Quality and Efficiency in High Dimensional Nearest Neighbor Search

Yufei Tao[1] Ke Yi[2] Cheng Sheng[1] Panos Kalnis[3]

[1]Chinese University of Hong Kong
Sha Tin, New Territories, Hong Kong
{taoyf, csheng}@cse.cuhk.edu.hk

[2]Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
yike@cse.ust.hk

[3]King Abdullah University of Science and Technology
Saudi Arabia
panos.kalnis@kaust.edu.sa

ABSTRACT

Nearest neighbor (NN) search in high dimensional space is an important problem in many applications. Ideally, a practical solution (i) should be implementable in a relational database, and (ii) its query cost should grow *sub-linearly* with the dataset size, regardless of the data and query distributions. Despite the bulk of NN literature, no solution fulfills both requirements, except *locality sensitive hashing* (LSH). The existing LSH implementations are either rigorous or adhoc. *Rigorous-LSH* ensures good quality of query results, but requires expensive space and query cost. Although *adhoc-LSH* is more efficient, it abandons quality control, i.e., the neighbor it outputs can be *arbitrarily* bad. As a result, currently no method is able to ensure both quality and efficiency simultaneously in practice.

Motivated by this, we propose a new access method called the *locality sensitive B-tree* (LSB-tree) that enables fast high-dimensional NN search with excellent quality. The combination of several LSB-trees leads to a structure called the *LSB-forest* that ensures the same result quality as *rigorous-LSH*, but reduces its space and query cost dramatically. The LSB-forest also outperforms *adhoc-LSH*, even though the latter has no quality guarantee. Besides its appealing theoretical properties, the LSB-tree itself also serves as an effective index that consumes linear space, and supports efficient updates. Our extensive experiments confirm that the LSB-tree is faster than (i) the state of the art of exact NN search by *two orders of magnitude*, and (ii) the best (linear-space) method of approximate retrieval by *an order of magnitude*, and at the same time, returns neighbors with much better quality.

ACM Categories and Subject Descriptors

H.3.1 [Content Analysis and Indexing]: Indexing Methods.

General Terms

Algorithms, Theory.

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.
Copyright 2009 ACM 978-1-60558-551-2/09/06 ...$5.00.

Keywords

Nearest Neighbor Search, Locality Sensitive Hashing.

1. INTRODUCTION

Nearest neighbor (NN) search is a classic problem with tremendous impacts on artificial intelligence, pattern recognition, information retrieval, and so on. Let $\mathcal{D}$ be a set of points in d-dimensional space. Given a query point q, its NN is the point $o^* \in \mathcal{D}$ closest to q. Formally, there is no other point $o \in \mathcal{D}$ satisfying $\|o, q\| < \|o^*, q\|$, where $\|, \|$ denotes the distance of two points.

In this paper, we consider *high-dimensional* NN search. Some studies argue [9] that high-dimensional NN queries may not be meaningful. On the other hand, there is also evidence [6] that such an argument is based on restrictive assumptions. Intuitively, a meaningful query is one where the query point q is much closer to its NN than to most data points. This is true in many applications involving high-dimensional data, as supported by a large body of recent works [1, 3, 14, 15, 16, 18, 21, 22, 23, 25, 26, 31, 33, 34].

Sequential scan trivially solves a NN query by examining the entire dataset $\mathcal{D}$, but its cost grows linearly with the cardinality of $\mathcal{D}$. Ideally, a practical solution should satisfy two requirements: (i) it can be implemented in a relational database, and (ii) its query cost should increase *sub-linearly* with the cardinality for *all* data and query distributions. Despite the bulk of NN literature (see Section 7), with a single exception to be explained shortly, all the existing solutions violate at least one of the above requirements. Specifically, the majority of them (e.g., those based on new indexes [2, 22, 23, 25, 32]) demand non-relational features, and thus cannot be incorporated in a commercial system. There also exist relational solutions (such as *iDistance* [27] and *MedRank* [16]), which are experimentally shown to perform well for some datasets and queries. The drawback of these solutions is that they may incur expensive query cost on other datasets.

Locality sensitive hashing (LSH) is the only known solution that fulfills both requirements (i) and (ii). It supports c*-approximate NN search*. Formally, a point o is a c-approximate NN of q if its distance to q is at most c times the distance from q to its exact NN o^*, namely, $\|o, q\| \le c\|o^*, q\|$, where $c \ge 1$ is the *approximation ratio*. It is widely recognized that approximate NNs already fulfill the needs of many applications [1, 2, 3, 15, 18, 21, 23, 25, 26, 30, 31, 33, 34]. LSH is originally proposed as a theoretical method [26] with attractive asymptotical space and query performance. As elaborated in Section 3, its practical implementation can be either rigorous or adhoc. Specifically, *rigorous-LSH* ensures good quality of query results, but requires expensive space and query cost. Although *adhoc-LSH* is more efficient, it abandons quality control, i.e., the neighbor it outputs can be *arbitrarily* bad. In other words, no LSH implementation is able to ensure both quality and efficiency simultaneously, which is a serious problem severely limiting the applicability of LSH.

Motivated by this, we propose an access method called *locality sensitive B-tree* (LSB-tree) that enables fast high-dimensional NN search with excellent quality. The combination of several LSB-trees leads to a structure called the *LSB-forest* that combines the advantages of both *rigorous-* and *adhoc-LSH*, without sharing their shortcomings. Specifically, the LSB-forest has the following features. First, its space consumption is the same as *adhoc-LSH*, and significantly lower than *rigorous-LSH*, typically by a factor over an order of magnitude. Second, it retains the approximation guarantee of *rigorous-LSH* (recall that *adhoc-LSH* has no such guarantee). Third, its query cost is substantially lower than *adhoc-LSH*, and as an immediate corollary, sub-linear to the dataset size. Finally, the LSB-forest adopts purely relational technology, and hence, can be easily incorporated in a commercial system.

All LSH implementations require duplicating the database multiple times, and therefore, entail large space consumption and update overhead. Many applications prefer an index that consumes only linear space, and supports insertions/deletions efficiently. The LSB-tree itself meets all these requirements, by storing every data point once in a conventional B-tree. Based on real datasets, we experimentally compare the LSB-tree to the best existing (linear-space) methods *iDistance* [27] and *MedRank* [16] for exact and approximate NN search, respectively. Our results reveal that the LSB-tree outperforms *iDistance* by *two orders of magnitude*, well confirming the advantage of approximate retrieval. Compared to *MedRank*, our technique is consistently superior in both query efficiency and result quality. Specifically, the LSB-tree is faster by *an order of magnitude*, and at the same time, returns neighbors with much better quality.

The rest of the paper is organized as follows. Section 2 presents the problem settings and our objectives. Section 3 points out the defects of the existing LSH-based techniques. Section 4 explains the construction and query algorithms of the LSB-tree, and Section 5 establishes its performance guarantees. Section 6 extends the LSB-tree to provide additional tradeoffs between space/query cost and the quality of query results. Section 7 reviews the previous work on nearest neighbor search. Section 8 contains an extensive experimental evaluation. Finally, Section 9 concludes the paper with a summary of our findings.

2. PROBLEM SETTINGS

Without loss of generality, we assume that each dimension has a range $[0, t]$, where t is an integer. Following the LSH literature [15, 21, 26], in analyzing the quality of query results, we assume that all coordinates are integers, so that we can put a lower bound of 1 on the distance between two different points. In fact, this is not a harsh assumption because, with proper scaling, we can convert the real numbers in most applications to integers. In any case, this assumption is needed only in theoretical analysis; neither the proposed structure nor our query algorithms rely on it.

We consider that distances are measured by ℓ_p norm, which has extensive applications in machine learning, physics, statistics, finance, and many other disciplines. Moreover, as ℓ_p norm generalizes or approximates several other metrics, our technique is directly applicable to those metrics as well. For example, in case all dimensions are binary (i.e., having only $t = 2$ distinct values), ℓ_1 norm is exactly Hamming distance, which is widely employed in text retrieval, time-series databases, etc. Hence, our technique can be immediately applied in those applications, too.

We study c-approximate NN queries, where c is a positive integer. As mentioned in Section 1, given a point q, such a query returns a point o in the dataset $\mathcal{D}$, such that the distance $\|o, q\|$ between o and q is at most c times the distance between q and its real NN o^*. We assume that q is not in $\mathcal{D}$. Otherwise, the NN problem becomes a lookup query, which can be easily solved by standard hashing.

We consider that the dataset $\mathcal{D}$ resides in external memory where each page has B words. Furthermore, we follow the convention that every integer is represented with one word. Since a point has d coordinates, the entire $\mathcal{D}$ occupies totally dn/B pages, where n is the cardinality of $\mathcal{D}$. In other words, all algorithms, which do not have provable sub-linear cost growth with n, incur I/O complexity $\Omega(dn/B)$. We aim at designing a *relational* solution beating this complexity.

Finally, to simplify the resulting bounds, we assume that the dimensionality d is at least $\log(n/B)$ (all the logarithms, unless explicitly stated, have base 2). This is reasonable because, for practical values of n and B, $\log(n/B)$ seldom exceeds 20, whereas $d = 20$ is barely “high-dimensional".

3. THE PRELIMINARIES

Our solutions leverage LSH as the building brick. In Sections 3.1 and 3.2, we discuss the drawbacks of the existing LSH implementations, and further motivate our methods. In Section 3.3, we present the technical details of LSH that are necessary for our discussion.

3.1 Rigorous-LSH and ball cover

As a matter of fact, LSH does not solve c-approximate NN queries directly. Instead, it is designed [26] for a different problem called *c-approximate ball cover* (BC). Let $\mathcal{D}$ be a set of points in d-dimensional space. Denote by $B(q, r)$ a ball that centers at the query point q and has radius r. A c-approximate BC query returns the following results:

(1) If $B(q, r)$ covers at least one point in $\mathcal{D}$, return a point whose distance to q is at most cr.

(2) If $B(q, cr)$ covers no point in $\mathcal{D}$, return nothing.

(3) Otherwise, the result is undefined.

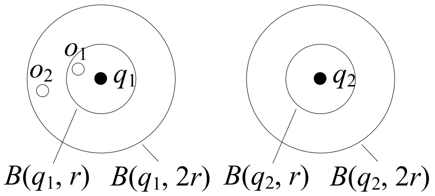


Figure 1: Illustration of ball cover queries

Figure 1 shows an example where $\mathcal{D}$ has two points o_1 and o_2. Consider first the 2-approximate BC query q_1 (the left black point). The two circles centering at q_1 represent balls $B(q_1, r)$ and $B(q_1, 2r)$ respectively. Since $B(q_1, r)$ covers a data point o_1, the query will have to return a point, but it can be either o_1 or o_2, as both of them fall in $B(q_1, 2r)$. Now, consider the 2-approximate BC query q_2. Since $B(q_2, 2r)$ does not cover any data point, the query must return empty.

