\,\mathrm{d}p}{\int_{p\in\mathcal{S}} Q_a(p)\,\mathrm{d}p} &\leq \frac{\int_{p\in\mathcal{S}} D_a(p)\lambda'\mathcal{B}_k t(p)\,\mathrm{d}p}{\int_{p\in\mathcal{S}} Q_a(p)\,\mathrm{d}p} \\ &= \lambda'\mathcal{B}_k \frac{\int_{p\in\mathcal{S}} D_a(p)\frac{Q_a(p)}{D_a(p)}\,\mathrm{d}p}{\int_{p\in\mathcal{S}} Q_a(p)\,\mathrm{d}p} \\ &= \lambda'\mathcal{B}_k \quad \square \end{aligned}$$

Based on the above analysis, the search range of selective hashing is smaller than the distance of $\lambda'\mathcal{B}_k$-NN of the query. Thus, the check rate is smaller than the cost of searching for $\lambda'\mathcal{B}_{kth}$-NN distance in radius search. In general, the cost of searching $\lambda'\mathcal{B}_k$-NN will be smaller than $\lambda'\mathcal{B}_k/k$ times the cost of searching k-NN, since we can always search k objects each time and repeat $\lambda'\mathcal{B}_k/k$ times, in the absence of ties. However, there exist some query cases that other nearest neighbor points are much more difficult to find than the k-NN (e.g., k points collide with the query, while all other points in the dataset have almost the same distance to the query point).

Recall that the LSH in selective hashing requires a recall of $1-\delta/3$, and it will lead to additional $(\log\delta - \log 3)/\log\delta$ times overhead. When k increases, $\lambda'\mathcal{B}_k/k$ decreases, and when δ increases, $(\log\delta - \log 3)/\log\delta$ also decreases. Thus, $\lambda'\mathcal{B}_k/k$ and $(\log\delta - \log 3)/\log\delta$ can be bounded by some constants.

[3] This definition is also consistent with the previous one $D_a(p,r)$, which can be represented as $\int_{p\in B(p,r)} D_a(p)\,\mathrm{d}p$.

Algorithm 2: QUERY

Input: Query: Point q, Index: List⟨LSH⟩ SH
Output: KNN result: List⟨Point⟩ $KNN_{current}$
1 $KNN_{current} = \{\}$;
2 List⟨Point⟩ pts;
/* search indices begining from the smallest radius one */
3 **for** i = *0 to* SH*.size-1* **do**
4 pts = $SH[i]$.search(q);
5 **foreach** p *in* pts **do**
6 KNN.checkAndUpdate($KNN_{current}, p$);
/* density-based pruning */
7 $dis = \max_{p\in KNN_{current}} \|p,q\|$;
8 **if** $dis < \mu \cdot SH[i].radius$ **then** break
9 **return** $KNN_{current}$

THEOREM 4 (EFFECTIVENESS OF SELECTIVE HASHING). *Selective hashing achieves the same quality for k-NN search as the fixed radius LSH search, with radius equals the distance to the $c_1 k_{th}$-NN, with as most c_2 times overhead, where $c_1 = \lambda'\mathcal{B}_k/k$ and $c_2 = (\log\delta - \log 3)/\log\delta$. c_1 and c_2 are bounded by constant.*

Theorem 4 is just a theoretical bound with several relaxations made during its derivation. For typical datasets and query requirements, c_1 is bounded by 4-5 and c_2 is around 1.3. The real performance is much better as we will see in the experimental study.

4.3 Density-based Pruning

In the query scheme discussed above, the query is pushed to all the index instances. In this subsection, we discuss how to further reduce the check rate by not checking some index instances while still obtaining correct results.

We observe that indices with larger radii have the highest check rate. Intuitively, the guard ranges for points in those indices are very large, potentially including many irrelevant areas (e.g., a small dense area nearby). In our experiments on real datasets, we found that points in the top 20% indices with the largest radius typically have 2-5x the average check rate. Thus, we would like to avoid checking those indices if at all possible.

In selective hashing, usually none of the indices can be skipped, as every point only exists in one index. However, if a query has already found enough points within a very small radius, then all its k-NN should fall in a high density area and therefore would not have been placed in a large radius index. Thus, we can search the applicable indices from the smallest radius to the largest radius one by one, and stop searching when all the remaining indices will not contain any potential k-NN. The following theorem gives a proof for the correctness of this pruning.

