# Efficient and Accurate Nearest Neighbor and Closest Pair Search in High-Dimensional Space

YUFEI TAO
Chinese University of Hong Kong
KE YI
Hong Kong University of Science and Technology
CHENG SHENG
Chinese University of Hong Kong
and
PANOS KALNIS
King Abdullah University of Science and Technology

**20**

Nearest Neighbor (NN) search in high-dimensional space is an important problem in many applications. From the database perspective, a good solution needs to have two properties: (i) it can be easily incorporated in a relational database, and (ii) its query cost should increase *sublinearly* with the dataset size, regardless of the data and query distributions. *Locality-Sensitive Hashing* (LSH) is a well-known methodology fulfilling both requirements, but its current implementations either incur expensive space and query cost, or abandon its theoretical guarantee on the quality of query results.

Motivated by this, we improve LSH by proposing an access method called the *Locality-Sensitive B-tree* (LSB-tree) to enable fast, accurate, high-dimensional NN search in relational databases. The combination of several LSB-trees forms a *LSB-forest* that has strong quality guarantees, but improves dramatically the efficiency of the previous LSH implementation having the same guarantees. In practice, the LSB-tree itself is also an effective index which consumes linear space,

supports efficient updates, and provides accurate query results. In our experiments, the LSB-tree was faster than: (i) iDistance (a famous technique for exact NN search) by two orders of magnitude, and (ii) MedRank (a recent approximate method with nontrivial quality guarantees) by one order of magnitude, and meanwhile returned much better results.

As a second step, we extend our LSB technique to solve another classic problem, called Closest Pair (CP) search, in high-dimensional space. The long-term challenge for this problem has been to achieve *subquadratic* running time at very high dimensionalities, which fails most of the existing solutions. We show that, using a LSB-forest, CP search can be accomplished in (worst-case) time significantly lower than the quadratic complexity, yet still ensuring very good quality. In practice, accurate answers can be found using just two LSB-trees, thus giving a substantial reduction in the space and running time. In our experiments, our technique was faster: (i) than distance browsing (a well-known method for solving the problem exactly) by several orders of magnitude, and (ii) than D-shift (an approximate approach with theoretical guarantees in low-dimensional space) by one order of magnitude, and at the same time, outputs better results.

---

## 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 article, we consider *high-dimensional* NN search. Some studies [Beyer et al. 1999] argue that high-dimensional NN queries may not be meaningful. On the other hand, there is also evidence [Bennett et al. 1999] 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 [Andoni and Indyk 2006; Athitsos et al. 2008; Ciaccia and Patella 2000; Datar et al. 2004; Fagin et al. 2003; Ferhatosmanoglu et al. 2001; Gionis et al. 1999; Goldstein and Ramakrishnan 2000; Har-Peled 2001; Houle and Sakuma 2005; Indyk and Motwani 1998; Li et al. 2002; Lv et al. 2007; Panigrahy 2006].

*Sequential scan* trivially solves a NN query by examining the entire dataset $\mathcal{D}$, but its cost grows linearly with the cardinality of $\mathcal{D}$. From the database perspective, a good solution should satisfy two requirements: (i) it can be easily implemented in a relational database, and (ii) its query cost should increase *sublinearly* with the cardinality for *all* data and query distributions. Despite the bulk of NN literature (see Section 8), with a single exception to be explained

shortly, we are not aware of any existing solution that is able to fulfill both requirements at the same time. Specifically, a majority of them (e.g., those based on new indexes [Arya et al. 1998; Goldstein and Ramakrishnan 2000; Har-Peled 2001; Houle and Sakuma 2005; Lin et al. 1994]) demand nonrelational features, and thus cannot be incorporated in a commercial system. There also exist relational solutions (such as iDistance [Jagadish et al. 2005] and MedRank [Fagin et al. 2003]) which are experimentally shown to perform well for some datasets and queries. Their drawback is that they may incur expensive query cost on other datasets.

*Locality-Sensitive Hashing* (LSH) is the only known solution that satisfies 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, $\|o, q\| \leq c\|o^*, q\|$, where $c \geq 1$ is the *approximation ratio*. It is widely recognized that approximate NNs already fulfill the needs of many applications [Andoni and Indyk 2006; Arya et al. 1998; Athitsos et al. 2008; Datar et al. 2004; Ferhatosmanoglu et al. 2001; Gionis et al. 1999; Har-Peled 2001; Houle and Sakuma 2005; Indyk and Motwani 1998; Krauthgamer and Lee 2004; Li et al. 2002; Lv et al. 2007; Panigrahy 2006]. LSH was originally proposed as a theoretical method [Indyk and Motwani 1998] with attractive asymptotical space and query performance. As elaborated in Section 3, its practical implementation can be either rigorous or ad hoc. Specifically, *rigorous-LSH* ensures good quality of query results (i.e., small approximation ratio $c$), but requires expensive space and query cost. Although *ad hoc-LSH* is more efficient, it abandons quality control, that is, 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 ad hoc-LSH, without sharing their shortcomings. Specifically, the LSB-forest has the following features. First, its space consumption is the same as ad hoc-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 ad hoc-LSH has no such guarantee). Third, its query cost is substantially lower than ad hoc-LSH, and as an immediate corollary, sublinear 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 replicating 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 compared the LSB-tree to iDistance [Jagadish et al. 2005], which is a famous technique for exact NN search, and to MedRank [Fagin et al. 2003], which is a recent approximate method with nontrivial quality

guarantees. The LSB-tree outperformed iDistance by two orders of magnitude, well confirming the advantage of approximate retrieval. Compared to MedRank, our technique was consistently superior in both query efficiency and result quality. Specifically, the LSB-tree was faster by one order of magnitude, and at the same time, returned neighbors with much better quality.

As a second step, we tackle another classic problem, called *Closest Pair* (CP) search, in high-dimensional space. Here, given a set $\mathcal{D}$ of points, the goal is to find two points whose distance is the smallest among all pairs of points in $\mathcal{D}$. This problem has abundant applications in geographic information systems, clustering, and numerous matching problems (such as stable marriage [Wong et al. 2007]), and has been very well solved in low-dimensional space [Corral et al. 2000; Hjaltason and Samet 1998; Lenhof and Smid 1992]. When the dimensionality increases, the challenge has been to achieve *subquadratic* running time, namely, faster than the naive approach that simply examines each pair of points in $\mathcal{D}$. Algorithms that work well in low-dimensional space generally see their computation cost quickly climb to quadratic even at a moderate dimensionality. The *c*-approximate version of the CP problem is to return a pair of points with distance at most *c* times the distance of the closest pair. The dimensionality curse haunts this approximate version, too. For example, when the dimensionality can be viewed as a constant, Lopez and Liao [2000] propose an algorithm, which we call D-shift, with a constant approximation ratio (i.e., $c = O(1)$). As the dimensionality grows, however, their approximation ratio increases superlinearly, and thus becomes unattractive very soon.

We conquer the preceding challenge in this article by giving an algorithm that runs in time significantly lower than the quadratic complexity and meanwhile, gives a very good worst-case guarantee on the quality of results (approximation ratio around 2), regardless of the dimensionality. As in the NN context, although such nice theoretical performance demands a full LSB-forest, in practice only 2 LSB-trees are already sufficient to return accurate results, thus substantially reducing the space and query cost. In the experiments, we compared the proposed algorithms against distance browsing [Corral et al. 2000], which is a well-cited exact solution, and the D-shift algorithm mentioned earlier. Our technique was faster than distance browsing by several orders of magnitude, and than D-shift by one order of magnitude. Moreover, our solutions returned much more accurate answers than D-shift.

The rest of the article is organized as follows. Section 2 presents the problem settings and our objectives. Section 3 points out the defects of the existing LSH implementations. Section 4 explains the construction and NN search algorithms of the LSB-tree, and Section 5 establishes its performance guarantees. Section 6 extends the LSB-tree to provide additional trade-offs between space/query cost and the quality of query results. Section 7 explains how to use LSB-trees for closest pair search. Section 8 reviews the previous work directly related to ours. Section 9 contains an extensive experimental evaluation. Finally, Section 10 concludes the article 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 [Datar et al. 2004; Gionis et al. 1999; Indyk and Motwani 1998], 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 $\ell_p$ norm, which has extensive applications in machine learning, physics, statistics, finance, and many other disciplines. Moreover, as $\ell_p$ 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 2 distinct values), $\ell_1$ 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.

The main problem studied is *c-approximate NN search*, 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. A direct extension of NN queries is *kNN search*, which finds the $k$ points in $D$ closest to $q$. The $c$-approximate version of $k$NN search aims at returning $k$ points, where the $i$th ($1 \leq i \leq k$) one is a $c$-approximation of the real $i$th nearest neighbor. Formally, let $o_1^*, ..., o_k^*$ be the real $k$ NNs in ascending order of their distances to $q$. Then, a set of points $o_1, ..., o_k$ (also sorted in the same way) is a *c-approximate answer* if $\|o_i, q\| \leq c\|o_i^*, q\|$ for all $i \in [1, k]$.

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 or real number 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 sublinear cost growth with $n$ incur I/O complexity $\Omega(dn/B)$. Our objective is to design a *relational* solution beating this complexity.

The second problem solved in this article is *c-approximate CP search*. Specifically, let us define the closest pair in $\mathcal{D}$ to be the pair of points $(o_1^*, o_2^*)$ having the minimum distance among all pairs of points in $\mathcal{D}$. Then the goal of $c$-approximate CP search is to return a pair of points $(o_1, o_2)$ in $\mathcal{D}$ whose distance is at most $c$ times the distance of the closest pair, namely, $\|o_1, o_2\| \leq c\|o_1^*, o_2^*\|$. A naive solution examines all pairs of points, and thus, has time complexity quadratic to $n$. Note that the CP problem has a *bichromatic* counterpart which includes two datasets $\mathcal{D}_1$ and $\mathcal{D}_2$. Here, the exact answer $(o_1^*, o_2^*)$ is the one with the smallest distance in the Cartesian product $\mathcal{D}_1 \times \mathcal{D}_2$, and a $c$-approximate
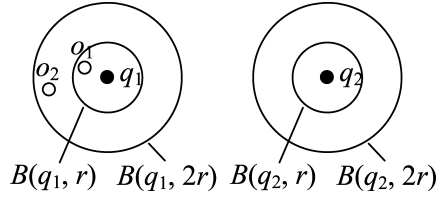
Fig. 1. Illustration of ball cover queries.

answer is a pair $(o_1, o_2) \in \mathcal{D}_1 \times \mathcal{D}_2$ such that $\|o_1, o_2\| \leq c\|o_1^*, o_2^*\|$. These two CP problems can also be extended to *kCP search*, whose $c$-approximate version can be defined in the same fashion as $c$-approximate $k$NN.

We denote by $M$ the amount of available memory, measured in number of words. Unless specifically stated, $M$ can be as small as $3B$ for our algorithms to work (i.e., there are at least 3 memory pages). This, however, excludes the memory needed to store the query results. Specifically, a set of $k$NN or $k$CP result requires $O(kd)$ extra words in memory, which we assume can be afforded.

Our theoretical analysis assumes that a point can fit in a constant number of disk pages (i.e., $d = O(B)$), which is almost always true in reality. For instance, we may set the constant to 10, thus comfortably supporting dimensionality up to $10B$. Also, 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 [Indyk and Motwani 1998] 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 result:

(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 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 $o_1$
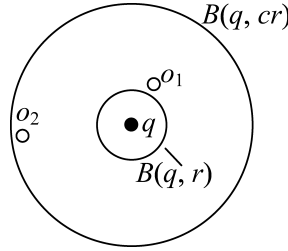
Fig. 2.   The rationale of the reduction from nearest neighbor to ball cover queries.

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 $r$ [Har-Peled 2001; Indyk and Motwani 1998]. 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 $q$.

Based on this idea, Indyk and Motwani [1998] propose a structure that supports $c$-approximate BC queries at $r = 1, c, c^2, c^3, \ldots, x$ 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 [Indyk and Motwani 1998] 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 [Har-Peled 2001; Indyk and Motwani 1998] with better complexities. However, those reductions are highly theoretical, and are difficult to implement in relational databases.

## 3.2 Ad Hoc-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. [1999] remedy this drawback with a heuristic approach which we refer to as ad hoc-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 predetermined by the system. Since only one radius needs to be supported, ad hoc-LSH improves rigorous-LSH by requiring only $O((dn/B)^{1+1/c})$ space and $O((dn/B)^{1/c})$ query time.
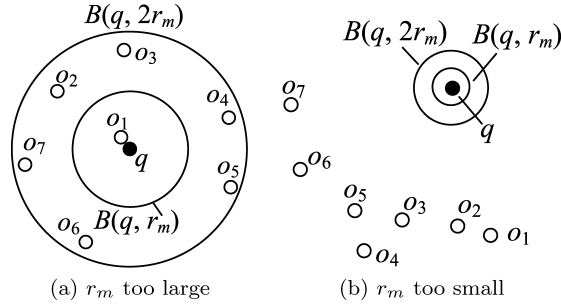
Fig. 3.   Drawbacks of *ad hoc-LSH*.
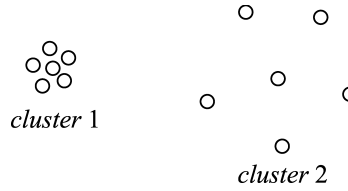


Fig. 4.   No good $r_m$ exists if clusters have different densities.