Interestingly, an approximate NN query can be reduced to a number of approximate BC queries with different radii r [23, 26]. The rationale is that: *if ball $B(q,r)$ is empty but $B(q,cr)$ is not, then any point in $B(q,cr)$ is a c-approximate NN of q.* Consider the query point q in Figure 2. Here, ball $B(q,r)$ is empty, but $B(q,cr)$ is not. It follows that the NN of q must have a distance between r and cr to q. Hence, any point in $B(q,cr)$ (i.e., either o_1 or o_2) is a c-approximate NN of q.

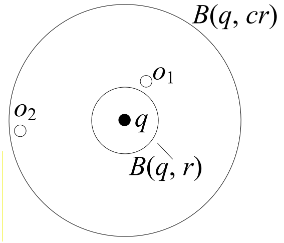


Figure 2: The rationale of the reduction from nearest neighbor to ball cover queries

Based on this idea, Indyk and Motwani [26] propose a structure that supports c-approximate BC queries at $r = 1$, c, c^2, c^3, ..., x respectively, where x is the smallest power of c that is larger than or equal to td (recall that t is the greatest coordinate on each dimension). They give an algorithm [26] to guarantee an approximation ratio of c^2 for NN search (in other words, we need a structure for $\sqrt{c}$-approximate BC queries to support c-approximate NN retrieval). Their method, which we call *rigorous-LSH*, consumes $O((\log_c t + \log_c d) \cdot (dn/B)^{1+1/c})$ space, and answers a query in $O((\log_c t + \log_c d) \cdot (dn/B)^{1/c})$ I/Os. Note that t can be a large value, thus making the space and query cost potentially very expensive. Our LSB-tree will eliminate the factor $\log_c t + \log_c d$ completely.

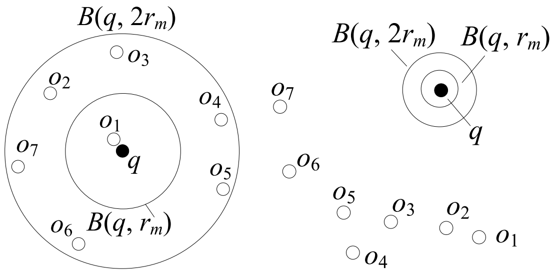
Finally, it is worth mentioning that there exist complicated NN-to-BC reductions [23, 26] with better complexities. However, those reductions are highly theoretical, and are difficult to implement in relational databases.

3.2 Adhoc-LSH

Although *rigorous-LSH* is theoretically sound, its space and query cost is prohibitively expensive in practice. The root of the problem is that it must support BC queries at too many (i.e., $\log_c t + \log_c d$) radii. Gionis et al. [21] remedy this drawback with a heuristic approach, which we refer to as *adhoc-LSH*. Given a NN query q, they return directly the output of *the* BC query that is at location q and has radius r_m, where r_m is a "magic" radius pre-determined by the system. Since only one radius needs to be supported, *adhoc-LSH* improves *rigorous-LSH* by requiring only $O((dn/B)^{1+1/c})$ space and $O((dn/B)^{1/c})$ query time.

Unfortunately, the cost saving of *adhoc-LSH* trades away the quality control on query results. To illustrate, consider Figure 3a, where the dataset $\mathcal{D}$ has 7 points o_1, o_2, ..., o_7, and the black point is a NN query q. Suppose that *adhoc-LSH* is set to support 2-approximate BC queries at radius r_m. Thus, it answers the NN query q by finding a data point that satisfies the 2-approximate BC query located at q with radius r_m. The two circles in Figure 3a represent $B(q,r_m)$ and $B(q,2r_m)$ respectively. As $B(q,r_m)$ covers some data of $\mathcal{D}$, (by the definition stated in the previous subsection) the BC query q may return *any* of the 7 data points in $B(q,2r_m)$. It is clear that no bounded approximation ratio can be ensured, as the real NN o_1 of q can be *arbitrarily* close to q.

The above problem is caused by an excessively large r_m. Conversely, if r_m is too small, *adhoc-LSH* may not return any result at all. To see this, consider Figure 3b. Again, the white points constitute the dataset $\mathcal{D}$, and the two circles are $B(q,r_m)$ and $B(q,2r_m)$. As $B(q,2r_m)$ is empty, the 2-approximate BC query q must not return anything. As a result, *adhoc-LSH* reports nothing too, and is said to have *missed* the query [21].



(a) r_m too large (b) r_m too small

Figure 3: Drawbacks of *adhoc-LSH*

Adhoc-LSH performs well if r_m is roughly equivalent to the distance between q and its exact NN, which is why *adhoc-LSH* can be effective when given the right r_m [21]. Unfortunately, finding such an r_m is non-trivial. Even worse, such r_m may not exist at all; namely, an r_m good for some queries may be bad for others. Figure 4 presents a dataset with two clusters whose densities are drastically different. Apparently, if a NN query q falls in cluster 1, the distance from q to its NN is significantly smaller than if q falls in cluster 2. Hence, it is impossible to choose an r_m that closely captures the NN distances of all queries. Note that clusters with different densities are common in real datasets [11].

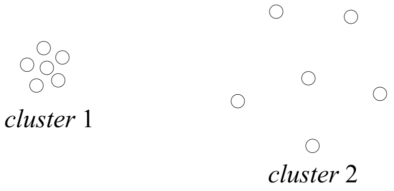


Figure 4: No good r_m exists if clusters have different densities

Recently, Lv et al. [33] present a variation of *adhoc-LSH* with less space consumption. This variation, however, suffers from the same drawback (i.e., no quality control) as *adhoc-LSH*, and entails higher query cost than *adhoc-LSH*.

In summary, currently a practitioner, who wants to apply LSH, faces a dilemma between space/query efficiency and approximation guarantee. If the quality of the retrieved neighbor is crucial (as in security systems such as finger-print verification), a huge amount of space is needed, and large query cost must be paid. On the other hand, to meet a tight space budget or stringent query time requirement, one would have to sacrifice the quality guarantee of LSH, which somewhat ironically is exactly the main strength of LSH.

3.3 Details of hash functions

Let $h(o)$ be a hash function that maps a d-dimensional point o to a one-dimensional value. It is *locality sensitive* if the chance of mapping two points o_1, o_2 to the same value grows as their distance $\|o_1, o_2\|$ decreases. Formally:

DEFINITION 1 (LSH). Given a distance r, approximation ratio c, probability values p_1 and p_2 such that $p_1 > p_2$, a hash function $h(.)$ is (r, cr, p_1, p_2) *locality sensitive* if it satisfies both conditions below:

1. If $\|o_1, o_2\| \leq r$, then $Pr[h(o_1) = h(o_2)] \geq p_1$;
2. If $\|o_1, o_2\| > cr$, then $Pr[h(o_1) = h(o_2)] \leq p_2$. □

LSH functions are known for many distance metrics. For ℓ_p norm, a popular LSH function is defined as follows [15]:

$$h(o) = \left\lfloor \frac{\vec{a} \cdot \vec{o} + b}{w} \right\rfloor. \tag{1}$$

Here, $\vec{o}$ represents the d-dimensional vector representation of a point o; $\vec{a}$ is another d-dimensional vector where each component is drawn independently from a so-called *p-stable* distribution [15]; $\vec{a} \cdot \vec{o}$ denotes the dot product of these two vectors. w is a sufficiently large constant, and finally, b is uniformly drawn from $[0, w)$.

Equation 1 has a simple geometric interpretation. To illustrate, let us consider $p = 2$, i.e., ℓ_p is Euclidean distance. In this case, a 2-stable distribution can be just a normal distribution (mean 0, variance 1), and it suffices to set $w = 16$ [15]. Assuming dimensionality $d = 2$, Figure 5 shows the line that crosses the origin, and its slope coincides with the direction of $\vec{a}$. For convenience, assume that $\vec{a}$ has a unit norm, so that the dot product $\vec{a} \cdot \vec{o}$ is the projection of point o onto line $\vec{a}$, namely, point A in the figure. The effect of $\vec{a} \cdot \vec{o} + b$ is to shift A by a distance b (along the line) to a point B. Finally, imagine we partition the line into intervals with length w; then, the hash value $h(o)$ is the ID of the interval covering B.

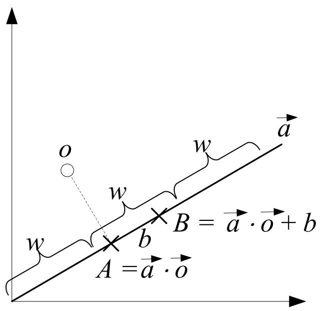


Figure 5: Geometric interpretation of LSH

The intuition behind such a hash function is that, if two points are close to each other, then with high probability their shifted projections (on line $\vec{a}$) will fall in the same interval. On the other hand, two faraway points are very likely to be projected into different intervals. The following is proved in [15]:

LEMMA 1 (PROVED IN [15]). Equation 1 is $(1, c, p_1, p_2)$ locality sensitive, where p_1 and p_2 are two constants satisfying $\frac{\ln 1/p_1}{\ln 1/p_2} \leq \frac{1}{c}$. □

4. LSB-TREE

This section includes everything that a practitioner needs to know to apply LSB-trees. Specifically, Section 4.1 explains how to build a LSB-tree, and Section 4.2 gives its NN algorithm. We will leave all the theoretical analysis to Section 5, including its space, query performance, and quality guarantee. For simplicity, we will assume ℓ_2 norm but the extension to arbitrary ℓ_p norms is straightforward.

4.1 Building a LSB-tree

The construction of a LSB-tree is very simple. Given a d-dimensional dataset $\mathcal{D}$, we first convert each point $o \in \mathcal{D}$ to an m-dimensional point $G(o)$, and then, obtain the *Z-order value* $z(o)$ of $G(o)$. Note that $z(o)$ is just a simple number. Hence, we can index all the resulting Z-order values with a conventional B-tree, which is the LSB-tree. The coordinates of o are stored along with its leaf entry. Next, we clarify the details of each step.