THEOREM 5 (CORRECTNESS OF DENSITY-BASED PRUNING). *For a query q, if $D_o(q,r) \geq k$ and $D_o(q,r') \geq \mathcal{B}_k$, all the indices with a radius larger than $c(r+r')$ will not contain k-NN of q.*

PROOF. Let $kNN(q)$ to be the k-NN points of a query q. Since $D_o(q,r) \geq k$, $\forall o \in kNN(q)$, $||o,q|| \leq r$. Due to the triangle inequality, $\forall p \in B(q,r')$, $||o,p|| \leq r+r'$. For this reason, we have: $D_o(o, r+r') \geq D_o(q,r') \geq \mathcal{B}_k$.

On the other hand, the indexing algorithm will select the smallest radius r that satisfies $D_o(o,r) \geq \mathcal{B}_k$. And the radius increases c times each time. Thus, there will be a radius $R \in [r+r', c(r+r'))$

used in our selective hashing index. $\forall o \in kNN(q)$, o should be indexed in a radius equal to or smaller than $R < c(r + r')$. $\square$

Using this pruning, r is the bound of k-NN and is naturally updated all the time. We may build another heap to maintain the $\mathcal{B}_{kth}$-NN distance r', or simply use a ratio μr to approximate $c(r + r')$ for better performance. Algorithm 2 outlines our querying strategy.

4.4 Incremental Maintenance

Efficient update is an essential requirement for dynamic index structures. Selective hashing can support efficient updates as well. Inserting or removing an individual point from a hash table is straightforward. The challenge is that updates can change the density of points in some regions, and therefore impact the choices made in constructing the index.

In the framework of selective hashing, the update process not only has to consider the insertion or deletion for the point to be processed, but also the impact on nearby points. For example, if there are many insertions in a sparse area, then the area becomes denser and all the points in this area should be moved to a smaller radius in order to reduce the check rate. Therefore, we need to make sure that the radius selection is based on the correct density observation. Thus, we need to maintain the density information for each data point, and re-select the radius when necessary.

First, we observe that the radius selection for each point is performed independently. That is, the choice of radius depends only on the existence of nearby points and not on the radius choice for these points. Therefore, when an update is applied, we only need to re-compute the density observation of points that may be affected. The number of affected points depends on the applied algorithm that estimates the density observation. For example, if it is estimated based on collisions in LSH, all the points that collide with the updated point should update their count of collisions and move to a new radius R where R is the smallest one satisfying $LB(p, R) \geq \mathcal{B}_k$. The number of points being affected is around $\mathcal{B}_k + P_b * N$ and such cost will not be high in most cases.

In write-intensive applications such as social networks and sensory observations, data points arrive as a stream. Most updates are new insertions and sub-optimal query performance may be acceptable in return for faster updates. To address this scenario, we propose a lazy updating strategy to postpone the radius re-selection to the moment when the check rate for an object becomes unacceptably high. For new insertions, the radius is only reduced and never increased. Thus, using the original radius will only affect the check rate, while the recall is still guaranteed. Instead of monitoring the density observation, lazy updating maintains the check rate information for each point, re-computes its density observation and re-selects its radius when its check rate is higher than some threshold. When a new point comes, we simply insert it into selective hashing index. Such updating strategy slight increases the query time while achieving an update speed as fast as the naive LSH.

5. EXPERIMENTS

5.1 Experimental Setup

Datasets. We evaluate the performance using two real datasets: an image dataset *Flickr1M* and an audio recording dataset *MillionSong*, both of them represented as high-dimensional feature vectors. Flickr1M [22] contains extracted features of 1 million images from Flickr. Each image is represented as a 3,857-dimensional feature vector comprising a concatenation of SIFT feature, color histogram, etc. MillionSong [4] is a collection of audio features and meta data for a million contemporary popular music tracks. We split the audio into sections such that each contains 50 continuous small voice segments. Each segment is described using 12 MFCC features. We get 2 million sections and each section is a 600-dimensional vector. For whitening, normalization and dimension reduction purposes, following common practice, we use PCA to pre-process these two datasets and keep 200 and 100 dimensions respectively.

Methods and implementations. We have implemented the following methods for comparison.