Unfortunately, the cost saving of ad hoc-LSH trades away the quality control on query results. To illustrate, consider Figure 3(a), where the dataset $\mathcal{D}$ has 7 points $o_1, o_2, \ldots, o_7$, and the black point is a NN query $q$. Suppose that ad hoc-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 3(a) 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 preceding problem is caused by an excessively large $r_m$. Conversely, if $r_m$ is too small, ad hoc-LSH may not return any result at all. To see this, consider Figure 3(b). 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, ad hoc-LSH reports nothing too, and is said to have *missed* the query [Gionis et al. 1999].

Ad hoc-LSH performs well if $r_m$ is roughly equivalent to the distance between $q$ and its exact NN, which is why ad hoc-LSH can be effective when given the right $r_m$. Unfortunately, finding such an $r_m$ is nontrivial. Even worse, such $r_m$ may not exist at all because 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 [Breunig et al. 2000].

Recently, Lv et al. [2007] present a variation of ad hoc-LSH with less space consumption. This variation, however, suffers from the same drawback (i.e., no quality control) as ad hoc-LSH, and entails higher query cost than ad hoc-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, we have the following.

*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 that follow:

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 $\ell_p$ norm, a popular LSH function is defined as follows [Datar et al. 2004].

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

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 [Datar et al. 2004]; $\vec{a} \cdot \vec{o}$ denotes the dot product of these two vectors. $w$ is a sufficiently large constant, and finally, $b$ is uniformly drawn from $[0, w)$.

Eq. (1) has a simple geometric interpretation. To illustrate, let us consider $p = 2$, that is, $\ell_p$ 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 $w = 4$ [Datar et al. 2004]. 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 $B$.

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
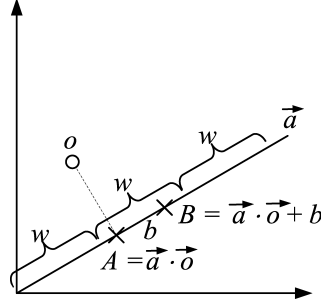
Fig. 5.   Geometric interpretation of LSH.

to be projected into different intervals. The following is proved in Datar et al. [2004].

LEMMA 1 (PROVED IN DATAR ET AL. [2004]).   *Eq. (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 $\ell_2$ norm but the extension to arbitrary $\ell_p$ 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 that the $G(o_1)$ and $G(o_2)$ of two faraway points $o_1, o_2$ 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

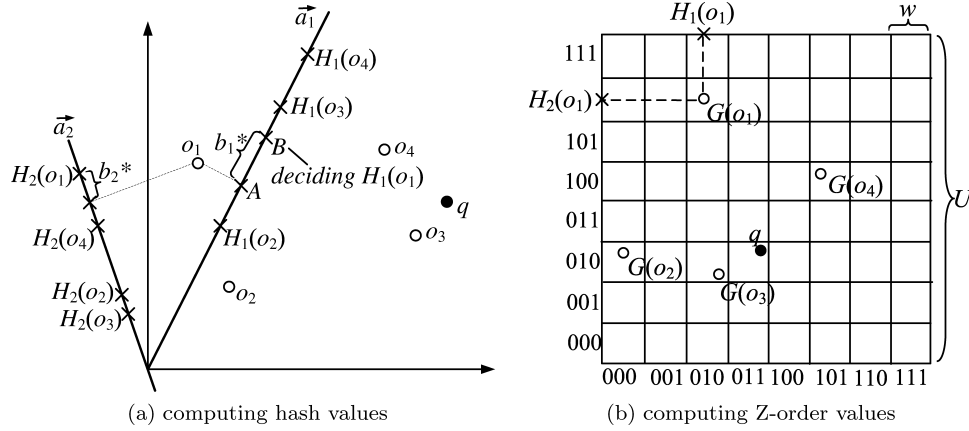(a) computing hash values          (b) computing Z-order values

Fig. 6.   Illustration of data conversion.

uniformly distributed in $[0, 2^f w)$, where $w$ is any constant at least 4, and

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

Recall that $t$ is the largest coordinate on each dimension. Note that while $\vec{a}$ and $w$ are the same as in Eq. (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 $H_1(.), \ldots, H_m(.)$ independently from the family described by Eq. (3). Then, $G(o)$ is the $m$-dimensional vector.

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

*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 hypergrid partitioning the space. We impose a grid where each cell is a hypersquare 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 [Gaede and Gunther 1998]. Let $u = \log(U/w)$. 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 $o_1, o_2, \ldots, o_4$ as shown in Figure 6(a). 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 6(a), 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 $b_2^*$).

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 6(b), and derive $z(o_1)$ as the Z-order value of point $G(o_1)$ in this space. In Figure 6(b), the Z-order calculation is based on a $8 \times 8$ grid. As $G(o_1)$ falls in a cell whose (binary) horizontal and vertical labels are 010 and $\underline{1}1\underline{0}$ respectively, $z(o_1)$ equals $0\underline{1}11\underline{0}0$ (in general, a Z-order value interleaves the bits of the two labels, starting from the most significant bits [Gaede and Gunther 1998]).

*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) $U/w \geq 2^f$, and (ii) $|H_i(o)|$ is confined to at most $U/2$ for any $i \in [1, m]$.

In the form of Eq. (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 $\ell_1$ 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 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,
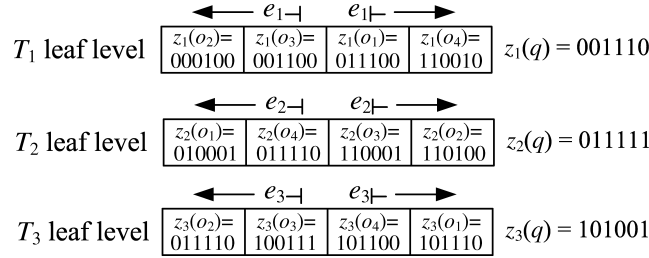
---

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

---

**Algorithm.** *NN1*

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 (the two conditions will be clarified later)
4. return the nearest point found so far

---

Fig. 7. The NN algorithm.

$T_1$ 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 |
|---|---|---|---|

$z_3(q) = 101001$

Fig. 8. Bidirectional expansion ($um = 6, l = 3$).

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.

*Finding the next greatest LLCP.* This can be done by a synchronous bidirectional expansion at the leaf levels of all trees. Specifically, recall that we have obtained the Z-order value $z_j(q)$ in each tree $T_j$ ($1 \le j \le l$). 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 $T_1$, for instance, $z_1(o_1) = 011100$ 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 $z_1(o_1)$.

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, that is, 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 *NN1* terminates when one of two events $E_1$ and $E_2$ happens. The first event is:

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

Event $E_2$ 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.

$E_2$: 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 NN1. Assume that the dataset consists of points $o_1, o_2, \ldots, o_4$ in Figure 6(a), 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 6(b) 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. NN1 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 $E_2$ does not hold. Assuming $E_1$ 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, NN1 finds $o_3$ which replaces $o_4$ as the nearest point so far. As now $\|o_3, q\| = 3 \leq 2^{u-\lfloor v/m \rfloor+1} = 4$, $E_2$ holds, and NN1 terminates by returning $o_3$.

*Retrieving k neighbors.* Algorithm NN1 can be easily adapted to answer $k$NN queries. Specifically, it suffices to modify $E_1$ to "*the total number of leaf entries accessed from all l LSB-trees has reached $(4Bl/d) + (k - 1)l$*", and $E_2$ 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. Finally, the value of $l$ in Eq. (7) needs to be increased by $O(\log(n))$ times. All these changes are to ensure strong quality guarantees in theory for any $k$ (as will be analyzed in the next section). In practice, as long as $k$ is small, only the change to $E_2$ is needed, and $E_1$ and $l$ can remain as they are for $k = 1$.

*kNN 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 $k$NN queries. Accordingly, we slightly modify the algorithm NN1 by simply ignoring event $E_1$ in the terminating condition (as this event is designed specifically for querying $l$ trees). Condition $E_2$, however, is retained. As a trade-off 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 $\mathbb{D}$ the original $d$-dimensional space of the dataset $\mathcal{D}$. Namely, $\mathbb{D} = [0, t]^d$, 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 Eq. (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 Eq. (1).

*Observation* 1.   Given any integer $x \geq 1$, define hash function

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

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

PROOF.   We first point out a useful fact. Imagine a line that has been partitioned into consecutive intervals of length $w$. Let $A$, $B$ be two points on this line with distance $y \leq w$. Shift both points towards right by a distance uniformly drawn from $[0, w\lambda)$, where $\lambda$ is any integer. After this, $A$ and $B$ fall in the same interval with probability $1 - y/w$, which is irrelevant to $\lambda$.

Consider the hash function $h(o)$ in Eq. (1). Use $\vec{a}$ to denote also the line passing the origin containing vector $\vec{a}$. As explained in Section 3.3, $\vec{a} \cdot \vec{o}$ decides a point in Line $\vec{a}$, and $\vec{a} \cdot \vec{o} + b$ shifts the point away from the origin by distance $b$ along the line. Call it the *shifted projection* of $o$. Let us partition line $\vec{a}$ with intervals of length $w$. By Eq. (1), two objects $o_1$, $o_2$ have the same hash value if and only if their shifted projections fall in the same interval.

Now assume that we change the shifting distance from $b$ to $bx$. Since $b$ is uniformly distributed in $[0, w)$, $bx$ is uniformly distributed in $[0, wx)$. Hence, the change does not alter the probability for the shifted projections of $o_1$ and $o_2$ to fall in the same interval. This means that Eq. (8) is also $(1, c, p_1, p_2)$ locality-sensitive with the same $p_1$ and $p_2$ as Eq. (1).   □

For any $s \in [0, f]$ with $f$ given in Eq. (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 Eq. (3). We have the next lemma.

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

PROOF.   Create another space $\mathbb{D}'$ by dividing all coordinates of $\mathbb{D}$ by $2^s$. It is easy to see that the distance of two points in $\mathbb{D}$ is $2^s$ times the distance of their converted points in $\mathbb{D}'$. 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 $\mathbb{D}'$. 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 $\mathbb{D}'$ with $(\ln 1/p_1)/(\ln 1/p_2) \leq 1/2$. Let $o$ be the corresponding point of $o'$ in $\mathbb{D}$. Clearly, $\vec{a} \cdot \vec{o'} = (\vec{a} \cdot \vec{o})/2^s$. Hence $h''(o') = H^*(o, s)$. The lemma thus holds.   □

As shown in Eq. (5), $G(o)$ is composed of hash values $H_1(o), \ldots, H_m(o)$. In the way we obtain $H^*(o, s)$ (Eq. (9)) from $H(o)$ (Eq. (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 $\mathbb{D}$. 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 [Gaede and Gunther 1998], namely, $LLCP(z(o_1), z(o_2)) \ge m(u-s)$. On the other hand, note that a level-*s* tile is a hypersquare 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 $\|q, o\| \le r$, then $LLCP(z(q), z(o)) \ge m(u - \log r)$ with probability at least $p_1^m$.*
(2) *If $\|q, o\| > 2r$, then $LLCP(z(q), z(o)) \ge m(u - \log r)$ with probability at most $p_2^m$.*

*Furthermore, $\frac{\ln 1/p_1}{\ln 1/p_2} \le 1/2$.*

The previous result holds for any LSB-tree. Recall that, for NN search, we need a forest of $l$ trees $T_1, \ldots, T_l$ 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 $\boldsymbol{P_2}$.

$\boldsymbol{P_1}$: $LLCP(z_j(q), z_j(o^*)) \ge m(u - \log r)$ in at least one tree $T_j$ $(1 \le j \le \ell)$.
$\boldsymbol{P_2}$: There are less than $4Bl/d$ leaf entries $z_j(o)$ from all trees $T_j$ $(1 \le j \le l)$ such that: (i) $LLCP(z_j(q), z_j(o)) \ge m(u - \log r)$, and (ii) $o$ is outside $B(q, 2r)$.

The property guaranteed by the $l$ trees is the next lemma.

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 [Gionis et al. 1999] of the correctness of LSH. □

Now we establish an approximation ratio of 4 for algorithm NN1. In the next section, we will extend the LSB-tree to achieve better approximation ratios.

THEOREM 1. *Algorithm NN1 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 $\ell_p$ norm). If when NN1 finishes, it has already found $o^*$ in any tree, apparently it will return $o^*$ which is optimal. Next, we assume NN1 has not seen $o^*$ at termination.