From o to $G(o)$. We set the dimensionality m of $G(o)$ as

$$m = \log_{1/p_2}(dn/B) \tag{2}$$

where p_2 is the constant given in Lemma 1 under $c = 2$, n is the size of dataset $\mathcal{D}$, and B is the page size. As explained in Section 5, this choice of m makes it rather unlikely for two far-away points o_1, o_2 to have $G(o_1)$, $G(o_2)$ that are similar on all m dimensions. Note that, the choice of $c = 2$ is not compulsory, and our technique can be adapted to any integer $c \geq 2$, as discussed in Section 6.

The derivation of $G(o)$ is based on a *family* of hash functions:

$$H(o) = \vec{a} \cdot \vec{o} + b^*. \tag{3}$$

Here, $\vec{a}$ is a d-dimensional vector where each component is drawn independently from the normal distribution (mean 0 and variance 1). Value b^* is uniformly distributed in $[0, 2^f w^2)$, where w is any constant at least 16, and

$$f = \lceil \log_2 d + \log_2 t \rceil. \tag{4}$$

Recall that t is the largest coordinate on each dimension. Note that while $\vec{a}$ and w are the same as in Equation 1, b^* is different, which is an important design underlying the efficiency of the LSB-tree (as elaborated in Section 5 with Lemma 2).

We randomly select m functions $H_1(.)$, ..., $H_m(.)$ independently from the family described by Equation 3. Then, $G(o)$ is the m-dimensional vector:

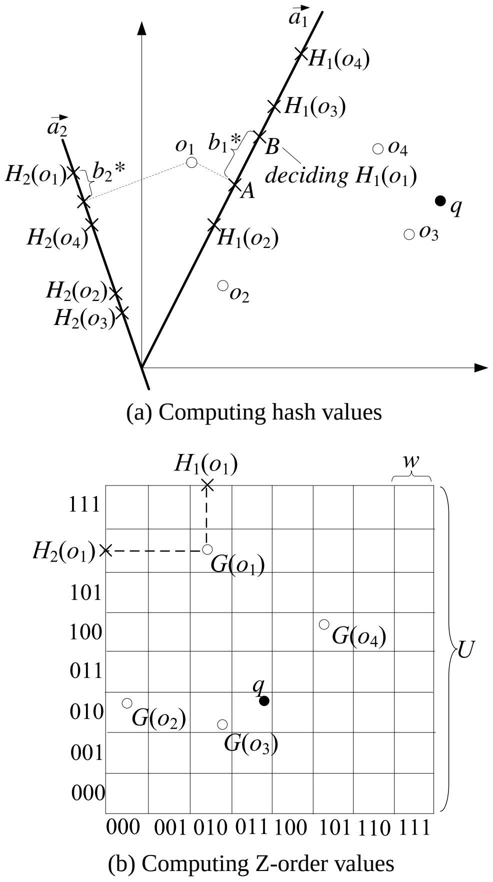
$$G(o) = \langle H_1(o), H_2(o), ..., H_m(o) \rangle. \tag{5}$$

From $G(o)$ to $z(o)$. Let U be the axis length of the m-dimensional space $G(o)$ falls in. As explained shortly, we will choose a value of U such that U/w is a power of 2. Computation of a Z-order curve requires a hyper-grid partitioning the space. We impose a grid where each cell is a hyper-square with side length w; therefore, there are U/w cells per dimension, and totally $(U/w)^m$ cells in the whole grid. Given the grid, calculating the Z-order value $z(o)$ of $G(o)$ is a standard process well-known in the literature [20]. Let $u = \log_2(U/w)$. Each $z(o)$ is thus a binary string with um bits.

Example. To illustrate the conversion, assume that the dataset $\mathcal{D}$ consists of 4 two-dimensional points o_1, o_2, ..., o_4 as shown in Figure 6a. Suppose that we select $m = 2$ hash functions $H_1(.)$ and $H_2(.)$. Let $\vec{a}_1$ ($\vec{a}_2$) be the "$\vec{a}$-vector" in function $H_1(.)$ ($H_2(.)$). For simplicity, assume that both $\vec{a}_1$ and $\vec{a}_2$ have norm 1. In Figure 6a, we slightly abuse notations by also using $\vec{a}_1$ ($\vec{a}_2$) to denote the line that passes the origin, and coincides with the direction of vector $\vec{a}_1$ ($\vec{a}_2$).

Let us take o_1 as an example. The first step of our conversion is to obtain $G(o_1)$, which is a 2-dimensional vector with components $H_1(o_1)$ and $H_2(o_2)$. The value of $H_1(o_1)$ can be understood in the same way as explained in Figure 5. Specifically, first project o_1 onto line $\vec{a}_1$, and then move the projected point A (along the line) by a distance b_1^* to a point B. $H_1(o_1)$ is the distance from B to the origin[1]. $H_2(o_2)$ is computed similarly on line $\vec{a}_2$ (note that the shifting distance is b_2^*).

[1] Precisely speaking, it is $|H_1(o_1)|$ that is equal to the distance. $H_1(o_1)$ itself can be either positive or negative, depending on which side of the origin B lies on.



(a) Computing hash values

(b) Computing Z-order values

Figure 6: Illustration of data conversion

Treating $H_1(o_1)$ and $H_2(o_2)$ as coordinates, in the second step, we regard $G(o_1)$ as a point in a data space as shown in Figure 6b, and derive $z(o_1)$ as the Z-order value of point $G(o_1)$ in this space. In Figure 6b, the Z-order calculation is based on a 8 × 8 grid. As $G(o_1)$ falls in a cell whose (binary) horizontal and vertical labels are 010 and $\underline{110}$ respectively, $z(o_1)$ equals 0$\underline{1}$1$\underline{1}$0$\underline{0}$ (in general, a Z-order value interleaves the bits of the two labels, starting from the most significant bits [20]).

Choice of U. In practice, U can be any value making U/w a sufficiently large power of 2. For theoretical reasoning, next we provide a specific choice for U. Besides U/w being a power of 2, our choice fulfills another two conditions: (i) $U/w \geq 2^f$, where f is given in Equation 4, and (ii) $|H_i(o)|$ is confined to at most $U/2$ for any $i \in [1, m]$.

In the form of Equation 3, for each $i \in [1, m]$, write $H_i(o) = \vec{a}_i \cdot \vec{o} + b_i^*$. Denote by $\|\vec{a}_i\|_1$ the ℓ_1 norm[2] of $\vec{a}_i$. Remember that o distributes in space $[0, t]^d$, where t is the largest coordinate on each dimension. Hence, $|H_i(.)|$ is bounded by

$$H_{\max} = \max_{i=1}^{m} (\|\vec{a}_i\|_1 \cdot t + b_i^*). \tag{6}$$

We thus determine U by setting U/w to the smallest power of 2 that bounds both 2^f and $2H_{\max}/w$ from above.

4.2 Nearest neighbor algorithm

In practice, a single LSB-tree already produces query results with very good quality, as demonstrated in our experiments. To elevate the quality to a theoretical level, we may independently build a number l of trees. We choose

$$l = \sqrt{dn/B}. \tag{7}$$

which, as analyzed in Section 5, ensures a high chance for nearby points o_1, o_2 to have close Z-order values in at least one tree.

Denote the l trees as T_1, T_2, ..., T_l respectively, and call them collectively as a *LSB-forest*. Use $z_j(o)$ to represent the Z-order value of o in tree T_j ($1 \leq j \leq l$). Without ambiguity, we also let $z_j(o)$ refer to the leaf entry of o in T_j. Remember that the coordinates of o are stored in the leaf entry.

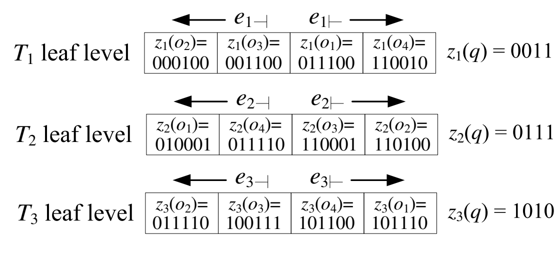
Given a NN query q, we first get its Z-order value $z_j(q)$ in each tree T_j ($1 \leq j \leq l$). As with the Z-order values of data points, $z_j(q)$ is a binary string with um bits. We denote by $LLCP(z_j(o), z_j(q))$ the *length of the longest common prefix* (LLCP) of $z_j(o)$ and $z_j(q)$. For example, suppose $z_j(o) = 100101$ and $z_j(q) = 100001$; then $LLCP(z_j(o), z_j(q)) = 3$. When q is clear from the context, we may refer to $LLCP(z_j(o), z_j(q))$ simply as the *LLCP of* $z_j(o)$.

Figure 7 presents our nearest neighbor algorithm at a high level. The main idea is to visit the leaf entries of all l trees in descending order of their LLCPs, until either enough points have been seen, or we have found a point that is close enough. Next, we explain the details of lines 2 and 3.

```
Algorithm NN
1. repeat
2.     pick, from all the trees $T_1$, ..., $T_l$, the leaf entry with
       the next greatest LLCP
3. until condition $E_1$ or $E_2$ holds
4. return the nearest point found so far
```

Figure 7: The NN algorithm

Finding the next greatest LLCP. This can be done by a synchronous bi-directional expansion at the leaf levels of all trees. Specifically, recall that we have obtained the Z-order value $z_j(q)$ in each tree T_j ($1 \leq j \leq l$). Search T_j to locate the leaf entry $e_{j\vdash}$ with the lowest Z-order value at least $z_j(q)$. Let $e_{j\dashv}$ be the leaf entry immediately preceding $e_{j\vdash}$. To illustrate, Figure 8 gives an example where each Z-order value has $um = 6$ bits, and $l = 3$ LSB-trees are used. The values of $z_1(q)$, $z_2(q)$, and $z_3(q)$ are given next to the corresponding trees. In, for instance, T_1, $z_1(o_1) = 011100$ is the lowest among all the Z-order values at least $z_1(q) = 001110$. Hence, $e_{1\vdash}$ is $z_1(o_1)$, and $e_{1\dashv}$ is the entry $z_1(o_3) = 001100$ preceding $z_1(o_1)$.