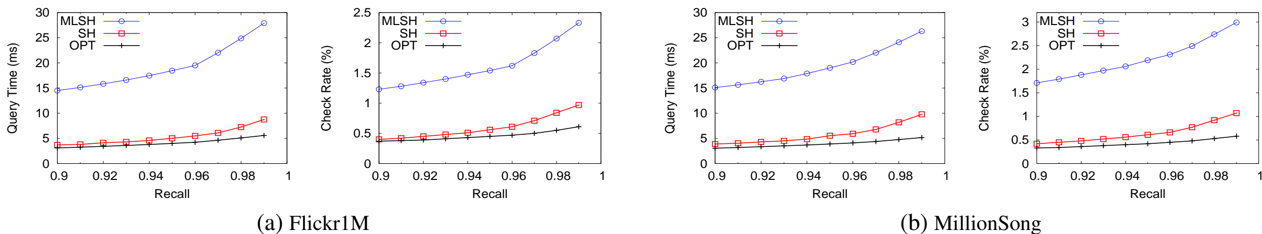
- *Selective hashing* **(SH)**. Our algorithm presented in Section 4. The default setting is with density-based pruning and normal updating.
- *Fixed radius LSH*. E2LSH [7] is the most common LSH implementation for exact Euclidean search. For each dataset, one small radius is selected for a more efficient solution *small radius LSH* **(SLSH)**, and one larger radius is selected for a more accurate solution *large radius LSH* **(LLSH)**. In general, the small radius one is more effective when recall is low. However, its recall cannot be further improved as there are many k-NN outside its radius. On the other hand, the larger radius one can offer higher recall, but is more likely to have too many collisions and hence requires a higher check rate.
- *Multi-radius LSH* **(MLSH)**. We build $|\mathcal{G}|$ E2LSHs using radii $\{R, cR, c^2R, \dots\}$. When a query is issued, we search the indices from the smallest radius E2LSH to those with larger radii. When there are at least k points in current radius R, all the k-NN are within radius R; the recall guarantee holds and search procedure is stopped. Its index size is $|\mathcal{G}|$ times that of the fixed radius LSH. MLSH provides a strict theoretical guarantee for the recall.
- *Optimal LSH* **(OPT)**. We have presented an *Optimal LSH* in Section 4.2.2. It gives a lower bound for the cost of k-NN search. We test its performance using multi-radius LSH, while only searching the LSH which has the same radius as the distance of k_{th}-NN (pre-calculated and given as input).
- *IsoHashing* **(ISO)**. ISO [15] is a learning-based hashing method trying to give equal variance for all dimensions where quantizers are applied.
- *Data sensitive hashing* **(DSH)**. DSH [9] is a learning-based high-dimensional data-sensitive hashing which is directly designed for k-NN search and especially targeted to give high recall guarantee.

For all these hash-based indices, we follow the typical index and query strategy used in E2LSH which is described in Section 2. For DSH and ISO, all the hashing functions are used to build the hashing family. In table construction, universal hashing is applied to all the hash tables and the total number of buckets per table is 9973 (the closest prime to 10000). For SH, MLSH and OPT, the number of radii $|\mathcal{G}|$ is set to 20, and the approximation ratio c is set to 1.2. The largest radius can be 30x larger than the smallest one. Finally, the memory is large enough for all the methods to keep the original dataset and all the indices in memory. Parameters for all methods are optimized to offer the best performance. All the experiments were performed on a Ubuntu system equipped with one Intel 3.4GHz processor, 16GB of memory.

Measurements. We evaluate the performance using the following measures: *Recall* is used to measure the quality of query result. *Check Rate* and *Query Time* are used to measure the efficiency of algorithm. Precision or MAP is not used as measurement since all the algorithms are used to generate a set of candidates. The exact distances are calculated for the candidates, and the top-k are used as k-NN result. As an example, to solve a 20-NN search problem in 1M points, we expect the algorithm to return 2000 points that contain all the k-NN (1% precision 100% recall), but not to return 20 points that contain 12 of them (60% precision & recall). We can