We will show that when both $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ are true, the output of NN1 is definitely 4-approximate. Denote by $j^*$ the $j$ stated in $\boldsymbol{P_1}$. Recall that NN1 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 NN1 visits leaf entries in descending order of their LLCP, the LLCP $v$ of the last fetched leaf entry is at least $LLCP(z_{j^*}(q), z_{j^*}(o^*)) \geq m(u - \log r)$. It follows that $\lfloor v/m \rfloor \geq u - \log r$. $\boldsymbol{E_2}$ 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 NN1 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 the next corollary.

COROLLARY 2. *Let $r^*$ be the distance from $q$ to its real NN. With at least constant probability,* NN1 *returns a point within distance $2r$ to $q$, where $r$ is the lowest power of 2 bounding $r^*$ from above.*

*Remark* 1. When defining the problem in Section 2, we restricted point coordinates to integers. In fact, the preceding analysis holds also for real coordinates as well, as long as the minimum distance between two points in $\mathcal{D}$ is at least 1.

*Remark* 2. As a standard trick in probabilistic algorithms, by repeating our solution $O(\log(1/p))$ times, we boost the success probability of algorithm NN1 from constant to at least $1 - p$, for any arbitrarily low $p > 0$. In other words, by repeating $O(\log n)$ times (namely, increasing $l$ to $O(\log n \sqrt{dn/B})$), the failure probability of NN1 can be lowered to at most $1/2n$. Using the Union Bound inequality (also called the Boole's inequality), it is easy to show that the $k$NN algorithm described in Section 4.2 gives a 4-approximate answer with at least constant probability.

## 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 NN1 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 $u = O(f) = O(\log d + \log t)$ and $m = O(\log(dn/B))$. As $\log d + \log t$ 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 NN1 (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* [Ferragina and Grossi 1999], the height $E$ is bounded by $O(\log_B n)$, resulting in query complexity $O(\sqrt{dn/B}\log_B n)$.

## 5.3 Comparison with Rigorous-LSH

As discussed in Section 3, for 4-approximate NN search, rigorous-LSH consumes $O((\log d + \log t)(dn/B)^{1.5})$ space, and answers a query in $O((\log d + \log t)\sqrt{dn/B})$ 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 $\log d + \log t$. 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 ad hoc-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 $O(\log_B n)$ by resorting to the string B-tree. Even if we simply implement a LSB-tree as a normal B-tree, the height $E$ never grows beyond 6 in our experiments. This is expected to be much smaller than $\log d + \log t$, 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 ad hoc-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.

---

**Algorithm.** *NN2* $(r)$

1.  $o = $ the output of algorithm *NN1* on $F$
2.  $o' = $ the output of algorithm *NN1* on $F'$
3.  return the point between $o$ and $o'$ closer to $q$

---

Fig. 9.    The 3-approximate algorithm.

*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 NN1 (Figure 7) using the query point. Let $o$ by the output of NN1. 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 previous 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 $r = 1, 2, 2^2, \ldots, 2^f$ where $f$ is given in Eq. (4). We build another LSB-forest to handle 2-approximate BC queries with any $r = 1.5, 1.5 \times 2, 1.5 \times 2^2, \ldots, 1.5 \times 2^f$. For this purpose, we can 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 $\mathcal{D}$. The only issue is that the distance of two points in $\mathcal{D}'$ may drop below 1, while our technique requires a lower bound of 1 (see Remark 1 in Section 5.1). This can be easily fixed by scaling up $\mathcal{D}$ first by a factor of 2 (i.e., doubling all the coordinates). Any two points in the new $\mathcal{D}$ have distance at least 2, so any two points in $\mathcal{D}'$ now have distance at least $2/1.5 > 1$.

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 NN2 in Figure 9.

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

PROOF.    Let $\mathbb{D}$ be the $d$-dimensional space of dataset $\mathcal{D}$, and $\mathbb{D}'$ 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 $\mathbb{D}'$ is between $(2^x/1.5, 2^x]$. Hence, by Corollary 2, with at least constant probability the distance between $o'$ and $q$ in $\mathbb{D}'$ 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 $\mathbb{D}$. Similarly, for Case 2, we can show that $o$ (output at line 1) is a 3-approximate NN with at least constant probability.    □

The preceding idea can be easily extended to $(2 + \epsilon)$-approximate NN search for any $0 < \epsilon < 2$. Specifically, we can maintain $1 + \lfloor 1/\log(1 + \epsilon/2) \rfloor$ LSB-forests, such that the $i$th forest $(1 \le i \le 1 + \lfloor 1/\log(1 + \epsilon/2) \rfloor)$ supports
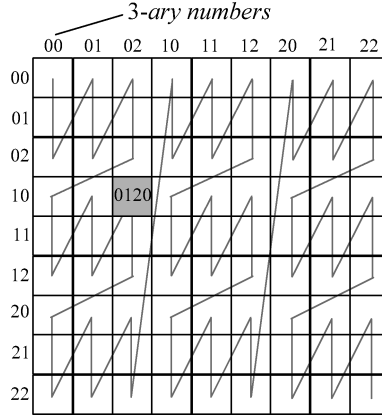
Fig. 10.   Computing Z-order values for the order-3 LSB-tree.

2-approximate BC queries at $r = \alpha, 2\alpha, 2^2\alpha, \ldots, 2^f\alpha$, where $\alpha = (1 + \epsilon/2)^{i-1}$. Given a query $q$, we run algorithm NN1 on all the forests, and return the nearest point found. By an argument similar to proving Theorem 3, we have the next theorem.

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

$(c+\epsilon)$-*approximate nearest neighbors*. In practice, an application may be able to tolerate an approximation ratio higher than that of the basic LSB-forest. In this case, it is possible to further reduce the space and query cost. In the sequel, we generalize the LSB-tree to offer any approximation ratio arbitrarily close to $c$, for any integer $c \geq 3$.

We make several changes in building a LSB-tree.

—Recall that $m$ equals $\log_{1/p_2}(dn/B)$ in Section 4.1. For $c \geq 3$, the expression for $m$ remains identical, but $p_2$ is the constant as given in the Lemma 1 for the value of $c$ we are considering.
—In Eq. (3), $b^*$ will be uniformly drawn from $[0, c^f w)$, where $f$, instead of following Eq. (4), is set to $\lceil \log_c d + \log_c t \rceil$.
—We will decide $U$ (i.e., the axis length of the $m$-dimensional space of $G(o)$) by setting $U/w$ to the smallest power of $c$ that bounds both $c^f$ and $2H_{\max}/w$ from above, where $H_{\max}$ is given in Eq. (5).

The last change lies in the way a Z-order value $z(o)$ is calculated from $G(o)$. Let us denote by $\mathbb{G}$ the $m$-dimensional space where $G(o)$ is distributed. Impose a hypergrid over $\mathbb{G}$ where each cell is a hypersquare with side length $w$. As mentioned earlier, $U/w$ is a power of $c$; therefore, the grid has totally $x^{cm}$ cells, for some integer $x$. Figure 10 shows an example where $\mathbb{G}$ has $m = 2$ dimensions, $c = 3$, and $\mathbb{G}$ is partitioned by a $3^2 \times 3^2$ grid.

We utilize the grid to compute $z(o)$ as follows. Recall that the grid partitions each dimension of $\mathbb{G}$ into $c^x$ intervals. Number these intervals consecutively using $c$-ary values. For instance, in Figure 10, each dimension of $\mathbb{G}$ is cut into $3^2 = 9$ intervals, which are numbered from 00 to 22. Then, the Z-order value of each cell in the grid is obtained by interleaving its $c$-ary digits on all dimensions. For example, the grey cell in Figure 10 is numbered 02 and 10 on the horizontal and vertical dimensions, respectively. Hence, its Z-order value is 0120, taking the first digits of 02 and 10, then followed by their second digits. $z(o)$ equals the Z-order value of the cell that $G(o)$ falls in. By the Z-order values thus calculated, we impose an ordering of the cells as depicted by the zigzag line in the figure.

We call the adapted LSB-tree an *order-c LSB-tree*, and build a forest of $l = (dn/B)^{1/c}$ such trees independently. Call it an *order-c LSB-forest*. The query algorithm NN1 in Section 4.2 can be deployed directly on the forest, except that the number $2^{u-\lfloor v/m \rfloor+1}$ in event $\boldsymbol{E_2}$ should be replaced by $c^{u-\lfloor v/m \rfloor+1}$. By an argument similar to the one in Section 5, we can show the next theorem.

THEOREM 5. *We can build a set of order-c LSB-trees that consume totally $O((dn/B)^{1+1/c})$ space. Given a query,* NN1 *returns a $c^2$-approximate NN in $O(E(dn/B)^{1/c})$ I/Os, where E is the height of an order-c LSB-tree.*

Notice that the order-$c$ LSB-forest captures the basic LSB-forest as a special case with $c = 2$. Recall that a basic LSB-forest is able to answer 2-approximate BC queries with $r$ being powers of 2. Likewise, an order-$c$ LSB-forest is able to answer $c$-approximate BC queries with $r = 1, c, c^2, \dots$. To lower the approximation ratio to $c + \epsilon$, we can build $1 + \lfloor \frac{c}{\log(1+\epsilon/c)} \rfloor$ order-$c$ LSB-forests. Specifically, the $i$th ($1 \leq i \leq 1 + \lfloor \frac{c}{\log(1+\epsilon/c)} \rfloor$) forest is responsible for $c$-approximate BC queries with radius $r = \alpha, c\alpha, c^2\alpha, \dots$, where $\alpha = (1+\epsilon/c)^{i-1}$. Following the way of establishing Theorem 4, we can prove the next theorem.

THEOREM 6. *For any $0 < \epsilon < c^2 - c$, we can build $O(\frac{c}{\log(1+\epsilon/c)})$ order-c LSB-trees that consume totally $O((dn/B)^{1+1/c} \frac{c}{\log(1+\epsilon/c)})$ space, and answer a $(c+\epsilon)$-approximate NN query in $O(E(dn/B)^{1/c} \frac{c}{\log(1+\epsilon/c)})$ I/Os, where E is the height of an order-c LSB-tree.*

Note that, for $c \geq 3$, the complexities in the previous theorem are strictly smaller than those in Theorem 4 because the polynomials in Theorem 6 have lower exponents.

## 7. CLOSEST PAIR SEARCH

In this section we will extend the LSB technique to solve the CP problem. There is a straightforward solution. Specifically, assume that a LSB-forest has been built on dataset $\mathcal{D}$. First, for every point $o \in \mathcal{D}$, run algorithm NN1 (Figure 7) to find its NN $o'$. Then, among all such pairs $(o, o')$, report the one with the smallest distance. This will give us a 4-approximate answer with high probability.

In main memory, the solution is quite efficient, requiring only $O(n^{1.5} \log(n))$ time [Datar et al. 2004]. In external memory where an access unit is a page of $B$ words, the running time becomes $O(n\sqrt{dn/B} \log_B n)$, which can be even worse
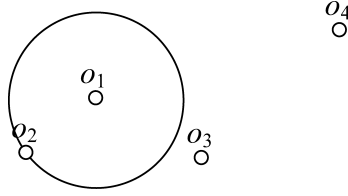
Fig. 11.   Illustration of the ball pair problem.

than the trivial bound $O((dn/B)^2)$. Next, we will propose a different approach that requires only $O((dn/B)^{1.5})$ I/Os. As will be clear shortly, the analysis of this approach's running time is drastically different from that in Datar et al. [2004].

## 7.1 Ball Pair Search

As explained in Section 3.1, LSH approaches NN search with ball cover. Similarly, we attack the CP problem with another problem we call *Ball Pair* (BP) search, which can be regarded as the counterpart of ball cover in the CP context. Formally, given a radius $r$, a *c-approximate BP* query on $\mathcal{D}$ returns the following:

(1) If there is a pair of points in $\mathcal{D}$ with distance at most $r$, return a pair of points in $\mathcal{D}$ with distance at most $cr$.
(2) If no two points in $\mathcal{D}$ have distance at most $cr$, return nothing.
(3) Otherwise, the result is undefined.

For example, consider Figure 11 where $\mathcal{D}$ has 4 points. Let $r$ be the distance between $o_1$ and $o_2$. Then, a 2-approximate BP query must return a pair of points within distance $2r$. In our example, there are two such pairs: $(o_1, o_2)$, $(o_1, o_3)$, either of which is a correct result. On the other hand, for any $r < \frac{1}{2}\|o_1, o_2\|$, a 2-approximate BP query must not return anything at all.

Our discussion will focus on radius $r$ that is a power of 2 between 1 and $2^f$, where $f$ is given in Eq. (4). In the sequel, let $\lambda = \log r$. We will first target an approximation ratio of $c = 2$, and then extend to other ratios later. For $c = 2$, we need a LSB-forest with $l$ trees built in exactly the way described in Section 4.1. Next, we will first clarify the algorithm for BP search, and then analyze its theoretical properties.