T_1 leaf level	$z_1(o_2)$= 000100	$z_1(o_3)$= 001100	$z_1(o_1)$= 011100	$z_1(o_4)$= 110010	$z_1(q)$ = 001110
T_2 leaf level	$z_2(o_1)$= 010001	$z_2(o_4)$= 011110	$z_2(o_3)$= 110001	$z_2(o_2)$= 110100	$z_2(q)$ = 011111
T_3 leaf level	$z_3(o_2)$= 011110	$z_3(o_3)$= 100111	$z_3(o_4)$= 101100	$z_3(o_1)$= 101110	$z_3(q)$ = 101001

Figure 8: Bi-directional expansion ($um = 6, l = 3$)

[2] Given a d-dimensional vector $\vec{a} = \langle a[1], a[2], ..., a[d] \rangle$, $\|\vec{a}\|_1 = \sum_{i=1}^{d} |a[i]|$.

The leaf entry with the greatest LLCP must be in the set $S = \{e_{1\vdash}, e_{1\dashv}, ..., e_{l\vdash}, e_{l\dashv}\}$. Let $e \in S$ be this entry. To determine the leaf entry with the next greatest LLCP, we move e *away* from q by one position in the corresponding tree, and then repeat the process. For example, in Figure 8, the leaf entry with the maximum LLCP is $e_{2\dashv}$ (whose LLCP is 5, as it shares the same first 5 bits with $z_2(q)$). Thus, we shift $e_{2\dashv}$ to its left, i.e., to $z_2(o_1) = 010001$. The entry with the next largest LLCP can be found again in $\{e_{1\vdash}, e_{1\dashv}, ..., e_{3\vdash}, e_{3\dashv}\}$.

Terminating condition. Algorithm *NN* terminates when one of two events $\boldsymbol{E_1}$ and $\boldsymbol{E_2}$ happens. The first event is:

$\boldsymbol{E_1}$: the total number of leaf entries accessed from all l LSB-trees has reached $4Bl/d$.

Event $\boldsymbol{E_2}$ is based on the LLCP of the leaf entry just retrieved from line 2. Denote the LLCP by v, which bounds from above the LLCP of all the leaf entries that have not been processed.

$\boldsymbol{E_2}$: the nearest point found so far (from all the leaf entries already inspected) has distance to q at most $2^{u-\lfloor v/m \rfloor+1}$.

Let us use again Figure 8 to illustrate algorithm *NN*. Assume that the dataset consists of points o_1, o_2, ..., o_4 in Figure 6a, and the query is the black point q. Notice that the Z-order values in tree T_1 are obtained according to the transformation in Figure 6b with $u = 3$ and $m = 2$. Suppose that $\|o_3, q\| = 3$ and $\|o_4, q\| = 5$.

As explained earlier, entry $z_2(o_4)$ in Figure 8 has the largest LLCP $v = 5$, and thus, is processed first. *NN* obtains the object o_4 associated with $z_2(o_4)$, and calculates its distance to q. Since $\|o_4, q\| = 5 > 2^{u-\lfloor v/m \rfloor+1} = 4$, condition $\boldsymbol{E_2}$ does not hold. Assuming $\boldsymbol{E_1}$ is also violated (i.e., let $4Bl/d > 1$), the algorithm processes the entry with the next largest LLCP, which is $z_1(o_3)$ in Figure 8 whose LLCP $v = 4$. In this entry, *NN* finds o_3 which replaces o_4 as the nearest point so far. As now $\|o_3, q\| = 3 \le 2^{u-\lfloor v/m \rfloor+1} = 4$, $\boldsymbol{E_2}$ holds, and *NN* terminates by returning o_3.

Retrieving $\boldsymbol{k}$ neighbors. A direct extension of NN search is k *nearest neighbor* (kNN) retrieval, which aims at finding the k points in the dataset $\mathcal{D}$ nearest to a query point q. Algorithm *NN* can be easily adapted to answer kNN queries. Specifically, it suffices to modify $\boldsymbol{E_2}$ to "*q is within distance $2^{u-\lfloor v/m \rfloor+1}$ to the k nearest points found so far*". Also, apparently line 4 should return the k nearest points.

$\boldsymbol{k}$NN search with a single tree. Maintaining a forest of l LSB-trees incurs large space consumption and update overhead. In practice, we may prefer an index that has linear space and supports fast data insertions/deletions. In this case, we can build only one LSB-tree, and use it to process kNN queries. Accordingly, we slightly modify the algorithm *NN* by simply ignoring event $\boldsymbol{E_1}$ in the terminating condition (as this event is designed specifically for querying l trees). Condition $\boldsymbol{E_2}$, however, is retained. As a tradeoff for efficiency, querying only a single tree loses the theoretical guarantees of the LSB-forest (as established in the next section). Nevertheless, this approach is expected to return neighbors with high quality, because the converted Z-order values adequately preserve the proximity of the data points in the original data space.

5. THEORETICAL ANALYSIS

We now proceed to study the theoretical characteristics of the LSB-tree. Denote by $\mathbb{R}$ the original d-dimensional space of the dataset $\mathcal{D}$. Namely, $\mathbb{R} = [0, t]^d$, where t is the maximum coordinate on each axis. Recall that, to construct a LSB-tree, we convert each point $o \in \mathcal{D}$ to an m-dimensional point $G(o)$ as in Equation 5. Denote by $\mathbb{G}$ the space where $G(o)$ is distributed. By the way we select U in Section 4.1, $\mathbb{G} = [-U/2, U/2]^m$.

5.1 Quality guarantee

We begin with an observation on the basic LSH in Equation 1:

OBSERVATION 1. Given any integer $x \ge 1$, define hash function

$$h'(o) = \left\lfloor \frac{\vec{a} \cdot \vec{o} + bwx}{w} \right\rfloor \tag{8}$$

where $\vec{a}$, b, and w are the same as in Equation 1. $h'(.)$ is $(1, c, p_1, p_2)$ locality sensitive, and $\frac{\ln 1/p_1}{\ln 1/p_2} \le 1/c$.

PROOF. (Sketch) Due to the space constraint, we point out only the most important fact behind the correctness. Imagine a line that has been partitioned into consecutive intervals of length w. Let A, B be two points on this line with distance $y \le w$. Shift both points towards right by a distance uniformly drawn from $[0, w^2 x)$, where x is any integer. After this, A and B fall in the same interval with probability $1 - y/w$. This probability does not depend on x. □

For any $s \in [0, f]$ with f given in Equation 4, define:

$$H^*(o, s) = \left\lfloor \frac{\vec{a} \cdot \vec{o} + b^*}{2^s w} \right\rfloor \tag{9}$$

where $\vec{a}$, b^* and w follow those in Equation 3. We have:

LEMMA 2. $H^*(o, s)$ is $(2^s, 2^{s+1}, p_1, p_2)$ locality sensitive, where p_1 and p_2 satisfy $\frac{\ln 1/p_1}{\ln 1/p_2} \le 1/2$.

PROOF. Create another space $\mathbb{R}'$ by dividing all coordinates of $\mathbb{R}$ by 2^s. It is easy to see that the distance of two points in $\mathbb{R}$ is 2^s times the distance of their converted points in $\mathbb{R}'$. Consider

$$h''(o') = \left\lfloor \frac{\vec{a} \cdot \vec{o'} + (b^*/2^f w)(2^{f-s} w)}{w} \right\rfloor \tag{10}$$

where o' is a point in $\mathbb{R}'$. As $b^*/(2^f w)$ is uniformly distributed in $[0, w]$, by Observation 1, $h''(.)$ is $(1, 2, p_1, p_2)$ locality sensitive in $\mathbb{R}'$ with $(\ln 1/p_1)/(\ln 1/p_2) \le 1/2$. Let o be the corresponding point of o' in $\mathbb{R}$. Clearly, $\vec{a} \cdot \vec{o'} = (\vec{a} \cdot \vec{o})/2^s$. Hence, $h''(o') = H_s(o, s)$. The lemma thus holds. □

As shown in Equation 5, $G(o)$ is composed of hash values $H_1(o)$, ..., $H_m(o)$. In the way we obtain $H^*(o, s)$ (Equation 9) from $H(o)$ (Equation 3), let $H_i^*(o, s)$ be the hash function corresponding to $H_i(o)$ ($1 \le i \le m$). Also remember that $z(o)$ is the Z-order value of $G(o)$ in space $\mathbb{G}$, and function $LLCP(.,.)$ returns the length of the longest common prefix of two Z-order values. Now we prove a crucial lemma that is the key to the design of the LSB-tree.

LEMMA 3. Let o_1, o_2 be two arbitrary points in space $\mathbb{R}$. A value s satisfies $s \ge u - \lfloor LLCP(z(o_1), z(o_2))/m \rfloor$ if and only if $H_i^*(o_1, s) = H_i^*(o_2, s)$ for all $i \in [1, m]$.

PROOF. Recall that, for Z-order value calculation, we impose on $\mathbb{G}$ a grid with 2^u cells (each with side length w) per dimension. Refer to the entire $\mathbb{G}$ as a level-u *tile*. In general, a level-s ($2 \le s \le u$) tile defines 2^m level-$(s-1)$ tiles, by cutting the level-s tile in half on every dimension. Thus, each cell in the grid partitioning $\mathbb{G}$ is a level-0 tile.

As a property of the Z-order curve, $G(o_1)$ and $G(o_2)$ belong to a level-s tile, if and only if their Z-order values share at least $m(u-s)$ most significant bits [20], namely, $LLCP(z(o_1), z(o_2)) \geq m(u-s)$. On the other hand, note that a level-s tile is a hyper-square with side length $2^s w$. This means that $G(o_1)$ and $G(o_2)$ belong to a level-s tile, if and only if $H_i^*(o_1, s) = H_i^*(o_2, s)$ for all $i \in [1, m]$. Thus, the lemma follows. □

Lemmas 2 and 3 allow us to rephrase the probabilistic guarantees of LSH using LLCP.

COROLLARY 1. Let r be any power of 2 at most 2^f. Given a query point q and a data point o, we have:

1. If $\|q, o\| \leq r$, then $LLCP(z(q), z(o)) \geq m(u - \log_2 r)$ with probability at least p_1^m.
2. If $\|q, o\| > 2r$, then $LLCP(z(q), z(o)) \geq m(u - \log_2 r)$ with probability at most p_2^m.

Furthermore, $\frac{\ln 1/p_1}{\ln 1/p_2} \leq 1/2$. □

The above result holds for any LSB-tree. Recall that, for NN search, we need a forest of l trees T_1, ..., T_l built independently. Next, we will explain an imperative property guaranteed by these trees. Let q be the query point, and r be any power of 2 up to 2^f such that there is a point o^* in the ball $B(q, r)$. Consider events $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$:

$\boldsymbol{P_1}$: $LLCP(z_j(q), z_j(o^*)) \geq m(u - \log_2 r)$ in at least one tree T_j ($1 \leq j \leq \ell$).