Table 1: Overall Comparison

method		(a) Flickr1M OPT	SH	MLSH	SLSH	LLSH	DSH	ISO	(b) MillionSong OPT	SH	MLSH	SLSH	LLSH	DSH	ISO
query time(ms)	recall = 0.99	5.5	8.8	27.9	/	/	/	/	5.3	9.8	26.3	/	/	34.8	/
	recall = 0.96	4.2	5.5	19.5	/	52.3	14.5	28.5	4.1	5.9	20.2	/	47.6	13.7	21.5
	recall = 0.90	3.1	3.7	14.5	11.4	21.5	7.9	6.9	3.1	3.9	15.1	13.1	24.4	6.8	5.75
check rate(%)	recall = 0.99	0.61	0.97	2.33	/	/	/	/	0.57	1.08	2.99	/	/	3.62	/
	recall = 0.96	0.47	0.61	1.62	/	5.82	1.45	3.15	0.44	0.66	2.31	/	5.28	1.43	2.24
	recall = 0.90	0.37	0.40	1.23	1.26	2.35	0.88	0.79	0.33	0.42	1.71	1.46	2.71	0.71	0.64
index size(M)	recall = 0.99		283	3485	/	/	/	/		148	1770	/	/	347	/
	recall = 0.96		170	1765	/	405	349	520		101	832	/	207	137	160
	recall = 0.90		105	827	264	124	152	145		66	441	113	70	90	80



(a) Flickr1M (b) MillionSong

Figure 2: Comparison of Methods with Strict Query Guarantee

validate those 2000 points and obtain the exact answer. However, we cannot expand the 12 points to obtain a higher recall. For this reason, we only focus on the query performance under high recall (90% - 99%) requirements. We also compare *Index Size*, *Construction Time* and *Update Cost*. 1000 points are randomly removed from each dataset and used as query points. Unless specified otherwise, k is set to 20 except in the experiment comparing the performance of different methods under various k.

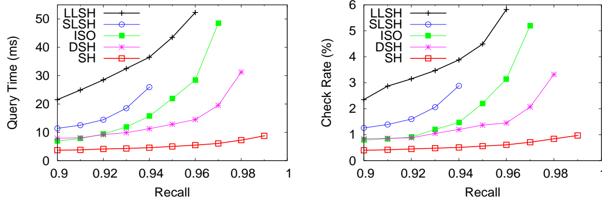
5.2 Query Performance

Query performance is the principal criterion for indexing. Table 1 summarizes the performance of the methods that are required to reach a specified recall on the two datasets. Only SH and MLSH (and the unrealizable OPT) can reach 99% recall on both datasets, and can be considered as having strict quality guarantee. Although DSH ensures the recall of the results on its training sets and performs much better than the other methods (ISO, SLSH and LLSH), there are still more than 1% missing results. In what follows, we first present the comparison results for the methods with strict quality guarantee, and then show the comparison results for ad-hoc and learning-based methods which are faster but achieve lower quality. After that, we study how different values of k will affect the performance for those methods, and also the effect of density-based pruning on SH.

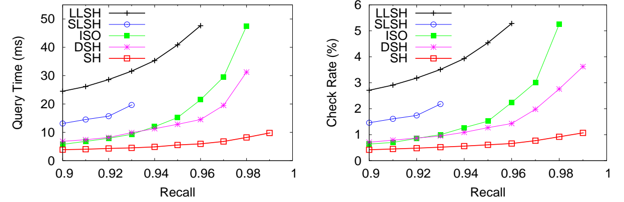
Methods with strict quality guarantee. We first present comparison result for SH, MLSH and OPT. Figure 2 shows their performance on check rate and query time to reach a certain recall. The trends in two datasets are similar. Furthermore, the trends of check rate are consistent with those of query time since most query time is spent in computing the exact distances for the candidates. Specifically, all the algorithms take about 9ms to scan 1% of data (about 8MB data). From the figure, we can see that SH almost reaches the lower bound — OPT, when the recall is around 90%. When recall increases, δ becomes smaller and $\Phi(1-\delta/3)$ will be slightly larger. For this reason, SH uses 1.5x-2x time than OPT when recall is 99%. However, the effect of δ is bounded, since Gaussian CDF decays fast to zero in its tail. Compared to MLSH, both query time and check rate of SH are only about one third. The query time of MLSH is about 5x larger than OPT. From the result we can see, even if multiple indices for different radii have been built, the overhead of attempting a suitable radius is still very high. If the distance of k-NN varies W times, the overhead of query time can only be bounded by $O(\log_c W)$, and index size is also $O(\log_c W)$ times the fixed radius one. Moreover, SH also benefits from density-based pruning, while MLSH does not.