*Algorithm.* Let us first concentrate on a single LSB-tree $T$. Remember that each leaf entry carries a Z-order value. Two points $o_1$, $o_2$ are said to be in the same *bucket* if they share the first $m(u - \lambda)$ bits in their Z-order values, namely, $LLCP(z(o_1), z(o_2)) \geq m(u - \lambda)$; see the definitions of $m$, $u$, and $LLCP(.)$ in Section 4. Intuitively, a bucket is essentially a hypersquare with $2^{\lambda m}$ cells in the grid partitioning the space $\mathbb{G}$ (that is used to define Z-values). For example, (same as Figure 6(b)) Figure 12(a) shows a space $\mathbb{G}$ with $m = 2$ dimensions, the coordinates of which are encoded with $u = 3$ bits. For $\lambda = 1$, there are 16 buckets, each with $2^{\lambda m} = 4$ cells that share the same first $m(u - \lambda) = 4$ bits in their Z-order values. Figure 12(b) demonstrates the case of $\lambda = 2$, where there
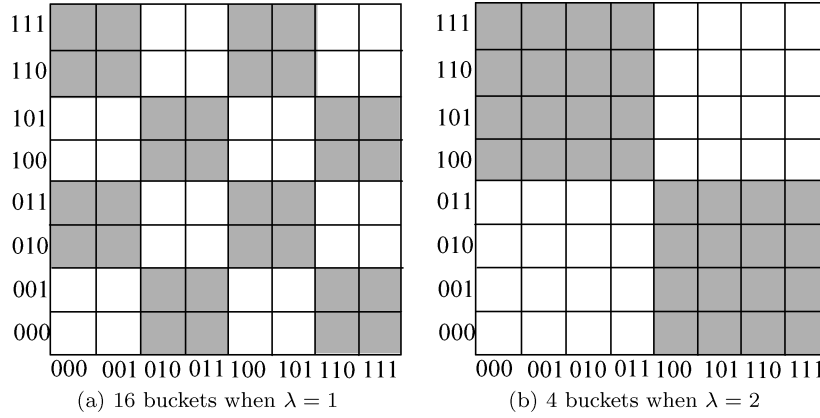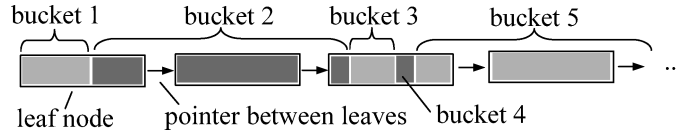
(a) 16 buckets when $\lambda = 1$      (b) 4 buckets when $\lambda = 2$

Fig. 12.   Coverage of buckets in space $\mathbb{G}$ ($m = 2, u = 3$).



Fig. 13.   Buckets at the leaf level of a LSB-tree.

are 4 buckets each with 16 cells. Note that a bucket of $\lambda = 2$ encloses 4 buckets of $\lambda = 1$. This is true in general: every time $\lambda$ grows by 1 (i.e., $r$ doubles), a new bucket covers $2^m$ old buckets. Also notice that, in any case, the cells of a bucket always have continuous Z-order values.

Let us divide the leaf entries of tree $T$ based on the buckets they belong to. Apparently, points of the same bucket must come together in *adjacent* leaf nodes, as illustrated in Figure 13. Note that a bucket may span multiple leaf nodes, but may also be so small that several buckets can fit in a single leaf. In any case, the important fact is that by scanning the leaf level from the leftmost node rightwards, we can easily determine the bucket boundaries, by comparing the Z-order values of consecutive leaf entries.

Now, let us take back the entire forest of $l$ trees, since we are ready to elaborate the algorithm for BP search, which is fairly simple as presented in Figure 14. For each tree, we scan its leaf level from left to right, starting from the leftmost leaf. For every bucket $I$ encountered during the scan, evaluate the distances of all the pairs of points in $I$ bruteforcely in $O(\lceil d|I|/B \rceil^2)$ I/Os. Meanwhile, we keep track of the total number of pairs evaluated (from all trees) so far, namely, the count increases by $|I|(|I|-1)$ after processing a bucket $I$. The algorithm terminates as soon as the count reaches $2Bnl/d$ (where $n$ is the size of $\mathcal{D}$); this includes even the duplicate pairs discovered from different trees. At termination, we return the closest pair (among all the pairs evaluated) if its distance is at most $2r$. Otherwise, we return nothing.

*Analysis.* Next we will first establish the quality guarantee of our algorithm BP, and then analyze its running time.

---

**Algorithm.** $BP(r)$

---

/* assume that the LSB-forest has trees $T_1$, ..., $T_l$ */

1.  for $i = 1$ to $l$
2.      scan from the leaf nodes of $T_i$ rightwards, starting from the leftmost one
3.      for each bucket $I$ encountered
4.          evaluate the $O(|I|^2)$ pairs of points in $I$
5.          break, as soon as $2Bnl/d$ pairs have been evaluated (from all trees)
6.  if the closest pair found so far has distance at most $2r$ then return it
7.  else return nothing

---

Fig. 14.   The BP algorithm.

LEMMA 5.   BP *returns a 2-approximate answer with at least constant probability.*

PROOF.    The proof is an adaptation of the standard LSH analytical framework for NN search; we will focus on the differences in the CP context. Given two points in a bucket of a tree, we say that they form a *bad pair* if their distance is larger than $2r$. In the sequel, we will assume the existence of a pair $(o_1^*, o_2^*)$ within distance at most $r$ (the proof is similar if such a pair does not exist). Observe that $BP$ finds a 2-approximate answer if both of the following hold.

$\boldsymbol{P_1'}$: There are less than $2Bnl/d$ bad pairs in all the $l$ trees in the LSB-forest.
$\boldsymbol{P_2'}$: $(o_1^*, o_2^*)$ appear in at least one bucket of a LSB-tree.

The rest of the proof will show that they hold at the same time with at least constant probability.

Recall that two points $o_1$, $o_2$ fall in the same bucket of some LSB-tree $T_j$ ($1 \leq j \leq l$) if and only if $LLCP(z(o_1), z(o_2)) \geq m(u - \lambda)$. By Corollary 1, if $\|o_1, o_2\| > 2r$, they form a bad pair in $T_j$ with probability at most $p_2^m$. Hence, in all $l$ trees, the expected number of bad pairs is at most $l \cdot n(n - 1)p_2^m$, which is smaller than $Bnl/d$ with the choice of $m$ in Eq. (2). By Markov Inequality, the probability for the total number of bad pairs in all $l$ trees to be at least $2Bnl/d$ is at most $1/2$, that is, $\boldsymbol{P_1'}$ fails with probability at most $1/2$.

By the same reasoning in the standard LSH framework, with the choice of $l$ in Eq. (7), $\boldsymbol{P_2'}$ fails with probability at most $1/e$. Hence, the probability that at least one of $\boldsymbol{P_1'}$ and $\boldsymbol{P_2'}$ fails is bounded by $1/2 + 1/e = 0.87$ from above, implying that they hold with probability at least 0.13.   □

Although the proof of quality generally follows the LSH framework, the running time analysis is substantially different.

LEMMA 6.    BP *performs* $O((dn/B)^{1.5})$ *I/Os.*

PROOF.    We consider only buckets with more than $B/d$ points. Each bucket with at most $B/d$ points fits in at most 2 pages; so examining all pairs of points in each of these buckets takes linear I/Os, namely $O((dn/B)^{1.5})$ I/Os.

Without loss of generality, assume that at the time BP finishes, it has encountered $J$ buckets in this order: $I_1$, $I_2$, ..., $I_J$ (they may come from different trees). Note that, except the last one $I_J$, all the other buckets have been *fully*

processed. Namely, if we denote by $x_i$ ($1 \leq i \leq J - 1$) the size of bucket $I_i$, $I_i$ contributed $x_i(x_i - 1)$ pairs to the count BP is maintaining. As for the last bucket $I_J$, assume that BP scanned $x_J$ points in it. Hence, $I_J$ contributed at least $(x_J - 1)(x_J - 2)$ to the count. It suffices to consider $x_J \geq 3 + B/d$ (otherwise, we can ignore $I_J$ but add only $O(1)$ I/Os to the overall cost). As totally BP evaluates no more than $2Bnl/d$ pairs, we have

$$(x_J - 1)(x_J - 2) + \sum_{i=1}^{J-1} x_i(x_i - 1) \leq 2Bnl/d. \tag{11}$$

Since $x_i \geq 1 + B/d$ ($1 \leq i \leq J - 1$) and $x_J \geq 3 + B/d$, inequality (11) implies

$$\sum_{i=1}^{J} (x_i \cdot B/d) < (x_J - 1)(x_J - 2) + \sum_{i=1}^{J-1} x_i(x_i - 1) \leq 2Bnl/d.$$

Hence,

$$\sum_{i=1}^{J} x_i \leq \frac{2Bnl/d}{B/d} = O(nl). \tag{12}$$

From inequalities (11) and (12), we have

$$\sum_{i=1}^{J} x_i^2 \leq 2Bnl/d + 3 \sum_{i=1}^{J} x_i = O(Bnl/d), \tag{13}$$

where the last equality is due to $d = O(B)$. Furthermore, $J$ satisfies

$$J \leq \frac{\sum_{i=1}^{J} x_i}{B/d} = O(dnl/B). \tag{14}$$

A bucket $I_i$ ($1 \leq i \leq J - 1$) with size $x_i$ occupies at most $O(\lceil dx_i/B \rceil)$ pages. Hence, brute-force examination of all pairs of points in $I_i$ requires $O(\lceil dx_i/B \rceil^2)$ I/Os. Likewise, examining the last bucket $I_J$ takes $O(\lceil dx_J/B \rceil^2)$ I/Os. Thus, the total cost on all buckets is bounded by

$$O\left( \sum_{i=1}^{J} \lceil dx_i/B \rceil^2 \right) = O\left( \sum_{i=1}^{J} (dx_i/B + 1)^2 \right)$$

$$= O\left( J + \frac{d^2}{B^2} \sum_{i=1}^{J} x_i^2 + \frac{d}{B} \sum_{i=1}^{J} x_i \right)$$

which, by inequalities (12) through (14), is bounded by $O(dnl/B) = O((dn/B)^{1.5})$.  □

## 7.2 Solving the Closest Pair Problem

The closest pair problem can be reduced to BP search. A simple approach is to invoke algorithm BP (Figure 14) with doubling radius $r = 1, 2, 4$, and so on, until it returns a pair of points whose distance is at most twice the current $r$. This procedure, referred to as algorithm CP1, is formally presented

---

**Algorithm.** *CP1*

---
1.   $r = 1$
2.   repeat
3.       call $BP(r)$
4.       if the above returns a pair of points $(o_1, o_2)$ and $\|o_1, o_2\| \leq 2r$ then return $(o_1, o_2)$
5.       else $r = 2r$

---

Fig. 15.   The first CP algorithm.

---

**Algorithm.** *CP2*

---
1.   find an appropriate radius $r_{good}$
2.   call $BP(r_{good}/2)$
3.   if the above returns a pair of points with distance greater than $r_{good}$
4.       call $BP(r_{good})$
5.   return the closest of all pairs of points examined

---

Fig. 16.   An improved CP algorithm.

in Figure 15, which can be easily shown to return a 4-approximate answer with at least constant probability.

The drawback of CP1 is that its running time may be $O((\log d + \log t)(dn/B)^{1.5})$ in the worst case, where $t$ is the maximum coordinate of a dimension. In the sequel, we give an alternative algorithm that requires only $O((dn/B)^{1.5})$ time, that is, eliminating the $\log d + \log t$ factor.

*An improved algorithm.* We refer to our new algorithm as CP2. Unlike CP1 that performs multiple BP search, CP2 first picks an appropriate value of $r$, denoted as $r_{good}$, and then, performs at most *two* BP search at $r = r_{good}/2$ and $r_{good}$, respectively. As shown later, $r_{good}$ can be found in $O((dn/B)^{1.5})$ I/Os, which is the same cost of one BP search, thus making the overall cost $O((dn/B)^{1.5})$ as well. CP2 is presented in Figure 16. Next we will focus on explaining line 1.

Recall that, given a particular $r$, algorithm BP examines a number of point pairs in $\mathcal{D}$. Let us denote the number as $C(r)$. Obviously, $C(r)$ is always bounded from above by $2Bnl/d$, as it is the largest number of pairs evaluated by BP. It is fairly simple to obtain the exact $C(r)$ by reading (from left to right) the leaf levels of all LSB-trees once as follows. First, set $C(r)$ to 0, and start reading the first tree. At any time, we keep a count $x$ of how many points have been seen in the current bucket being scanned. When the boundary of the bucket is reached, we add $x(x - 1)$ to $C(r)$, and then, reset $x$ to 0 for the next bucket. At the time all trees have been scanned, $C(r)$ becomes final. Since every leaf node of each tree is read once, the total cost is $O((dn/B)^{1.5})$ I/Os.

A nice feature of the preceding strategy is that it needs to store only two values in memory at any time: $C(r)$ and $x$. There are, however, merely $f = \lceil \log d + \log t \rceil$ different values of $r$. Hence, we can compute the $C(r)$ of all possible $r$ in a *single* pass. The memory size required is only one memory page (as the reading buffer) plus $2f$ integers! Then, $r_{good}$ is decided as

$$r_{good} = \min\{r \mid C(r) \geq 2Bnl/d\}, \tag{15}$$

namely, $r_{good}$ is the lowest $r$ such that $C(r) \geq 2Bnl/d$.