$\boldsymbol{P_2}$: There are less than $4Bl/d$ leaf entries $z_j(o)$ from all trees T_j ($1 \leq j \leq l$) such that (i) $LLCP(z_j(q), z_j(o)) \geq m(u - \log_2 r)$, and (ii) o is outside $B(q, 2r)$.

The property guaranteed by the l trees is:

LEMMA 4. $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ hold at the same time with at least constant probability.

PROOF. Equipped with Corollary 1, this proof is analogous to the standard proof [21] of the correctness of LSH. □

Now we establish an approximation ratio of 4 for algorithm *NN*. In the next section, we will extend the LSB-tree to achieve better approximation ratios.

THEOREM 1. Algorithm *NN* returns a 4-approximate NN with at least constant probability.

PROOF. Let o^* be the NN of query q, and $r^* = \|o^*, q\|$. Let r be the smallest power of 2 bounding r^* from above. Obviously $r < 2r^*$ and $r \leq 2^f$ (notice that r^* is at most $td \leq 2^f$ under any ℓ_p norm). If when *NN* finishes, it has already found o^* in any tree, apparently it will return o^* which is optimal. Next, we assume *NN* has not seen o^* at termination.

We will show that when both $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ are true, the output of *NN* is definitely 4-approximate. Denote by j^* the j stated in $\boldsymbol{P_1}$. Recall that *NN* may terminate due to the occurrence of either event $\boldsymbol{E_1}$ or $\boldsymbol{E_2}$. If it is due to $\boldsymbol{E_2}$, and given the fact that *NN* visits leaf entries in descending order of their LLCP, the LLCP v of the last fetched leaf entry is at least $LLCP(z_{j^*}(q), z_{j^*}(o^*)) \geq m(u - \log_2 r)$. It follows that $\lfloor v/m \rfloor \geq u - \log_2 r$. $\boldsymbol{E_2}$ ensures that we return a point o with $\|o, q\| \leq 2r < 4r^*$.

In case the termination is due to $\boldsymbol{E_1}$, by $\boldsymbol{P_2}$, we know that *NN* has seen at least one point o inside $B(q, 2r)$. Hence, the point returned has distance to q at most $2r < 4r^*$. Finally, Lemma 4 indicates that $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ are true with at least constant probability, thus completing the proof. □

Also, the proof of Theorem 1 actually shows:

COROLLARY 2. Let r^* be the distance from q to its real NN. With at least constant probability, *NN* returns a point within distance $2r$ to q, where r is the lowest power of 2 bounding r^* from above. □

As a standard trick in probabilistic algorithms, by repeating our solution a constant $O(\log 1/p)$ number of times, we boost the success probability of algorithm *NN* from constant to at least $1-p$, for any arbitrarily low $p > 0$.

5.2 Space and query time

THEOREM 2. We can build a forest of l LSB-trees that consume totally $O((dn/B)^{1.5})$ space. Given these trees, algorithm *NN* answers a 4-approximate NN query in $O(E\sqrt{dn/B})$ I/Os, where E is the height of a LSB-tree.

PROOF. Each leaf entry of a LSB-tree stores a Z-order value $z(o)$ and the coordinates of o. $z(o)$ has um bits where $u = O(f) = O(\log_2 d + \log_2 t)$ and $m = O(\log(dn/B))$. As $\log_2 d + \log_2 t$ bits fit in 2 words, $z(o)$ occupies $O(\log(dn/B))$ words. It takes d words to store the coordinates of o. Hence, overall a leaf entry is $O(d)$ words long. Hence, a LSB-tree consumes $O((dn/B))$ pages, and $l = \sqrt{dn/B}$ of them require totally $O((dn/B)^{1.5})$ space.

Algorithm *NN* (i) first accesses a single path in each LSB-tree, and then (ii) fetches at most $4Bl/d$ leaf entries. The cost of (i) is bounded by $O(lE)$. As a leaf entry consumes $O(d)$ words, $4Bl/d$ of them occupy at most $O(l)$ pages. □

By implementing each LSB-tree as a *string B-tree* [19], the height E is bounded by $O(\log_B(n/B))$, resulting in query complexity $O(\log_B(n/B) \cdot \sqrt{dn/B})$.

5.3 Comparison with rigorous-LSH

As discussed in Section 3, for 4-approximate NN search, *rigorous-LSH* consumes $O((\log_2 d + \log_2 t)(dn/B)^{1.5})$ space, and answers a query in $O((\log_2 d + \log_2 t)\sqrt{dn/B})$ I/Os. Comparing these complexities with those in Theorem 2, it is clear that the LSB-forest improves *rigorous-LSH* significantly in the following ways.

First, the performance of the LSB-forest is not sensitive to t, the greatest coordinate of a dimension. This is a crucial improvement because t can be very large in practice. As a result, *rigorous-LSH* is suitable only when data are confined to a relatively small space. The LSB-forest enjoys much higher applicability by retaining the same efficiency regardless of the size of the data space.

Second, the space consumption of a LSB-forest is lower than that of *rigorous-LSH* by a factor of $\log_2 d + \log_2 t$. For practical values of d and t (e.g., $d = 50$ and $t = 10000$), the space of a LSB-forest is lower than that of *rigorous-LSH* by more than *an order of magnitude*. Furthermore, note that the LSB-forest is as space efficient as *adhoc-LSH*, even though the latter does not guarantee the quality of query results at all.

Third, the LSB-forest promises higher query efficiency than *rigorous-LSH*. As mentioned earlier, the height E can be strictly confined to $O(\log_B(n/B))$ by resorting to the string B-tree. Even if we simply implement a LSB-tree as a normal B-tree, the height

```
Algorithm NN2 ($r$)
1. $o$ = the output of algorithm NN on $F$
2. $o'$ = the output of algorithm NN on $F'$
3. return the point between $o$ and $o'$ closer to $q$
```

Figure 9: The 3-approximate algorithm

E never grows beyond 6 in our experiments. This is expected to be much smaller than $\log_2 d + \log_2 t$, rendering the query complexity of the LSB-forest considerably lower than that of *rigorous-LSH*.

In summary, the LSB-forest outperforms *rigorous-LSH* significantly in applicability, space and query efficiency. It therefore eliminates the reason for resorting to the theoretically vulnerable approach of *adhoc-LSH*. Finally, remember that the LSB-tree achieves all of its nice characteristics by leveraging purely relational techniques.

6. EXTENSIONS

This section presents several interesting extensions to the LSB-tree, which are easy to implement in a relational database, and extend the functionality of the LSB-tree significantly.

Supporting ball cover. A LSB-forest, which is a collection of l LSB-trees as defined in Section 4.2, is able to support 2-approximate BC queries whose radius r is any power of 2. Specifically, given such a query q, we run algorithm *NN* (Figure 7) using the query point. Let o by the output of *NN*. If $\|o, q\| \le 2r$, we return o as the result of the BC query q. Otherwise, we return nothing. By an argument similar to the proof of Theorem 1, it is easy to prove that the above strategy succeeds with high probability.

$(2 + \epsilon)$-approximate nearest neighbors. A LSB-forest ensures an approximation ratio of 4 (Theorem 1). Next we will improve the ratio to 3 with only 2 LSB-forests. As shown earlier, a LSB-forest can answer 2-approximate BC queries with any $r = 1, 2, 2^2, ..., 2^f$ where f is given in Equation 4. We build another LSB-forest to handle 2-approximate BC queries with any $r = 1.5, 1.5 \times 2, 1.5 \times 2^2, ..., 1.5 \times 2^f$. For this purpose, we only need to create another dataset $\mathcal{D}'$ from $\mathcal{D}$, by dividing all coordinates in $\mathcal{D}$ by 1.5. Then, a LSB-forest on $\mathcal{D}'$ is exactly what we need, noticing that the distance of two points in $\mathcal{D}'$ is 1.5 times smaller than that of their original points in $\mathcal{D}$. Denote by F and F' the LSB-forest on $\mathcal{D}$ and $\mathcal{D}'$ respectively.

Given a NN query q, we answer it using simple the algorithm in Figure 9.

THEOREM 3. Algorithm *NN2* returns a 3-approximate NN with at least constant probability.

PROOF. Let $\mathbb{R}$ be the d-dimensional space of dataset $\mathcal{D}$, and $\mathbb{R}'$ the space of $\mathcal{D}'$. Denote by r^* the distance between q and its real NN o^*. Apparently, r^* must fall in either $(2^x, 1.5 \times 2^x]$ or $(1.5 \times 2^x, 2^{x+1}]$ for some $x \in [0, f]$. Refer to these possibilities as Case 1 and 2, respectively.

For Case 1, the distance $r^{*\prime}$ between q and o^* in space $\mathbb{R}'$ is between $(2^x/1.5, 2^x]$. Hence, by Corollary 2, with at least constant probability the distance between o' and q in $\mathbb{R}'$ is at most 2^{x+1}, where o' is the point output at line 2 of *NN2*. It thus follows that o' is within distance $1.5 \times 2^{x+1} \le 3r^*$ in $\mathbb{R}$. Similarly, for Case 2, we can show that o (output at line 1) is a 3-approximate NN with at least constant probability. □

The above idea can be easily extended to $(2 + \epsilon)$-approximate NN search for any $\epsilon \in (0, 2]$. Specifically, we can maintain $\lfloor 2/\epsilon \rfloor$ LSB-forests, such that the i-th forest ($0 \le i < \lfloor 2/\epsilon \rfloor$) supports 2-approximate BC queries at $r = (1 + \alpha), 2(1 + \alpha), 2^2(1 + \alpha), ..., 2^f(1 + \alpha)$, where $\alpha = i\epsilon/2$. Given a query q, we run algorithm *NN* on all the forests, and return the nearest point found. By an argument similar to proving Theorem 3, we have:

THEOREM 4. For any $\epsilon \in (0, 2]$, we can build $\lfloor 2/\epsilon \rfloor$ LSB-forests that consume totally $O(\frac{1}{\epsilon}(dn/B)^{1.5})$ space, and answer a $(2 + \epsilon)$-approximate NN query in $O(\frac{1}{\epsilon}E\sqrt{dn/B})$ I/Os, where E is the height of a LSB-tree. □

By further generalizing the idea, we can achieve the approximation ratio $c + \epsilon$ for any integer $c \ge 3$ with space and query time that monotonically decrease as c increases. This provides a graceful tradeoff between quality and efficiency. We leave the details to the full paper.