Ad-hoc and learning-based methods. From Table 1, it is clear that the index size of MLSH is huge and may not be acceptable in lots of real scenarios. Here we give a comparison for the methods using only one group of hash tables, including SLSH, LLSH, ISO, DSH and SH. Figure 3 shows their check rate and query time as a function of recall. SH performs more than 1.75x faster than DSH. This is because although DSH also aims to offer a better structure to maintain the k-NN relations, it is based on the optimization of global hash functions, whereas SH offers a more fine-grained local optimization for every data point. LLSH and SLSH give the worst performance, due to the fact that they do not have any optimization on k-NN at all. Since SLSH cannot provide high recall, say 96%, we shall use LLSH as the representative of the fixed radius LSH in subsequent comparisons.

Sensitivity to k. In most real-life applications (e.g., displaying the first page of results, k-NN joins, k-NN classifier), k is typically pre-defined throughout all the queries. However, it is important for an index to be insensitive to the value of k so that we only need to maintain one k-NN index to serve most potential applications. Here we study the query performance when k of the query varies while keeping $k = 20$ in index construction. Figure 4 shows the query time for achieving 96% recall for queries with different values of k. Obviously, the query with a larger k is harder to compute and needs longer query time. MLSH is the only method that does not use k for index construction and hence is not sensitive to k. For LSH-based methods SH and LLSH, their relative query time performance compared with MLSH increase around 25% for $k = 1$, 20% for $k = 100$ and 90% for $k = 500$. This increase can be viewed as the overhead due to using an incorrect k during index construction. For all the cases, SH is 2-5x faster than all the other methods. This is because the segment width of LSH may be effective for an interval of radius like $[r, 1.2r]$, with the performance dropping significantly outside the interval [7], while the distances of 100_{th}-NN (or 1-NN) and 20_{th}-NN usually do not have a significant difference. Therefore a single index (e.g., $k = 20$) can serve a moderate range of k values (e.g., 1 to 100), which serves the need of most applications and searches.



(a) Flickr1M



(b) MillionSong

Figure 3: Comparison of Ad-hoc and Learning-based Methods

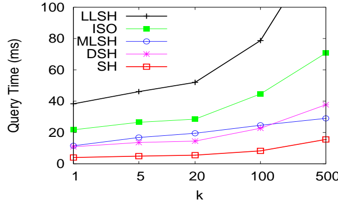
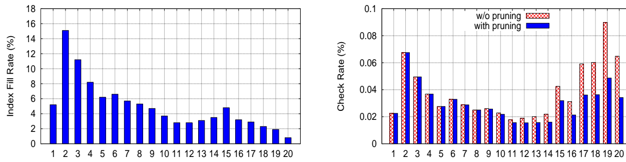


Figure 4: Sensitivity to k



(a) Data Distribution among Indices (b) Pruning Effect on Indices

Figure 5: Density-based Pruning

Density-based pruning. Here we study the effect of density-based pruning on selective hashing. The distribution of data among different radii is more balanced in the Flickr1M dataset and therefore we use it to show how the check rate is changed. The distribution of data among all the 20 indices is shown in Figure 5a, where the first bar represents the index with the smallest radius. Figure 5b reports the check rate to reach 96% recall before and after the density-based pruning is adopted. Although the proportion of points in 4 indices with largest radii are only 10%, they contribute about 40% of the check rate. By using the density pruning strategy, their check rate is reduced by almost a half.

5.3 Index Construction

The index size of each method has already been described in Table 1, and SH requires the least storage space to reach a given recall. Compared with MLSH, the index size of SH is only 7% - 10%. Compared with the other methods, the hash tables of SH are also the most space efficient. It will be fairer to compare the index size, construction time and update cost of each method to reach a given recall. For this reason, the cost of the fixed radius LSH can be higher than the other methods where fewer tables are constructed.

Efficient construction is also an essential requirement for indexing, since periodical index reconstruction is commonly adopted as a way to improve the query efficiency. Table 2 summarizes the index construction time for all the methods. We also conduct a scalability test where all the data points are duplicated 10 times. All the methods take about 9x-10x longer time in the 10x dataset, suggesting that the construction time is linear with respect to the size of the dataset. SH and LLSH incur the least time to construct the tables, followed by ISO and DSH. The construction time typically consists of four components: training hashing functions (or estimating density for SH); computing the inner product between data points and projection vectors; getting the key for each hash table; and inserting points into hash tables. Most approaches have the same inner product computation step and have the same cost if their hashing families have the same size. The major differences of construction time happen in the last two components, i.e., table construction. For SH, its hash tables are very efficient, thus fewer tables are built and the cost of insertions is therefore lower.