THEOREM 7. *Algorithm* CP2 *returns a 4-approximate answer with at least constant probability.*

PROOF.    We will make a claim $X$: every pair whose distance is evaluated by CP1 is also evaluated by CP2. Under the claim, CP2 never returns a worse answer than CP1, which will establish the theorem.

Assume that when CP1 finishes, the value of $r$ is $r'$. Clearly, $r' \leq r_{good}$ because algorithm BP never evaluates more than $2Bnl/d$ pairs. If $r' = r_{good}$, it means that the best pair CP1 returns is found by the last BP search, namely, $BP(r')$. Then, $X$ is true because CP2 also needs to perform the same BP search $BP(r_{good})$ (notice that line 4 of CP2 will definitely be executed, that is, the if-condition at line 3 will fail).

Now consider $r' < r_{good}$. The crucial fact is that, for any $r_1 < r_2 < r_{good}$ where $r_1$ and $r_2$ are powers of 2, the set of point pairs $BP(r_1)$ evaluates is always a subset of the set $BP(r_2)$ evaluates, due to the selection of $r_{good}$. Hence, whatever is evaluated by $BP(r')$ is also evaluated by $BP(r_{good}/2)$. Hence, $X$ also holds.    □

Thus, we arrive at the next theorem.

THEOREM 8.    *Given a LSB-forest, we can perform 4-approximate closest pair search in* $O((dn/B)^{1.5})$ *I/Os as long as the memory size M is at least* $\max\{3B, B + 2f\}$ *words.*

Note that the value of $f$ is smaller than $B$ in practice. As a reference, let $d = 1000$ and $t = 10^{10}$; in this case, $f < 44$, which means $2f$ integers can easily fit in a page of $B = 1024$ words. Thus, the memory size needed to run CP2 is only 3 pages. In case the LSB-forest does not exist in advance, then the total time increases by a factor of $\log_{M/B}(dn)$, because building all the leaf levels with external sort takes $O((dn/B)^{1.5} \log_{M/B}(dn))$ I/Os (the nonleaf levels are unnecessary).

*Extensions.* In theory, $k$CP search can also be supported in a way similar to how $k$NN is handled. First, we need to increase $l$ to $O(\sqrt{dn/B} \log n)$. Second, the limit on how many point pairs are evaluated by algorithm BP should be raised to $2Bnl/d + (k-1)l$. In practice when $k$ is small, no change is required, and we can simply output the $k$ closest pairs among all the pairs CP2 evaluates during its execution.

So far we have been targeting an approximation ratio of 4, but the ratio can be improved to arbitrarily close to 2 with only a constant blowup in the computation cost. Conversely, one may also opt for a higher ratio as a trade-off for lower running time. Using the methods explained in Section 6, for any integer $c \geq 2$ and arbitrary $\epsilon$ satisfying $0 < \epsilon < c^2 - c$, we can find a $(c + \epsilon)$-approximate answer in $O((dn/B)^{1+1/c} \frac{c}{\log(1+\epsilon/c)})$ I/Os. If the LSB-forests need to be built on-the-fly, the cost is $O(\log_{M/B}(dn))$ times higher.

All the results can be extended to bichromatic CP search as well. In particular, for $k = 1$, a 4-approximate closest pair between $\mathcal{D}_1$ and $\mathcal{D}_2$ can be found
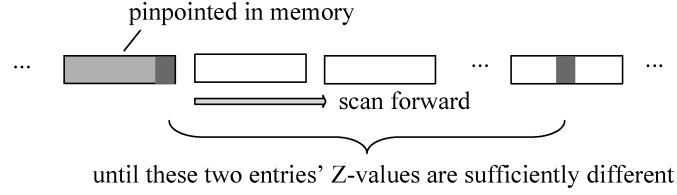
Fig. 17. CP search with only one LSB-tree.

with at least constant probability in

$$O\left(\sqrt{d\sqrt{n_1 n_2}/B} \cdot \left((dn_1/B)\log_{M/B}(dn_1) + (dn_2/B)\log_{M/B}(dn_2)\right)\right)$$

I/Os, where $n_1$ and $n_2$ are the cardinalities of the participating datasets $\mathcal{D}_1$ and $\mathcal{D}_2$, respectively. Note that when $n_1 = n_2 = n$, this complexity degenerates into the one obtained earlier for a single dataset.

*Using a single LSB-tree.* As mentioned in Section 4.2, a practical application may choose to maintain only a single LSB-tree, because this consumes only linear space and allows logarithmic update time. Before finishing this section, we give an algorithm, referred to as CP3, which performs approximate $k$CP search using only such a tree.

The rationale behind CP3 is that a LSB-tree generally captures the proximity of the points in the original space. Namely, if points $o_1$ and $o_2$ are close, they tend to have similar Z-values $Z(o_1)$ and $Z(o_2)$. Hence, for each leaf entry, we will evaluate its distances only to its nearby leaf entries. More specifically, at any time, we pinpoint a leaf node $N$ in memory. After computing the distances of all pairs of points in $N$, we use another memory page $N'$ to scan forward the subsequent leaf pages one by one. Every point in $N'$ has its distances to all points in $N$ computed. This continues until the Z-order value of an entry in $N'$ is "sufficiently faraway" (to be elaborated shortly) from that of the rightmost entry in $N$ (see Figure 17). When this happens, we move $N$ to the leaf node on its right, and repeat the process. The first $N$ pinpointed is the leftmost leaf node.

It remains to clarify what we mean by "two entries $Z(o_1)$ and $Z(o_2)$ are faraway." We adopt a heuristic similar to the one used in Section 4.2 for $k$NN search. Specifically, let *dist* be the distance of the $k$th closest pair of points CP3 has discovered so far. We rule that $Z(o_1)$ and $Z(o_2)$ are faraway if

$$dist \leq 2^{u-\lfloor LLCP(Z(o_1),Z(o_2))/m \rfloor + 1},$$

where $u$, $m$, and $LLCP(.,.)$ are as defined in Section 4. The algorithm CP3 is formally presented in Figure 18.

## 8. RELATED WORK

NN search is well understood in low-dimensional space [Hjaltason and Samet 1999; Roussopoulos et al. 1995]. This problem, however, becomes much more difficult in high-dimensional space. Many algorithms (e.g., those based on data or space partitioning indexes [Gaede and Gunther 1998]) that perform nicely

| **Algorithm.** *CP3* |
| --- |
| 1.  $N$ = the leftmost leaf node |
| 2.  repeat |
| 3.      compute the distances of all pairs of points in $N$ |
| 4.      $N'$ = the leaf node to the right of $N$ |
| 5.      repeat |
| 6.          compute the distance of each point in $N'$ to each point in $N$ |
| 7.          if an entry in $N'$ is sufficiently faraway from the rightmost entry of $N$ |
| 8.              break |
| 9.          else $N'$ = the leaf node to the right of $N'$ |
| 10.     until $N' = \emptyset$ |
| 11.     $N$ = the leaf node to the right of $N$ |
| 12. until $N = \emptyset$ |
| 13. return the $k$ closest pairs found so far |

Fig. 18.   A CP algorithm using only a single LSB-tree.

on low-dimensional data, deteriorate rapidly as the dimensionality increases [Bohm 2000; Weber et al. 1998], 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. [1994] propose the *TV-tree* which improves conventional R-trees [Beckmann et al. 1990; Guttman 1984] by creating MBRs only in selected subspaces. Weber et al. [1998] 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. [2000] develop the *IQ-tree*, combining features of the R-tree and VA-file. Chaudhuri and Gravano [1999] perform NN search by converting it to range queries. In Berchtold et al. [2000] Berchtold et al. provide a solution leveraging high-dimensional Voronoi diagrams, whereas Korn et al. [2001] tackle the problem by utilizing the fractal dimensionality of the dataset. Koudas et al. [2004] give a bitmap-based approach. The state of the art is due to Jagadish et al. [2005]. 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 [Bennett et al. 1999; Ciaccia and Patella 2000]. Approximate retrieval improves efficiency by relaxing the precision of verification. Goldstein and Ramakrishnan [2000] assume that the query distribution is known, and leverage the knowledge to balance the efficiency and result quality. Ferhatosmanoglu et al. [2001] find NNs by examining only the interesting subspaces. Chen and Ling [2002] combine sampling with a reduction [Chaudhuri and Gravano 1999] to range search. Li et al. [2002] first partition the dataset into clusters, and then prune the irrelevant clusters according to their radii. Houle and Sakuma [2005] 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. [2003] 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 [Fagin et al. 2001] for top-$k$ search. We will also evaluate MedRank in the experiments.

None of the aforementioned solutions ensures sublinear growth of query cost in the worst case. How to achieve this has been carefully studied in the theory community (see, for example, Har-Peled [2001] and Krauthgamer and Lee [2004] 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 [1998] for in-memory data. Gionis et al. [1999] 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. [2004]. Bawa et al. [2005] 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" [Bawa et al. 2005]. Unfortunately, no existing hash function for $\ell_p$ norms is known to possess this property. Charikar [2002] investigates LSH for several distance metrics different from $\ell_p$ norms. LSH has also received other theoretical improvements [Andoni and Indyk 2006; Panigrahy 2006] which cannot be implemented in relational databases. Furthermore, several heuristic variations of LSH have also been suggested. For example, Lv et al. [2007] reduce space consumption by probing more data in answering a query, while recently Athitsos et al. [2008] introduce the notion of *distance-based hashing*. The solutions of Athitsos et al. [2008], and Lv et al. [2007] guarantee neither sublinear cost nor good approximation ratios.

CP search is also one of the oldest problems studied in computational geometry. In two-dimensional space, Shamos and Hoey [1975] give an optimal algorithm that runs in $O(n \log n)$ time. Interestingly, for any fixed dimensionality $d$, the problem can also be settled optimally in $O(n \log n)$ time, as discovered by Lenhof and Smid [1992]. The optimality of the aforesaid algorithms, however, assumes that $d$ is a constant; when it is not, their running time increases exponentially with $d$. Avoiding such exponential growth turns out to be a hard problem. Recently, by resorting to matrix multiplication, Indyk et al. [2004] give several algorithms with nontrivial bounds that are applicable to $\ell_1$ and $\ell_\infty$ norms, but not to the other $\ell_p$ norms. The methods mentioned earlier are rather theoretical. On the practical side, Hjaltason and Samet [1998] give a solution, called *distance browsing*, that utilizes R-trees to report point pairs in ascending order of their distances. Following the same idea, Corral et al. [2000] propose an enhanced algorithm with smaller cost which will be evaluated in the experiments.

The preceding solutions aim at solving the CP problem exactly. There have been attempts to address the approximate version, but most of those algorithms require running time that is quadratic to the cardinality $n$ (albeit faster than $dn^2$); see for example Kleinberg [1997]. Based on the LSH technique, Datar et al. [2004] propose an algorithm with subquadratic time, but their analysis targets internal memory only. Our discussion in Section 7.1 can in fact also be applied to LSH, and shows that the case of external memory requires a

more elaborate reasoning approach. Furthermore, our result in Section 7.2 is actually better (by a logarithmic factor) than the obvious bound adapted from Datar et al. [2004] (which corresponds to the performance of algorithm CP1 in Figure 15). Another algorithm worth mentioning is due to Lopez and Liao [2000]. When $d$ is regarded as a constant, their algorithm, which we call D-shift, guarantees an answer with constant approximation ratio. Their algorithm can be incorporated in relational databases, and will be compared to our solutions in the experiments.

Finally, it is worth pointing out that this article substantially extends its preliminary version [Tao et al. 2009] in the following ways. First, in Section 6, we have shown how to modify our NN techniques to achieve approximation $c+\epsilon$ for any integer $c \geq 3$ (only $c = 2$ is discussed in Tao et al. [2009]), thus giving a stronger trade-off between the result quality and the query/space efficiency. Second, while the preliminary work concentrates on NN search, the current version contains a full set of results on the CP problem (Section 7), together with the corresponding experiment in the next section.

## 9. EXPERIMENTS

Next we experimentally evaluate the performance of LSB-trees, using the existing methods as benchmarks. Section 9.1 describes the datasets and queries. Sections 9.2 and 9.3 list the techniques to be examined for NN and CP search, respectively. Section 9.4 explains the computing environments as well as the assessment metrics. Section 9.5 demonstrates the superiority of the LSB-forest over alternative LSH implementations. Then, Section 9.6 (9.7) shows that our techniques significantly outperform the previous methods, in both exact and approximate NN (CP) search.

### 9.1 Data and Queries

We experimented with both synthetic and real datasets. Synthetic data were generated according to a *varden* distribution to be clarified shortly. As for real data, we deployed datasets *color* and *mnist*, which were also used in the papers Fagin et al. [2003] and Jagadish et al. [2005] where MedRank and iDistance are invented, respectively (both methods were included in our experiments). For all datasets, the universe was normalized to have domain $[0, 10000]$ on each dimension.