7. RELATED WORK

NN search is well understood in low dimensional space [24, 35]. This problem, however, becomes much more difficult in high dimensional space. Many algorithms (e.g., those based on data or space partitioning indexes [20]) that perform nicely on low dimensional data, deteriorate rapidly as the dimensionality increases [10, 36], and are eventually outperformed even by sequential scan.

Research on high-dimensional NN search can be divided into *exact* and *approximate* retrieval. In the exact category, Lin et al. [32] propose the *TV-tree* which improves conventional R-trees [5] by creating MBRs only in selected subspaces. Weber et al. [36] design the *VA-file*, which compresses the dataset to minimize the cost of sequential scan. Also based on the idea of compression, Berchtold et al. [7] develop the *IQ-tree*, combining features of the R-tree and VA-file. Chaudhuri and Gravano [12] perform NN search by converting it to range queries. In [8] Berchtold et al. provide a solution leveraging high-dimensional Voronoi diagrams, whereas Korn et al. [28] tackle the problem by utilizing the fractal dimensionality of the dataset. Koudas et al. [29] give a bitmap-based approach. The state of the art is due to Jagadish et al. [27]. They develop the *iDistance* technique that converts high-dimensional points to 1D values, which are indexed using a B-tree for NN processing. We will compare our solution to *iDistance* experimentally.

In exact search, a majority of the query cost is spent on *verifying* a point as a real NN [6, 14]. Approximate retrieval improves efficiency by relaxing the precision of verification. Goldstein and Ramakrishnan [22] leverage the knowledge of the query distribution to balance the efficiency and result quality. Ferhatosmanoglu et al. [18] find NNs by examining only the interesting subspaces. Chen and Lin [13] combine sampling with a reduction [12] to range search. Li et al. [31] first partition the dataset into clusters, and then prunes the irrelevant clusters according to their radii. Houle and Sakuma [25] develop *SASH* which is designed for memory-resident data, but is not suitable for disk-oriented data due to severe I/O thrashing. Fagin et al. [16] develop the *MedRank* technique that converts the dataset to several sorted lists by projecting the data onto different vectors. To answer a query, *MedRank* traverses these lists in a way similar to the *threshold* algorithm [17] for top-k search. We will also evaluate *MedRank* in the experiments.

None of the aforementioned solutions ensures sub-linear growth of query cost in the worst case. How to achieve this has been carefully studied in the theory community (see, for example, [23, 30] and the references therein). Almost all the results there, however, are excessively complex for practical implementation, except

LSH. This technique is invented by Indyk and Motwani [26] for in-memory data. Gionis et al. [21] adapt it to external memory, but as discussed in Section 3.2, their method loses the guarantee on the approximation ratio. The locality-sensitive hash functions for l_p norms are discovered by Datar et al. [15]. Bawa et al. [4] propose a method to tune the parameters of LSH automatically. Their method, however, no longer ensures the same query performance as LSH unless the adopted hash function has a so-called "$(\epsilon, f(\epsilon))$ property" [4]. Unfortunately, no existing hash function for ℓ_p norms is known to possess this property. LSH has also received other theoretical improvements [1, 34] which cannot be implemented in relational databases. Furthermore, several heuristic variations of LSH have also been suggested. For example, Lv et al. [33] reduce space consumption by probing more data in answering a query, while recently Athitsos et al. [3] introduce the notion of *distance-based hashing*. The solutions of [3, 33] guarantee neither sub-linear cost nor good approximation ratios.

8. EXPERIMENTS

Next we experimentally evaluate the performance of the LSB-tree, using the existing methods as benchmarks. Section 8.1 describes the datasets and queries. Section 8.2 lists the techniques to be evaluated, and Section 8.3 elaborates the assessment metrics. Section 8.4 demonstrates the superiority of the LSB-forest over alternative LSH implementations. Finally, Section 8.5 verifies that the LSB-tree significantly outperforms the state of the art in both exact and approximate NN search.

8.1 Data and queries

We experiment with both synthetic and real datasets. Synthetic data are generated according to a *varden* distribution to be clarified shortly. As for real data, we deploy datasets *color* and *mnist*, which are used in the papers [16, 27] that develop the best linear-space method for exact and approximate NN retrieval, respectively. Each *workload* contains 50 queries that follow the same distribution as the underlying dataset. All data and queries are normalized such that each dimension has domain $[0, 10000]$. The distance metric is Euclidean distance. The details of *varden*, *color*, and *mnist* are as follows.

Varden. This distribution contains two clusters with drastically different densities. The sparse cluster has 10 points, whereas all the other points belong to the dense cluster. Furthermore, the dense cluster has the shape of a ring, whose radius is comparable to the average mutual distance of the points in the sparse cluster. The two clusters are well separated. Figure 10 illustrates the idea with a 2D example. We vary the cardinality of a *varden* dataset from 10k to 100k, and its dimensionality from 25 to 100. In the sequel, we will denote a d-dimensional *varden* dataset with cardinality n by *varden*-nd. The corresponding workload of a *varden* dataset has 10 and 40 query points that fall in the areas of the sparse and dense clusters, respectively. No query point coincides with any data point.

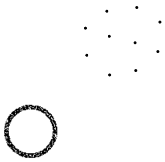


Figure 10: The *varden* distribution

Color. This is a 32-dimensional dataset[3] with 68,040 points, where each point describes the color histogram of an image in the Corel collection [27]. We randomly remove 50 points to form a query set. As a result, our *color* dataset has cardinality 67,990.

Mnist. The original *mnist* dataset[4] is a set of 60,000 points. Each point is 784-dimensional, capturing the pixel values of a 28×28 image. Since, however, most pixels are insignificant, we reduce dimensionality by taking the 50 dimensions with the largest variances. The *mnist* collection also contains a test set of 10,000 points [16], among which we randomly pick 50 to form our workload. Obviously, each query point is also projected onto the same 50 dimensions output by dimensionality reduction.

8.2 Methods

Sequential scan (*SeqScan*). The brute-force approach is included because it is known to be a strong competitor in high dimensional NN retrieval. Furthermore, the relative performance with respect to *SeqScan* serves as a reliable way to compare against methods that are reported elsewhere but not in our experiments.

LSB-forest. As discussed in Section 4.2, this method keeps l LSB-trees, and applies algorithm *NN* in exactly the way given in Figure 7 for query processing.

LSB-no$\boldsymbol{E_2}$. Same as *LSB-forest* except that it disables the terminating condition $\boldsymbol{E_2}$ in algorithm *NN*. In other words, *LSB-no*E_2 terminates on condition $\boldsymbol{E_1}$ only.

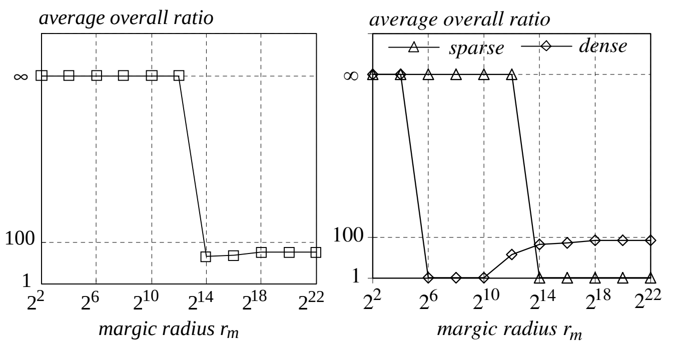
LSB-tree. This method deploys a single LSB-tree (as opposed to l in *LSB-forest*), and hence, requires only linear space and can be updated efficiently. As mentioned at the end of Section 4.1, it disables condition $\boldsymbol{E_1}$, and terminates on $\boldsymbol{E_2}$ only.

***Rigorous-* [26] and *adhoc-LSH* [21].** These are the existing LSH-implementations as reviewed in Sections 3.1 and 3.2, respectively. Recall that both methods are designed for c-approximate BC search. We set c to 2 because a LSB-forest also guarantees the same approximation ratio as mentioned in Section 6. *Adhoc-LSH* requires a set of l hash tables to enable BC queries at a magic radius (to be tuned experimentally later), where l is the same as in Equation 7. *Rigorous-LSH* can be regarded as combining multiple versions of *adhoc-LSH*, one for every radius supported.

***iDistance* [27].** The state of the art for exact NN search. As mentioned in Section 7, it indexes a dataset using a single B-tree after converting all points to 1D values. As with *LSB-tree*, it consumes linear space and supports data insertions and deletions efficiently.

***MedRank* [16].** The best linear-space method for approximate NN search. Given a dataset, *MedRank* creates M sorted lists, such that every data point has an entry in each list. Each entry has the form (id, key), where id uniquely identifies a point, and key is its sorting key (a point has various keys in different lists). Each list is indexed by a B-tree on the keys. Point coordinates are stored in a separate hash table to facilitate probing by id. The number M of lists equals $\log n$ (following Theorem 4 in [16]), where n is the dataset cardinality. It should be noted that *MedRank* is not friendly to updates, because a single point insertion/deletion requires updating all the $\log n$ lists.

[3] *http://kdd.ics.uci.edu/databases/CorelFeatures/.*
[4] *http://yann.lecun.com/exdb/mnist.*



(a) Quality vs. r_m (b) Quality of sparse and dense queries

Figure 11: Magic radius tuning for *adhoc-LSH* (*varden-10k50d*)

d	25	50	75	100
rigorous-LSH	1			
adhoc-LSH	43	66.4	87	104.2
LSB-forest	1.02	1.02	1.02	1.01
LSB-noE$_2$	1			

(a) Average overall ratio vs. dimensionality d (n = 50k)

n	10k	25k	50k	75k	100k
rigorous-LSH	1				
adhoc-LSH	66.4	68.1	70.3	76.5	87.1
LSB-forest	1.02	1.02	1.03	1.02	1.02
LSB-noE$_2$	1				

(b) Average overall ratio vs. cardinality n (d = 50)

Table 1: Result quality on *varden* data

8.3 Assessment metrics

We compare alternative techniques by their quality of results (for approximate solutions), query cost, and space consumption. For query cost, we measure the number of I/Os. CPU time is ignored because it is significantly dominated by I/O overhead for all methods. The page size is fixed to 4,096 bytes.