Table 2: Index Construction Time(s)

	Flickr1M			MillionSong		
Methods	#table	1x	10x	#table	1x	10x
SH	15	112	1020	17	86	800
MLSH	11	205	1922	10	143	1346
LLSH	40	112	1098	44	85	833
DSH	34	135	1215	26	122	1188
ISO	30	120	1140	24	90	880

Table 3: Update Cost

	Flickr1M		MillionSong	
Methods	insertion	throughput	insertion	throughput
SH	196	2.14	301	2.79
SH-Lazy	97	4.33	165	5.09
MLSH	236	1.78	329	2.55
LLSH	146	2.88	221	3.80
DSH	310	1.36	448	1.86

5.4 Update

We also evaluate the update cost, measured by the average processing time per thousand insertions, and the throughput, measured by the amount (MB) of data processed per second. Table 3 summarizes the results. Lazy updating version of SH, named SH-Lazy, is the fastest, reaching a throughput of 5MB per second which is usually sufficient to handle sensor data. This is because the update cost per table for lazy updating is very close to LLSH, while fewer tables are maintained to get the same quality of results. The update for DSH is relatively slow since it involves re-training a fraction of hashing functions. ISO does not support update, limiting its applicability as an indexing structure for dynamic databases.

We also study the effect of lazy updating in different scenarios. Table 4 shows the performance under different update/query ratios. In this experiment, the query and insertion of points are processed at the same time. Half of the points are selected as original points in the dataset and the other half are used as points for updating. Thus, the average density changes 2x during update when all the update points are processed. For lazy updating, the query performance degrades about 5% to 30%, whereas the update cost is reduced by half. Interestingly, query performance converges to the density monitoring approach when more queries are issued. This is because more queries may lead to more frequent updates, and therefore the estimation is more accurate. In general, when the update/query ratio is larger than 100, lazy updating is definitely a better choice.

6. RELATED WORK

There is extensive work on high-dimensional indices for k-NN queries, described in good literature surveys [5, 11]. In the high-

Table 4: Lazy Updating

Ratio	Method	Update(ms)	Query(ms)	Total(ms)
1:1	SH	3.0	97.7	100.7
	SH-Lazy	1.7	104.2	105.9
1:10	SH	30.2	97.7	127.9
	SH-Lazy	16.8	112.8	129.6
1:100	SH	301.6	97.7	399.3
	SH-Lazy	166.3	119.1	285.4
1:1000	SH	3015.3	97.7	3113.0
	SH-Lazy	1650.7	126.3	1777.0

dimensional scenario, most tree-based solutions suffer from the curse of dimensionality [27]. Locality sensitive hashing (LSH) methods [10, 7] are the best known approaches for approximate high-dimensional nearest neighbor search. Recent result [1] about the complexity of LSH has almost reaches the lower bound for radius search [18]. However, as illustrated earlier, there remains a huge gap between effective k-NN search and effective radius search. C2LSH [8] was proposed as alternatives of LSH for effective radius search, which consume significantly less index storage. Fast LSH [6] was proposed to accelerate the computation of Euclidean distance between high-dimensional vectors. These methods are *orthogonal* to our proposed SH, which is designed as an effective k-NN search mechanism on the top of them.

Recently, there has been increasing attention focusing on data-sensitive hashing structures to solve the k-NN search problems. The LSB-tree [24] leverages the intrinsic features of the B-tree to provide a data-sensitive solution. Spectral hashing [29, 28], data sensitive hashing [9], Boostmap [2] and other learning-based methods [16, 12, 19, 25] aim to optimize the hash functions. Although these learning-based methods are effective to capture most *global* distribution trends, they fail to catch specific *local* patterns such as dense groups or sparse hulls. In contrast, SH is designed to give a local optimal structure for each data point individually. There are also many methods [17, 20, 13] focusing on exploiting more buckets in a table to search far away k-NN. However, radius is the decisive parameter in LSH, and it remains hard to retrieve far away points.