The distance metric employed was Euclidean distance. Each NN *workload* contained 50 query points that followed the same distribution as the underlying dataset. CP search takes no query point; it simply finds the $k$ closest pairs in a dataset.

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 19 illustrates
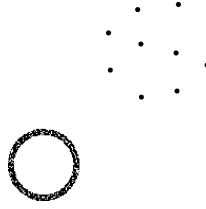
Fig. 19.　The *varden* distribution.

the idea with a 2D example. We varied 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 had 10 and 40 query points that fell in the areas of the sparse and dense clusters, respectively. No query point coincided with any data point.

*Color.* This is a 32-dimensional dataset[3] with 67,967 points, where each point describes the color histogram of an image in the Corel collection [Jagadish et al. 2005]. We randomly removed 50 points to form a query set. As a result, our *color* dataset has cardinality 67,917.

*Mnist.* The original *mnist* dataset[4] is a set of 60,000 points. Each point is 784-dimensional, capturing the pixel values of a $28 \times 28$ image. Since, however, most pixels are insignificant, we reduced dimensionality by taking the 50 dimensions with the largest variances. After this, we got two identical points so one of them was removed, rendering the final cardinality to be 59,999. The *mnist* collection also contains a test set of 10,000 points [Fagin et al. 2003], among which we randomly picked 50 to form our workload. Obviously, each query point was also projected onto the same 50 dimensions output by the dimensionality reduction.

## 9.2 Competitors for Nearest Neighbor Search

*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 takes $l$ LSB-trees ($l$ given by Eq. (7)), and applies algorithm NN1 (Figure 7) for query processing. For $k$NN queries with $k > 1$, LSB-forest still uses the same $l$ (i.e., no increase in the number of trees) and query algorithm, except that the terminating condition $E_2$ is modified in the way explained in Section 4.2.

---

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

*LSB-no$E_2$*. Same as LSB-forest except that it disables condition $E_2$ in algorithm NN1. In other words, LSB-no$E_2$ terminates on condition $E_1$ only. LSB-no$E_2$ is applied only for $k = 1$.

*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 4.2, it disables condition $E_1$, and terminates on $E_2$ only (again, $E_2$ needs to be modified for $k > 1$).

*Rigorous- [Indyk and Motwani 1998] and* ad hoc-LSH *[Gionis et al. 1999]*. They 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 to match the guarantee of the LSB-forest (see Section 6). Ad hoc-LSH requires a set of $l$ hash tables, which is used to perform BC queries at a magic radius (to be tuned experimentally later), where $l$ is the same as in Eq. (7). Rigorous-LSH can be regarded as combining multiple versions of ad hoc-LSH, one for every radius supported.

*iDistance [Jagadish et al. 2005]*. This is a famous approach for exact NN search. As mentioned in Section 8, 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 [Fagin et al. 2003]*. This is a recently proposed method for approximate NN search with a nontrivial quality guarantee. Given a dataset, MedRank creates several sorted lists, such that every data point has an entry in each list. More specifically, an 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 of lists equals $\log n$ (following Theorem 4 in Fagin et al. [2003]), 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.

## 9.3 Competitors for Closest Pair Search

*Quadratic*. This is the naive approach that examines all pairs of points. It serves as a benchmark for comparison with other solutions to the CP problem not included in our experiments.

*LSB-forest*. This method uses $l$ LSB-trees, where $l$ is given by Eq. (7), and applies algorithm CP2 (Figure 16). The same $l$ and algorithm are also used for $k$CP search with $k > 1$, except that CP2 needs to report the $k$ best pairs (instead of 1).

*2LSB-tree*. The method uses two LSB-trees; it applies algorithm CP3 (Figure 18) on each tree separately, and returns the $k$ best pairs after combining the outputs from both trees. Here we are using one more tree than the LSB-tree method in the previous subsection, in order to outperform the

competitor D-shift (to be introduced later) in result quality. Note that using 2 trees does not increase the space and update-time complexities. Namely, 2LSB-tree still occupies linear space and can be updated in logarithmic time.

*Distance browsing (DistBrowsing) [Corral et al. 2000].* This is an extensively-cited solution to exact $k$CP search. Similar to Hjaltason and Samet [1998], it leverages an R-tree on the underlying dataset to enumerate point pairs in ascending order of distances.

*Diagonal shift (D-shift) [Lopez and Liao 2000].* This is an approximate algorithm with a nontrivial quality guarantee. Given a $d$-dimensional dataset, it creates $d$ copies of the dataset, where each copy is obtained by shifting the original dataset along the direction of the main diagonal by a different offset (hence the name D-shift). Then, the closest pairs are found by sorting and scanning each list once.

## 9.4 Computing Environments and Assessment Metrics

The page size was fixed to 4,096 bytes. All the experiments were run on a computer equipped with a 3 GHz CPU. A memory buffer of 50 pages was adopted in all cases. Under such settings, the running time of all (NN and CP) algorithms was dominated by their I/O overhead. Therefore, we will report the number of I/Os as the computation cost.

*Quality of NN search.* We evaluate the quality of a $k$NN 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

$$R_i(q) = \|o_i, q\| / \|o_i^*, q\|. \tag{16}$$

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 was used in most experiments. Sometimes we needed to scrutinize the quality of neighbors at individual ranks. In that case, we measured the *average rank-i ratio* $(1 \leq i \leq k)$, which is the mean of the rank-$i$ ratios of all queries in $W$, namely, $(\sum_{\forall q \in W} R_i(q))/|W|$.

*Quality of CP search.* This is also assessed based on rank-$i$ ratio and overall ratio, both of which are extended from the earlier definitions in a straightforward manner.

## 9.5 Behavior of LSH Implementations

This section explores the characteristic behavior of LSB-forest, LSB-no$E_2$, rigorous-LSH, and ad hoc-LSH. For this purpose, we focused on $k = 1$
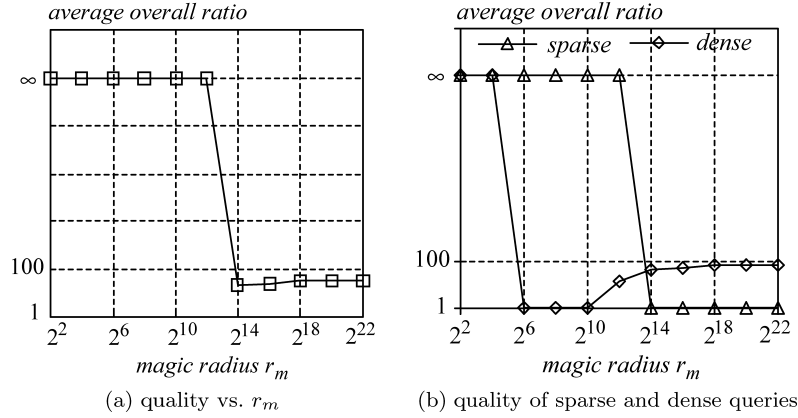
Fig. 20.   Magic radius tuning for *ad hoc-LSH* (*varden-10k50d*).

because the LSH methodology was originally designed in the context of single NN retrieval. Note that when $k = 1$, the overall ratio of a query is identical to its rank-1 ratio. The data distribution examined was *varden*, as it allowed us to adjust the dimensionality and cardinality. Unless otherwise stated, a *varden* dataset had a default cardinality $n$ = 10k and dimensionality $d$ = 50.

Recall that ad hoc-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 $r_m$ good for all NN queries. To demonstrate this, Figure 20(a) shows the average overall ratio of ad hoc-LSH as $r_m$ varied from $2^2$ to $2^{22}$. For small $r_m$, the ratio is $\infty$, implying that ad hoc-LSH *missed* at least one query in the workload, namely, returning nothing at all. The ratio improved suddenly to 66 when $r_m$ reached $2^{14}$, and stabilized as $r_m$ grew further. It is thus clear that, given any $r_m$, the result of ad hoc-LSH was 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, ad hoc-LSH may return an empty result. Conversely, if $r_m$ is considerably larger, ad hoc-LSH may output a point much worse than the real NN. We performed an experiment to verify this. Recall that a workload for *varden* has queries in both the sparse and dense clusters. Let us call the former the *sparse* queries, and the latter the *dense* queries. We observed that the average NN distance of a sparse (dense) query was around 12,000 (15). The phenomenon in Figure 20(a) occurred because *the values of $r_m$ good for sparse queries were bad for dense queries, and vice versa*. To support the claim, Figure 20(b) plots the average overall ratios of sparse and dense queries separately. When $r_m$ was smaller than or equal to $2^{13}$ = 8,192, it was much lower than the NN-distances of sparse queries; hence, ad hoc-LSH returned nothing for them, as is why the *sparse* curve in Figure 20 (b) stays at $\infty$ for all $r_m \leq 2^{13}$. As $r_m$ climbed to $2^{12}$, ad hoc-LSH started to return bad results for many dense queries. The situation was worse for larger $r_m$, so the *dense* curve Figure 20(b) increases continuously from $2^{12}$. In all the following experiments, we fixed $r_m$ to the optimal value $2^{14}$.

Table I.  Result Quality on *varden* Data

| $d$ | 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-no$E_2$ | 1 | | | |

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

| $n$ | 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-no$E_2$ | 1 | | | | |

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



(a) cost vs. $d$ $(n = 50\text{k})$          (b) cost vs. $n$ $(d = 50)$
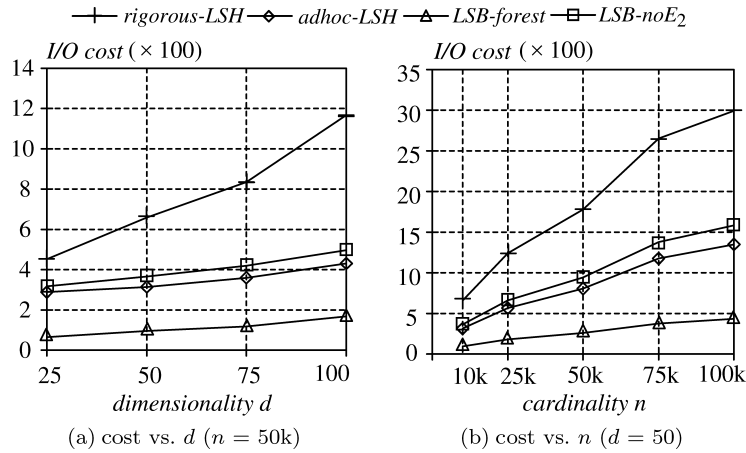
Fig. 21.   Query efficiency on *varden* data.

The next experiment compares the result quality of rigorous-LSH, ad hoc-LSH, LSB-forest, and LSB-no$E_2$. Table I(a) and (I(b)) shows their average overall ratios under different dimensionalities (cardinalities). Both rigorous-LSH and LSB-no$E_2$ achieved perfect quality, namely, they successfully returned the *exact* NN for all queries. LSB-forest incurred slightly higher error because in general it accesses fewer points than LSB-no$E_2$, and thus, has a lower chance of encountering the real NN. Ad hoc-LSH was by far the worst method, and its effectiveness deteriorated rapidly as the dimensionality or cardinality increased.

To evaluate the query efficiency of the four methods, Figure 21(a) (21(b)) plots their I/O cost as a function of dimensionality $d$ (cardinality $n$). LSB-forest considerably outperformed its competitors in all cases. Notice that while LSB-no$E_2$ was slightly more costly than ad hoc-LSH, LSB-forest entailed only a fraction of the overhead of ad hoc-LSH. This phenomenon reveals the importance of having terminating condition $E_2$ in the NN1 algorithm. Rigorous-LSH was much more expensive than the other approaches, which is consistent with its vast asymptotical complexity.

Table II. Space Consumption on *varden* Data in Megabytes

| $d$ | 25 | 50 | 75 | 100 |
|---|---|---|---|---|
| rigorous-LSH | 382 | 896 | 1,563 | 2,436 |
| adhoc-LSH | 24 | 57 | 101 | 159 |
| LSB-forest | 24 | 57 | 101 | 159 |

(a) space vs. dimensionality $d$ ($n = 50k$)

| $n$ | 10k | 25k | 50k | 75k | 100k |
|---|---|---|---|---|---|
| rigorous-LSH | 895 | 3,624 | 10,323 | 18,892 | 29,016 |
| adhoc-LSH | 57 | 231 | 660 | 1,208 | 1,855 |
| LSB-forest | 57 | 231 | 660 | 1,208 | 1,855 |

(b) space vs. cardinality $n$ ($d = 50$)

Tables II(a) and II(b) show the space consumption (in megabytes) 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 had the same space cost as LSB-forest. Furthermore, ad hoc-LSH also occupied 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 required more space than LSB-forest by a factor of $\log d + \log t$, where $t$ (the largest coordinate on each dimension) was 10,000 in our experiments.

It is evident that LSB-forest is overall the best technique in the preceding experiments. It retains the query accuracy of rigorous-LSH, consumes the same space as ad hoc-LSH, and incurs significantly smaller query cost than both.