We evaluate the quality of a kNN result by how many times farther a reported neighbor is than the real NN. Formally, let o_1, o_2, ..., o_k be the k neighbors that a method retrieves for a query q, in ascending order of their distances to q. Let o_1^*, o_2^* ..., o_k^* be the actual first, second, ..., k-th NNs of q, respectively. For any $i \in [1, k]$, we define the *rank-i* (*approximation*) *ratio*, denoted by $R_i(q)$, as

$$R_i(q) = \|o_i, q\| / \|o_i^*, q\|. \tag{11}$$

The *overall* (*approximation*) *ratio* is the mean of the ratios of all ranks, namely, $(\sum_{i=1}^{k} R_i(q))/k$. When a query result is exact, all ratios are 1.

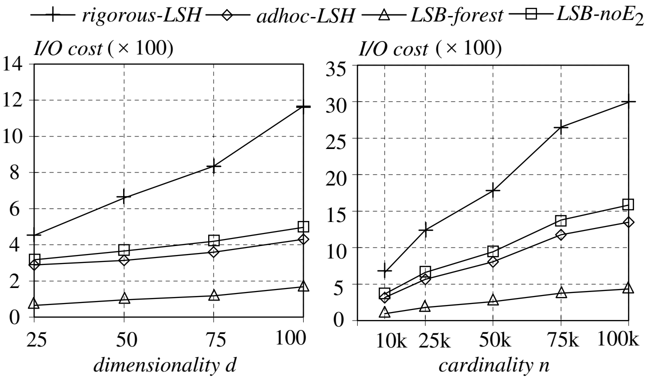
Given a workload W, define its *average overall ratio* as the mean of the overall ratios of all queries in W. This metric reflects the general quality of all k neighbors, and is used in most experiments. In some cases, we may need to scrutinize the quality of neighbors at individual ranks. For this purpose, we will inspect the *average rank-i ratio* ($1 \le i \le k$), which is the mean of the rank-i ratios of all queries in W, namely, $(\sum_{\forall q \in W} R_i(q))/|W|$.

8.4 Behavior of LSH implementations

This section explores the properties of *LSB-forest*, *LSB-noE*$_2$, *rigorous-LSH*, and *adhoc-LSH*. Since their theoretical guarantees hold on $k = 1$ only, we focus on single NN search, where the overall ratio of a query is identical to its rank-1 ratio. We deploy *varden* data, as it allows us to examine different dimensionalities and cardinalities. Unless otherwise stated, a *varden* dataset has default cardinality n = 10k and dimensionality d = 50.

Recall that *adhoc-LSH* answers a NN query by processing instead a BC query with a magic radius r_m. As argued in Section 3.2, there may not exist an r_m that can ensure the quality of all NN queries. To demonstrate this, Figure 11a shows the average overall ratio of *adhoc-LSH* as r_m varies from 2^2 to 2^{22}. For small r_m, the ratio is ∞, implying at least one query in the workload which *adhoc-LSH missed*, namely, returning nothing at all. The ratio improves suddenly to 66 when r_m reaches 2^{14}, and stabilizes as r_m grows further. It is thus clear that, given any r_m, the result of *adhoc-LSH* is at least 66 times worse than the real NN on average!

As discussed in Section 3.2, if r_m is considerably smaller than the NN-distance of a query, *adhoc-LSH* may return an empty result. Conversely, if r_m is considerably larger, *adhoc-LSH* may output a point that is much worse than the real NN. Next, we will experimentally confirm these findings. Recall that a workload for *varden* has queries in both the sparse and dense clusters. Let us call the former (latter) *sparse* (*dense*) *queries*. We observe that the average NN distance of a sparse (dense) query is around 12,000 (15). The phenomenon in Figure 11a occurs because *values of r_m good for sparse queries are bad for dense queries, and vice versa*. To support the claim, Figure 11b plots the average overall ratios of sparse and dense queries separately. For $r_m \le 2^{13}$ = 8,192, it is much lower than the NN-distances of sparse queries, for which *adhoc-LSH* returns nothing (hence, the *sparse* curve in Figure 11b stays at ∞ for all $r_m \le 2^{13}$). Starting at $r_m = 2^{12}$, on the other hand, *adhoc-LSH* often returns very bad results for dense queries. Since the situation gets worse for larger r_m, the *dense* curve Figure 11b increases continuously since 2^{12}. In all the following experiments, we fix r_m to the optimal value 2^{14}.



(a) Cost vs. d (n = 50k) (b) Cost vs. n (d = 50)

Figure 12: Query efficiency on *varden* data

The next experiment compares the result quality of *rigorous-LSH*, *adhoc-LSH*, *LSB-forest*, and *LSB-noE*$_2$. Table 1a (1b) shows their average overall ratios under different dimensionalities (cardinalities). Both *rigorous-LSH* and *LSB-noE*$_2$ achieve perfect quality, namely, they successfully return exactly the real NN for all queries. *LSB-forest* incurs slightly higher error because it accesses fewer points than *LSB-noE*$_2$, and thus, has a lower chance of encountering the real NN. *Adhoc-LSH* is by far the worst method, and its effectiveness deteriorates rapidly as the dimensionality or cardinality increases.

To evaluate the query efficiency of the four methods. Figure 12a (12b) plots their I/O cost as a function of dimensionality d (car-

d	25	50	75	100
rigorous-LSH	894	1,670	2,587	3,420
adhoc-LSH	55	106	167	223
LSB-forest	55	106	167	223

(a) Space vs. dimensionality d (n = 50k)

n	10k	25k	50k	75k	100k
rigorous-LSH	1,670	7,206	20,695	43,578	66,676
adhoc-LSH	106	460	1,323	2,785	4,262
LSB-forest	106	460	1,323	2,785	4,262

(b) Space vs. cardinality n (d = 50)

Table 2: Space consumption on *varden* data in mega bytes

dinality n). *LSB-forest* considerably outperforms its competitors in all cases. Notice that while *LSB-no*E_2 is slightly more costly than *adhoc-LSH*, *LSB-forest* entails only a fraction of the overhead of *adhoc-LSH*. This phenomenon reveals the importance of having terminating condition $\boldsymbol{E_2}$ in the *NN* algorithm. *Rigorous-LSH* is much more expensive than the other approaches, which is consistent with its vast asymptotical complexity.

Tables 2a and 2b show the space consumption (in mega bytes) of each solution as a function of d and n, respectively. *LSB-no*E_2 is not included because it differs from *LSB-forest* only in the query algorithm, and thus, has the same space cost as *LSB-forest*. Furthermore, *adhoc-LSH* also occupies as much space as *LSB-forest*, because a hash table of the former stores the same information as a LSB-tree of the latter. As predicted by their space complexities, *rigorous-LSH* requires more space than *LSB-forest* by a factor of $\log d + \log t$, where t (the largest coordinate on each dimension) equals 10,000 in our experiments.

It is evident from the above discussion that *LSB-forest* is overall the best technique. Specifically, it retains the query accuracy of *rigorous-LSH*, consumes the same space as *adhoc-LSH*, and incurs significantly smaller query cost than both.

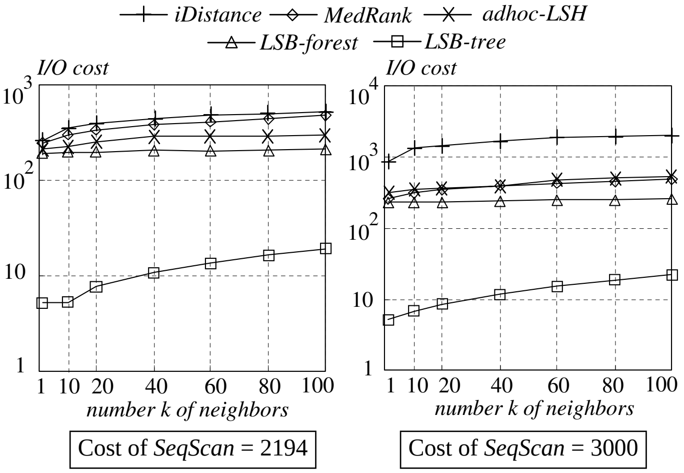
8.5 Practical comparison

Having verified the correctness of our theoretical analysis, in the sequel we assess the practical performance of *SeqScan*, *LSB-tree*, *LSB-forest*, *adhoc-LSH*, *MedRank*, and *iDistance*. *Rigorous-LSH* and *LSB-no*E_2 are omitted because the former incurs gigantic space/query cost, and the latter is merely an auxiliary method for demonstrating the importance of condition $\boldsymbol{E_2}$. Remember that *SeqScan* and *iDistance* return exact NNs, whereas the other methods are approximate.

Only real dataset *color* or *mnist* is adopted in the subsequent evaluation. The workload on *color* (*mnist*) has an average NN distance of 833 (11,422). We set the magic radius of *adhoc-LSH* to the smallest power of 2 that bounds the average NN distance from above, namely, 1,024 and 16,384 for *color* and *mnist*, respectively. The number k of retrieved neighbors will vary from 1 to 100. A buffer of 50 pages is allowed for all methods.

Let us start with query efficiency. Figure 13a (13b) illustrates the average cost of a kNN query on dataset *color* (*mnist*) as a function of k. *LSB-tree* is by far the fastest method, and outperforms all the other approaches by a factor at least an order of magnitude. In particular, on *mnist*, *LSB-tree* even achieves a speedup of two orders of magnitude over *iDistance* (the state of the art of exact NN search), justifying the advantages of approximate retrieval. *LSB-forest* is also much faster than *iDistance*, *MedRank*, and *adhoc-LSH*, especially in returning a large number of neighbors.

The next experiment inspects the result quality of the approximate techniques. Focusing on *color* (*mnist*), Figure 14a (14b) plots the average overall ratios of *MedRank*, *LSB-forest*, and *LSB-tree* as



Cost of *SeqScan* = 2194

(a) *Color*

Cost of *SeqScan* = 3000

(b) *mnist*

Figure 13: Efficiency of kNN search

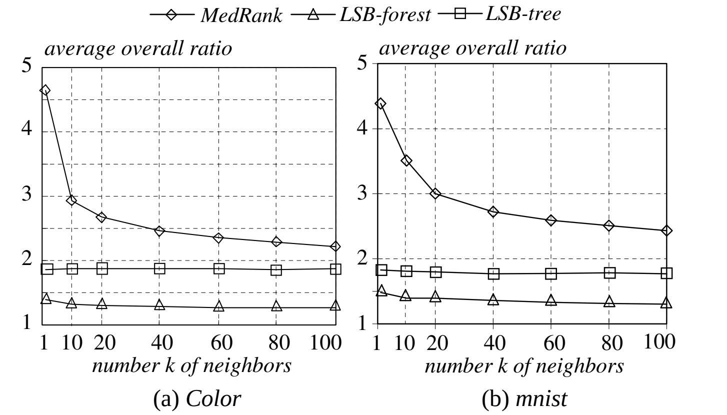
a function of k. Since *adhoc-LSH* may *miss* a query (i.e., unable to return k neighbors), we present its results as a table in Figure 14c, where each cell contains two numbers. Specifically, the number in the bracket indicates how many queries are missed (out of 50), and the number outside is the average overall ratio of the queries that are answered properly. No ratio is reported if *adhoc-LSH* misses more than 30 queries.