SH is also inspired by the idea of partial indexing [23] in Postgres. Both partial indexing and SH consider the effectiveness of the index for each data object. However, there are major differences between the two concepts, especially in query processing. In partial indexing, the index method chosen for use is still based on the query. Thus, its starting point is to build a set of small but effective partial indices to facilitate a wider range of query tasks. However, in selective hashing, the index to be accessed for a certain data object depends on itself, but not the query. SH is also related to some density estimation topics [3, 21, 26]. In SH, we follow some common assumptions in density estimation, but adapt the definition of density to serve our analysis. Moreover, our focus is on the data observation, while most density-related works aim to discover a smooth function behind the observation.

7. CONCLUSION

The LSH family of algorithms is considered to be very good at similarity search in high-dimensional space. However, it has been designed to address the problem of fixed radius search, rather than the k-NN problem. In this paper, we introduced Selective Hashing (SH) as a meta-technique for k-NN search, constructed on top of any fixed radius search techniques, such as LSH. The key innovation in SH is the ability of choosing the index independently for each point, and then consulting multiple disjoint indexes for each query. The effectiveness of the proposed technique was established theoretically through careful analysis and demonstrated experimentally through performance studies using real datasets.

Acknowledgments

This work was supported by the National Research Foundation, Prime Minister's Office, Singapore under Grant No. NRF-CRP8-2011-08. Jagadish was partially supported by NSF grant IIS 1250880.

8. REFERENCES

[1] A. Andoni, P. Indyk, H. L. Nguyen, and I. Razenshteyn. Beyond locality-sensitive hashing. In *SODA*, 2014.
[2] V. Athitsos, J. Alon, S. Sclaroff, and G. Kollios. Boostmap: A method for efficient approximate similarity rankings. In *CVPR*, 2004.
[3] K. P. Bennett, U. Fayyad, and D. Geiger. Density-based indexing for approximate nearest-neighbor queries. In *SIGKDD*, 1999.
[4] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere. The million song dataset. In *ISMIR*, 2011.
[5] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 2001.
[6] A. Dasgupta, R. Kumar, and T. Sarlós. Fast locality-sensitive hashing. In *SIGKDD*, 2011.
[7] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*, 2004.
[8] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*, 2012.
[9] J. Gao, H. V. Jagadish, W. Lu, and B. C. Ooi. Dsh: data sensitive hashing for high-dimensional k-nnsearch. In *SIGMOD*, 2014.
[10] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.
[11] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces (survey article). *TODS*, 2003.
[12] Y. Hwang, B. Han, and H.-K. Ahn. A fast nearest neighbor search algorithm by nonlinear embedding. In *CVPR*, 2012.
[13] H. Jégou, L. Amsaleg, C. Schmid, and P. Gros. Query adaptative locality sensitive hashing. In *ICASSP*, 2008.
[14] J. F. C. Kingman. *Poisson processes*, volume 3. Oxford university press, 1992.
[15] W. Kong and W.-J. Li. Isotropic hashing. In *NIPS*, 2012.
[16] Y. Lin, R. Jin, D. Cai, S. Yan, and X. Li. Compressed hashing. In *CVPR*, 2013.
[17] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *VLDB*, 2007.
[18] R. Motwani, A. Naor, and R. Panigrahy. Lower bounds on locality sensitive hashing. *Discrete Mathematics*, 2007.
[19] Y. Mu, J. Shen, and S. Yan. Weakly-supervised hashing in kernel space. In *CVPR*, 2010.
[20] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *SODA*, 2006.
[21] D. W. Scott. *Multivariate density estimation: theory, practice, and visualization*, volume 383. John Wiley & Sons, 2009.
[22] N. Srivastava and R. Salakhutdinov. Multimodal learning with deep boltzmann machines. In *NIPS*, 2012.
[23] M. Stonebraker. The case for partial indexes. *SIGMOD Record*, 1989.
[24] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*, 2009.
[25] J. Wang, S. Kumar, and S.-F. Chang. Semi-supervised hashing for large-scale search. *TPAMI*, 2012.
[26] Q. Wang, S. R. Kulkarni, and S. Verdú. Divergence estimation for multidimensional densities via-nearest-neighbor distances. *Information Theory*, 2009.
[27] 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.
[28] Y. Weiss, R. Fergus, and A. Torralba. Multidimensional spectral hashing. In *ECCV*. 2012.
[29] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *NIPS*, 2008.