## 9.6 Comparison of NN Solutions

Having verified the correctness of our theoretical analysis, in the sequel we assess the practical performance of SeqScan, LSB-tree, LSB-forest, ad hoc-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 datasets *color* and *mnist* were adopted in the subsequent evaluation. The workload on *color* (*mnist*) had an average NN distance of 833 (11,422). We set the magic radius of ad hoc-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 varied from 1 to 100.

Let us start with query efficiency. Figure 22(a) (22(b)) illustrates the average cost of a $k$NN query on dataset *color* (*mnist*) as a function of $k$. LSB-tree was by far the fastest method, and outperformed all the other approaches by a factor of at least an order of magnitude. In particular, on *mnist*, LSB-tree even achieved a speedup of two orders of magnitude over iDistance, justifying the advantages of approximate retrieval. LSB-forest was also much faster than iDistance, MedRank, and ad hoc-LSH, especially in returning a large number of neighbors.
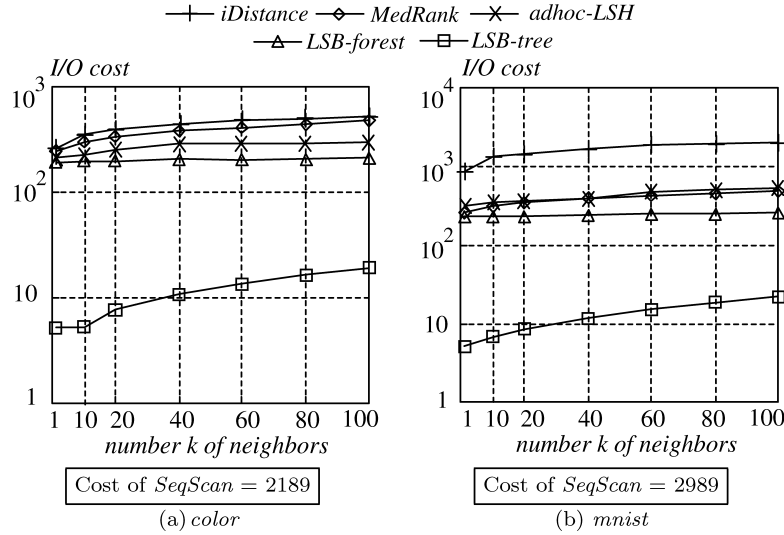
Fig. 22.   Efficiency of $k$NN search.

The next experiment inspects the result quality of the approximate techniques. Focusing on *color* (*mnist*), Figure 23(a) (23(b)) plots the average overall ratios of MedRank, LSB-forest, and LSB-tree as a function of $k$. Since ad hoc-LSH may *miss* a query (i.e., unable to return $k$ neighbors), we present its results as a table in Figure 23(c), where each cell contains two numbers. The number in the bracket indicates how many queries were missed (out of 50), and the number outside is the average overall ratio of the queries that were answered properly. No ratio is reported if ad hoc-LSH missed more than 30 queries.

LSB-forest incurred low error in all cases (maximum ratio below 1.5), owing to its nice theoretical properties. LSB-tree also had good precision (maximum ratio 2), indicating that the proposed conversion (from a $d$-dimensional point to a Z-order value) adequately preserved the spatial proximity of data points. MedRank, in contrast, exhibited much worse precision than the proposed solutions. In particular, observe that MedRank was not effective in the important case of single NN search ($k = 1$), for which its average overall ratio was over 4. Finally, ad hoc-LSH was clearly unreliable due to the large number of queries it missed.

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 measured the average rank-$i$ ratios at each $i \in [1, 10]$. Figures 24(a) and 24(b) demonstrate the results on *color* and *mnist*, respectively (ad hoc-LSH is not included because it missed many queries). Apparently, both LSB-forest and LSB-tree provided results significantly better than MedRank at all ranks. Observe that the quality of MedRank deteriorated considerably at high ranks, whereas our solutions returned fairly good neighbors even at the highest rank. Note that the results in Figure 24

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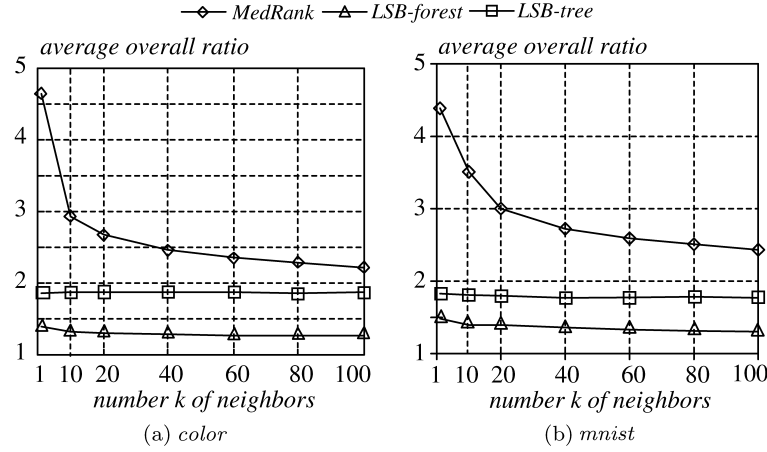
| $k$ | 1 | 10 | 20 | 40 | 60 | 80 | 100 |
|---|---|---|---|---|---|---|---|
| *color* | 1.2 (0) | 1.3 (30) | - (42) | - (46) | - (46) | - (47) | - (48) |
| *mnist* | 1.2 (0) | 1.3 (13) | 1.3 (19) | 1.4 (28) | - (37) | - (39) | - (41) |

(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)

Fig. 23.   Average overall ratio vs. $k$.

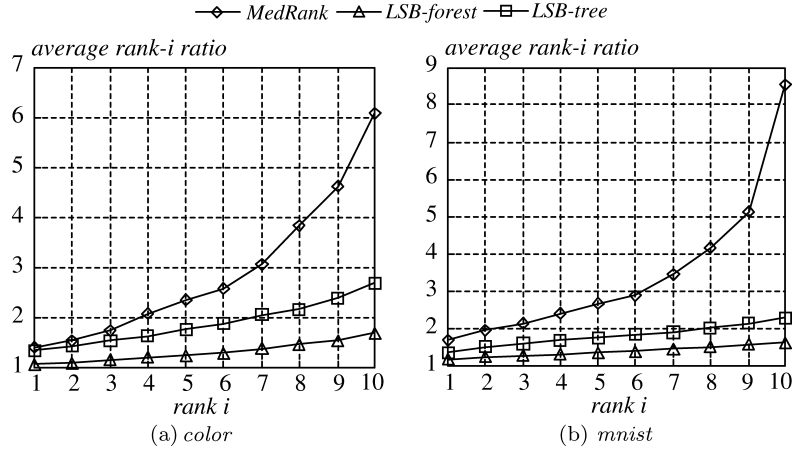

(a) *color*    (b) *mnist*

Fig. 24.   Average ratios at individual ranks for 10NN queries.

should not be confused with those of Figure 23. For example, the average rank-1 ratio (of $k = 10$) is different from the overall average ratio of $k = 1$.[5]

Table III compares the space consumption of different methods. LSB-tree required slightly less space than iDistance and MedRank. We, however, ought to point out that, at least in theory, LSB-tree needs to store more information than iDistance, so the latter should be more space economical.

_____

[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.

Table III.  Space Consumption on Real Data in Megabytes

|  | iDistance | MedRank | adhoc-LSH | LSB-forest | LSB-tree |
|---|---|---|---|---|---|
| color | 14 | 17 | 573 | 573 | 13 |
| mnist | 18 | 19 | 874 | 874 | 16 |
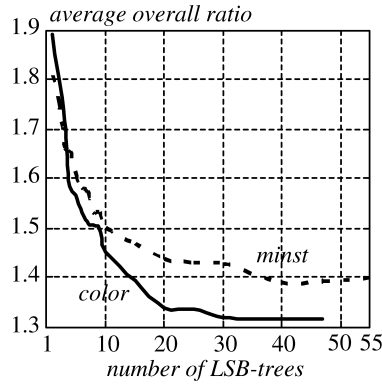


Fig. 25.    Benefits of using multiple LSB-trees ($k = 10$).

However, the actual space consumption may contain some extra overhead depending on the concrete implementation. The implementation of iDistance we deployed was exactly the one written by the authors of Jagadish et al. [2005]. Also note that our implementations of LSB-tree, LSB-forest, and ad hoc-LSH have been improved compared to those in the preliminary version [Tao et al. 2009].

Recall that LSB-forest utilizes a large number $l$ of LSB-trees, where the number $l$ was 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 trade-off, Figure 25 shows the average overall ratio of 10NN queries on the two real datasets, when $x$ grew from 1 to the corresponding $l$ of LSB-forest. Interestingly, the precision improved 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.

In summary, our experiment results suggest that an exact solution such as iDistance often requires excessively long query response time in practice, confirming the motivation to study approximate solutions. The most serious drawback of ad hoc-LSH is that it may fail to report enough neighbors for many queries. In any case, its query overhead is still too high to provide fast response time. MedRank is even more expensive than ad hoc-LSH; furthermore, its result quality is relatively low (i.e., the answers it returns are quite far from the real $k$ NNs). LSB-forest is the only (approximate) method that guarantees
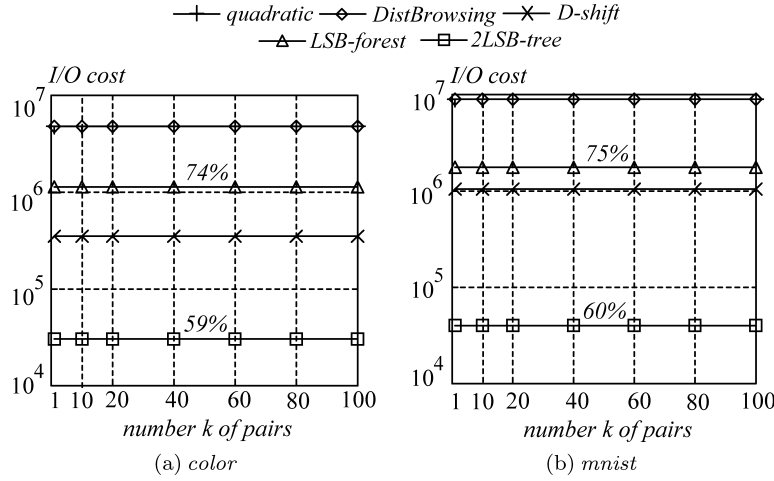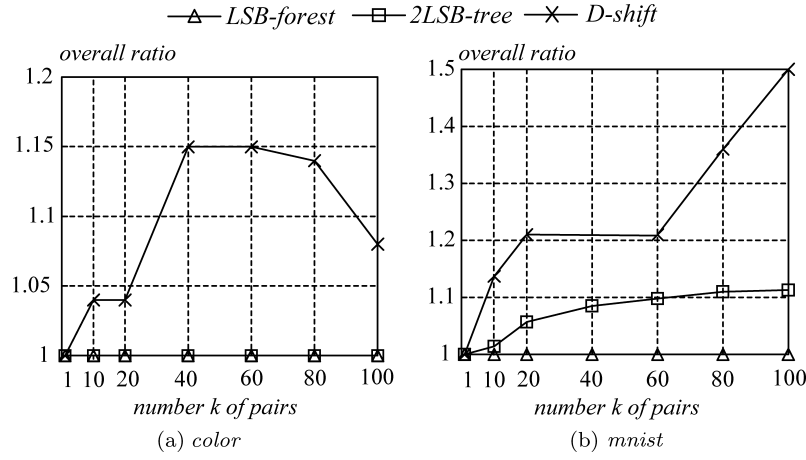
Fig. 26.   Cost of $k$CP search.

high result quality and sublinear query cost in all cases. However, as with ad hoc-LSH and MedRank, it is not friendly to updates. Overall the best solution is LSB-tree: it demands only linear space, supports fast updates, returns very accurate results, and is extremely efficient in query processing.

## 9.7 Comparison of CP Solutions

We now proceed to study the methods for closest pair search, using again the real datasets *color* and *mnist*. The first experiment compares their efficiency of finding $k$ closest pair *from scratch*. Namely, we do not assume any precomputation; if a method requires an index (e.g., an R-tree or LSB-trees), it must construct it on-the-fly, with the construction time added into its total cost.