LSB-forest incurs low error in all cases (maximum ratio below 1.5), owing to its nice theoretical properties. *LSB-tree* also has good precision (maximum ratio 2), indicating that the proposed conversion (from a d-dimensional point to a Z-order value) adequately preserves the spatial proximity of data points. *MedRank*, in contrast, exhibits much worse precision than the proposed solutions. In particular, observe that *MedRank* is not effective in the important case of single NN search ($k = 1$), for which its average overall ratio can be over 4.5. Finally, *adhoc-LSH* is clearly unreliable due to the large number of queries it misses.

The average overall ratio reflects the general quality of all k neighbors reported. It does not, however, indicate how good the neighbors are at individual ranks. To find out, we set k to 10, and measure the average rank-i ratios at each $i \in [1, 10]$. Figures 15a and 15b demonstrate the results on *color* and *mnist*, respectively (*adhoc-LSH* is not included because it misses many queries). Apparently, both *LSB-forest* and *LSB-tree* provide results that are significantly better than *MedRank* at all ranks. Observe that the quality of *MedRank* deteriorates considerably at high ranks, whereas our solutions return fairly good neighbors even at the greatest rank. Note that the results in Figure 15 should not be confused with those of Figure 14. For example, the average rank-1 ratio (of $k = 10$) is different from the overall average ratio of $k = 1$[5].

Table 3 compares the space consumption of different methods. *LSB-tree* requires a little more space than *iDistance* and *MedRank*, but this is well justified by its excellent query efficiency and accuracy. Remember that both *LSB-tree* and *iDistance* support efficient data insertions/deletions, because they require updating only a single B-tree. *MedRank*, however, entails expensive update overhead because, as mentioned in Section 8.2, inserting/deleting a single point demands modifying $\log n$ B-trees, where n is the dataset car-

[5] The average rank-1 ratio is lower because processing a query with $k = 10$ needs to access more data than a query with $k = 1$, and therefore, has a better chance of encountering the nearest neighbor.

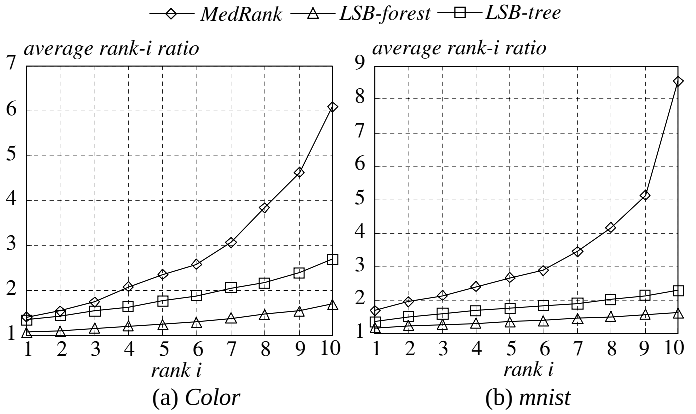


(a) *Color* (b) *mnist*

k	1	10	20	40	60	80	100
color	1.2 (0)	1.3 (30)	- (42)	- (46)	- (46)	- (47)	- (48)
mnist	1.2 (0)	1.3 (13)	1.3 (19)	1.4 (28)	- (37)	- (39)	- (41)

(c) Results of *adhoc-LSH* (in each cell, the number inside the bracket is the number of missed queries, and the number outside is the average overall ratio of the queries answered properly)

Figure 14: Average overall ratio vs. *k*



(a) *Color* (b) *mnist*

Figure 15: Average ratios at individual ranks for 10NN queries

dinality.

	iDistance	*MedRank*	*adhoc-LSH*	*LSB-forest*	*LSB-tree*
color	14	17	1,503	1,503	32
mnist	18	19	1,746	1,746	32

Table 3: Space consumption on real data in mega bytes

Recall that *LSB-forest* utilizes a large number l of LSB-trees, where the number l equals 47 and 55 for *color* and *mnist*, respectively. *LSB-tree* represents the other extreme that uses only a single tree. Next, we explore the compromise of these two extremes, by using multiple, but less than l, trees. The query algorithm is the same as the one adopted by *LSB-tree*. In general, leveraging x trees increases the query, space, and update cost by a factor of x. The benefit, however, is that a larger x also improves the quality of results. To explore this tradeoff, Figure 16 shows the average overall ratio of 10NN queries on the two real datasets, when x grows from 1 to the corresponding l of *LSB-forest*. Interestingly, the precision improves dramatically with just a small number of trees. In other words, we can obtain much better results without increasing the space or query overhead considerably, which is especially appealing for datasets that are not updated frequently.

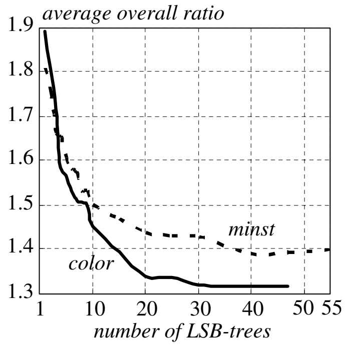


Figure 16: Benefits of using multiple LSB-trees (*k* = 10)

In summary, the exact solution *iDistance* is not adequate due to its costly query time. *Adhoc-LSH* is not reliable because it fails to report enough neighbors for many queries. Furthermore, it also entails large query overhead. *MedRank* is even more expensive than *adhoc-LSH*, and is not friendly to updates. On the other hand, *LSB-forest* guarantees high quality of results, and sub-linear query cost for any data/query distribution. Overall the best solution is the *LSB-tree*, which demands only linear space, permits fast updates, offers very good results, and is extremely efficient in query processing.

9. CONCLUSIONS

Nearest neighbor search in high dimensional space finds numerous applications in a large number of disciplines. This paper develops an access method called the LSB-tree that enables efficient approximate NN queries with excellent result quality. This structure carries both theoretical and practical significance. In theory, it dramatically improves the (asymptotical and actual) space and query efficiency of the previous LSH implementations, without compromising the result quality. In practice, it is faster than the state of the art of exact NN retrieval by two orders of magnitude. Compared to the best existing approach for approximate NN queries, our technique requires only a fraction of its query overhead, and produces results of considerably better quality. Furthermore, the LSB-tree consumes space linear to the dataset cardinality, supports updates efficiently, and can be easily incorporated in relational databases.

Acknowledgements

Yufei Tao and Cheng Sheng were supported by Grants GRF 1202/06, GRF 4161/07, and GRF 4173/08 from HKRGC. Ke Yi was supported by Hong Kong Direct Allocation Grant DAG07/08.

REFERENCES

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS*, pages 459–468, 2006.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[3] V. Athitsos, M. Potamias, P. Papapetrou, and G. Kollios. Nearest neighbor retrieval using distance-based hashing. In *ICDE*, pages 327–336, 2008.

[4] M. Bawa, T. Condie, and P. Ganesan. Lsh forest: self-tuning indexes for similarity search. In *WWW*, pages 651–660, 2005.

[5] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[6] K. P. Bennett, U. Fayyad, and D. Geiger. Density-based indexing for approximate nearest-neighbor queries. In *SIGKDD*, pages 233–243, 1999.

[7] S. Berchtold, C. Bohm, H. V. Jagadish, H.-P. Kriegel, and J. Sander. Independent quantization: An index compression technique for high-dimensional data spaces. In *ICDE*, pages 577–588, 2000.

[8] S. Berchtold, D. A. Keim, H.-P. Kriegel, and T. Seidl. Indexing the solution space: A new technique for nearest neighbor search in high-dimensional space. *TKDE*, 12(1):45–57, 2000.

[9] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *ICDT*, pages 217–235, 1999.

[10] C. Bohm. A cost model for query processing in high dimensional data spaces. *TODS*, 25(2):129–178, 2000.

[11] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: Identifying density-based local outliers. In *SIGMOD*, pages 93–104, 2000.

[12] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *VLDB*, pages 397–410, 1999.

[13] C.-M. Chen and Y. Ling. A sampling-based estimator for top-k query. In *ICDE*, pages 617–627, 2002.

[14] P. Ciaccia and M. Patella. Pac nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In *ICDE*, pages 244–255, 2000.

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

[16] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *SIGMOD*, pages 301–312, 2003.

[17] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[18] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. E. Abbadi. Approximate nearest neighbor searching in multimedia databases. In *ICDE*, pages 503–511, 2001.

[19] P. Ferragina and R. Grossi. The string b-tree: a new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.

[20] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[21] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[22] J. Goldstein and R. Ramakrishnan. Contrast plots and p-sphere trees: Space vs. time in nearest neighbour searches. In *VLDB*, pages 429–440, 2000.

[23] S. Har-Peled. A replacement for voronoi diagrams of near linear size. In *FOCS*, pages 94–103, 2001.

[24] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2):265–318, 1999.

[25] M. E. Houle and J. Sakuma. Fast approximate similarity search in extremely high-dimensional data sets. In *ICDE*, pages 619–630, 2005.

[26] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.

[27] 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. *TODS*, 30(2):364–397, 2005.

[28] F. Korn, B.-U. Pagel, and C. Faloutsos. On the 'dimensionality curse' and the 'self-similarity blessing'. *TKDE*, 13(1):96–111, 2001.

[29] N. Koudas, B. C. Ooi, H. T. Shen, and A. K. H. Tung. Ldc: Enabling search by partial distance in a hyper-dimensional space. In *ICDE*, pages 6–17, 2004.

[30] R. Krauthgamer and J. R. Lee. Navigating nets: simple algorithms for proximity search. In *SODA*, pages 798–807, 2004.

[31] C. Li, E. Y. Chang, H. Garcia-Molina, and G. Wiederhold. Clustering for approximate similarity search in high-dimensional spaces. *TKDE*, 14(4):792–808, 2002.

[32] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The tv-tree: An index structure for high-dimensional data. *The VLDB Journal*, 3(4):517–542, 1994.

[33] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *VLDB*, pages 950–961, 2007.

[34] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *SODA*, pages 1186–1195, 2006.

[35] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.

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