Figure 26(a) (26(b)) shows the I/O cost of each method on dataset *color* (*mnist*), as the number $k$ of pairs retrieved changed from 1 to 100. 2LSB-tree was by far the most efficient, and outperformed D-shift and LSB-forest by more than an order of magnitude. As expected, LSB-forest was slower than D-shift because the former's time complexity is superlinear with respect to the dataset cardinality $n$, whereas the latter's is linear [Lopez and Liao 2000]. Nevertheless, as shown shortly, the advantage of LSB-forest is that it returned much more accurate answers than D-shift. DistBrowsing was as expensive as the naive algorithm quadratic. This is not surprising because the effectiveness of spatial access methods (particular, R-trees) deteriorates seriously in high-dimensional space such that they hardly offer any pruning in distance browsing. The deficiency of exact solutions, once again, confirms the importance of approximate methods. The parameter $k$, in the tested range of values, did not affect the efficiency of any method. The percentages on the curves of 2LSB-tree and LSB-forest indicate how much of the overall cost was spent on on-the-fly index construction. In other words, if the LSB-trees already existed before the CP search, the costs of 2LSB-tree and LSB-forest would be 60% and

Fig. 27.    Quality of *k*CP search.
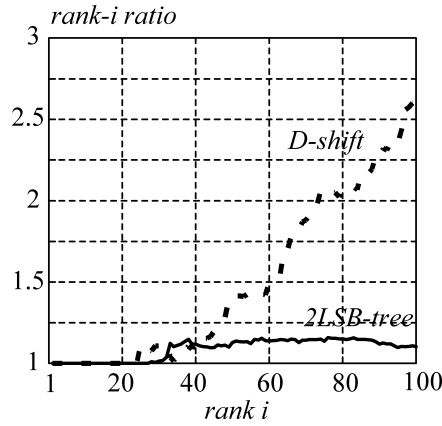


Fig. 28.    Quality of individual pairs returned (*minst*, $k = 100$).

75% less expensive. Note that stripping 75% from the cost of LSB-forest would make it even faster than D-shift.

To assess the quality of results, Figure 27(a) (27(b)) compares the overall ratios of the three approximate methods in the experiment of Figure 26(a) (26(b)). Recall that the overall ratio indicates the average quality of all *k* pairs returned by a method. LSB-forest achieved an overall ratio of 1 on both datasets, meaning that it found the *exact* 100 *k* closest pairs in both cases. 2LSB-tree also produced perfectly accurate answers on *color*; on *mnist*, on average the answers it found were worse than the exact ones by merely 10%. The quality of D-shift was clearly much worse, thus leaving its expensive computation cost unjustified (see Figure 26).

To zoom into the quality of individual closest pairs, Figure 28 plots the rank-*i* ratios for all $i \in [1, 100]$, of the results returned by 2LSB-tree and D-shift in performing 100CP on *mnist*. Recall that the rank-*i* ratio measures how much times worse the *i*th pair found by a method is, compared to the
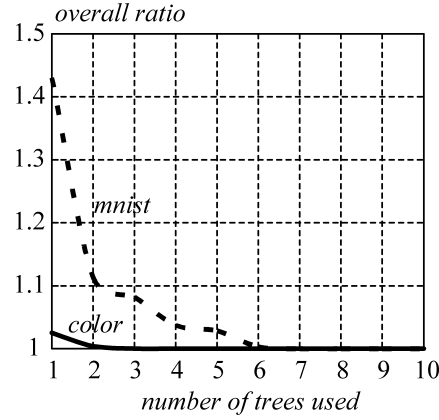
Fig. 29.   Benefits of using multiple trees ($k = 100$).

exact $i$th closest pair. LSB-forest is omitted because, as mentioned earlier, it produced perfect answers. Similarly, we also omit the results on *color* because even 2LSB-tree was able to achieve perfect accuracy on that dataset. In Figure 28, we can see that all the pairs found by 2LSB-tree were fairly accurate, whereas the quality of the "high-rank" pairs returned by D-shift was rather poor.

We have seen in Section 9.2 that, for NN retrieval, there exists a graceful trade-off between the result quality and the number of LSB-trees used for query processing. The last experiment aims at identifying a similar phenomenon for CP search. For this purpose, we generalized the strategy of 2LSB-tree; namely, given $x$ LSB-trees, we ran algorithm CP3 on each of them, and then reported the $k$ best pairs among the outputs from all trees. Apparently, the result quality should improve as $x$ increases, but it is most interesting to identify when the quality would improve to perfection, that is, an overall ratio of 1! The results on *color* and *mnist* are plotted in Figure 29 for $k = 100$. Clearly, the growth of $x$ brought dramatic improvements on the result quality. On *mnist*, only 7 trees were necessary to attain perfect precision (the number was 2 for *color*, as we already knew from Figure 27(a)).

In summary, our experiments show that an exact solution to CP search such as DistBrowsing is not suitable in high-dimensional space due to their prohibitive running time. D-shift has the drawbacks that it: (i) still entails expensive overhead, and (ii) cannot guarantee accurate answers, especially at high ranks. Same as in NN search, LSB-forest is the only (approximate) method able to guarantee excellent result quality in any case. Nevertheless, it still cannot escape the trap of costly execution time. The best approach is to perform CP search using a small (e.g., 2) number of LSB-trees, which is significantly faster than all the other methods (over an order of magnitude), and returns close-to-exact answers in most cases.

## 10. CONCLUSIONS

Nearest neighbor search in high-dimensional space finds numerous applications in a large number of disciplines. This article develops an access method called the LSB-tree to enable fast NN search with excellent result quality. Our discovery carries both theoretical and practical significance. In theory, by combining several LSB-trees, we dramatically improve the (asymptotical and actual) space and query efficiency of the previous LSH implementations, without compromising the result quality. In practice, by using a single LSB-tree, we give an effective indexing scheme that can be easily incorporated in a relational database, consumes linear space, supports logarithmic-time updates, and can be used to answer NN queries accurately and efficiently.

As a second step, we have extended our LSB-technique to attack the closest pair problem in high-dimensional space, which is another classic problem with many applications. Our contributions on this topic also have important values in both theory and reality. In particular, we have shown that, in external memory, the closest pair problem can be solved in time strictly lower than the quadratic complexity, regardless of the dimensionality. In practice, our purely-relational solutions can be immediately applied in a commercial system. Furthermore, these solutions can directly leverage the indexing scheme mentioned earlier for NN search. This is a fairly attractive feature because, with only a single structure, one is able to adequately tackle two difficult problems at the same time.

## REFERENCES

ANDONI, A. AND INDYK, P. 2006. Near-Optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*. 459–468.

ARYA, S., MOUNT, D. M., NETANYAHU, N. S., SILVERMAN, R., AND WU, A. Y. 1998. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM 45*, 6, 891–923.

ATHITSOS, V., POTAMIAS, M., PAPAPETROU, P., AND KOLLIOS, G. 2008. Nearest neighbor retrieval using distance-based hashing. In *Proceedings of the International Conference on Data Engineering (ICDE'08)*. 327–336.

BAWA, M., CONDIE, T., AND GANESAN, P. 2005. Lsh forest: Self-Tuning indexes for similarity search. In *Proceedings of the International World Wide Web Conference (WWW'05)*. 651–660.

BECKMANN, N., KRIEGEL, H., SCHNEIDER, R., AND SEEGER, B. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 322–331.

BENNETT, K. P., FAYYAD, U., AND GEIGER, D. 1999. Density-Based indexing for approximate nearest-neighbor queries. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 233–243.

BERCHTOLD, S., BOHM, C., JAGADISH, H. V., KRIEGEL, H.-P., AND SANDER, J. 2000. Independent quantization: An index compression technique for high-dimensional data spaces. In *Proceedings of the International Conference on Data Engineering (ICDE'00)*. 577–588.

BERCHTOLD, S., KEIM, D. A., KRIEGEL, H.-P., AND SEIDL, T. 2000. Indexing the solution space: A new technique for nearest neighbor search in high-dimensional space. *IEEE Trans. Knowl. Data Engin. 12*, 1, 45–57.

BEYER, K. S., GOLDSTEIN, J., RAMAKRISHNAN, R., AND SHAFT, U. 1999. When is "nearest neighbor" meaningful? In *Proceedings of the International Conference on Database Theory (ICDT'99)*. 217–235.

BOHM, C.  2000.  A cost model for query processing in high dimensional data spaces. *ACM Trans. Datab. Syst. 25*, 2, 129–178.

BREUNIG, M. M., KRIEGEL, H.-P., NG, R. T., AND SANDER, J.  2000.  Lof: Identifying density-based local outliers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 93–104.

CHARIKAR, M.  2002.  Similarity estimation techniques from rounding algorithms. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC'02)*.

CHAUDHURI, S. AND GRAVANO, L.  1999.  Evaluating top-k selection queries. In *Proceedings of the International Conference on Very Large Databases (VLDB'99)*. 397–410.

CHEN, C.-M. AND LING, Y.  2002.  A sampling-based estimator for top-k query. In *Proceedings of the Interational Conference on Data Engineering (ICDE'02)*. 617–627.

CIACCIA, P. AND PATELLA, M.  2000.  Pac nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In *Proceedings of the International Conference on Data Engineering (ICDE'00)*. 244–255.

CORRAL, A., MANOLOPOULOS, Y., THEODORIDIS, Y., AND VASSILAKOPOULOS, M.  2000.  Closest pair queries in spatial databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 189–200.

DATAR, M., IMMORLICA, N., INDYK, P., AND MIRROKNI, V. S.  2004.  Locality-Sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Annual ACM Symposium on Computational Geometry (SoCG'04)*. 253–262.

FAGIN, R., KUMAR, R., AND SIVAKUMAR, D.  2003.  Efficient similarity search and classification via rank aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 301–312.

FAGIN, R., LOTEM, A., AND NAOR, M.  2001.  Optimal aggregation algorithms for middleware. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'01)*. 102–113.

FERHATOSMANOGLU, H., TUNCEL, E., AGRAWAL, D., AND ABBADI, A. E.  2001.  Approximate nearest neighbor searching in multimedia databases. In *Proceedings of the International Conference on Data Engineering (ICDE'01)*. 503–511.

FERRAGINA, P. AND GROSSI, R.  1999.  The String B-tree: A new data structure for string search in external memory and its applications. *J. ACM 46*, 2, 236–280.

GAEDE, V. AND GUNTHER, O.  1998.  Multidimensional access methods. *ACM Comput. Surv. 30*, 2, 170–231.

GIONIS, A., INDYK, P., AND MOTWANI, R.  1999.  Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Databases (VLDB'99)*. 518–529.

GOLDSTEIN, J. AND RAMAKRISHNAN, R.  2000.  Contrast plots and p-sphere trees: Space vs. time in nearest neighbour searches. In *Proceedings of the International Conference on Very Large Databases (VLDB'00)*. 429–440.

GUTTMAN, A.  1984.  R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 47–57.

HAR-PELED, S.  2001.  A replacement for voronoi diagrams of near linear size. In *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science (FOCS'01)*. 94–103.

HJALTASON, G. R. AND SAMET, H.  1998.  Incremental distance join algorithms for spatial databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 237–248.

HJALTASON, G. R. AND SAMET, H.  1999.  Distance browsing in spatial databases. *ACM Trans. Datab. Syst. 24*, 2, 265–318.

HOULE, M. E. AND SAKUMA, J.  2005.  Fast approximate similarity search in extremely high- dimensional data sets. In *Proceedings of the International Conference on Data Engineering (ICDE'05)*. 619–630.

INDYK, P., LEWENSTEIN, M., LIPSKY, O., AND PORAT, E.  2004.  Closest pair problems in very high dimensions. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP'04)*. 782–792.

INDYK, P. AND MOTWANI, R.  1998.  Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC'98)*. 604–613.

JAGADISH, H. V., OOI, B. C., TAN, K.-L., YU, C., AND ZHANG, R. 2005. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Trans. Datab. Syst. 30*, 2, 364–397.

KLEINBERG, J. M. 1997. Two algorithms for nearest-neighbor search in high dimensions. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC'97)*. 599–608.

KORN, F., PAGEL, B.-U., AND FALOUTSOS, C. 2001. On the 'dimensionality curse' and the 'self-similarity blessing'. *IEEE Trans. Knowl. Data Engin. 13*, 1, 96–111.

KOUDAS, N., OOI, B. C., SHEN, H. T., AND TUNG, A. K. H. 2004. Ldc: Enabling search by partial distance in a hyper-dimensional space. In *Proceedings of the International Conference on Data Engineering (ICDE'04)*. 6–17.

KRAUTHGAMER, R. AND LEE, J. R. 2004. Navigating nets: Simple algorithms for proximity search. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*. 798–807.

LENHOF, H.-P. AND SMID, M. 1992. Enumerating the k closest pairs optimally. In *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science (FOCS'92)*. 380–386.

LI, C., CHANG, E. Y., GARCIA-MOLINA, H., AND WIEDERHOLD, G. 2002. Clustering for approximate similarity search in high-dimensional spaces. *IEEE Trans. Knowl. Data Engin. 14*, 4, 792–808.

LIN, K.-I., JAGADISH, H. V., AND FALOUTSOS, C. 1994. The tv-tree: An index structure for high-dimensional data. *VLDB J. 3*, 4, 517–542.

LOPEZ, M. A. AND LIAO, S. 2000. Finding k-closest-pairs efficiently for high dimensional data. In *Proceedings of the Canadian Conference on Computational Geometry (CCCG'00)*. 197–204.

LV, Q., JOSEPHSON, W., WANG, Z., CHARIKAR, M., AND LI, K. 2007. Multi-Probe lsh: Efficient indexing for high-dimensional similarity search. In *Proceedings of the International Conference on Very Large Databases (VLDB'07)*. 950–961.

PANIGRAHY, R. 2006. Entropy based nearest neighbor search in high dimensions. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'06)*. 1186–1195.

ROUSSOPOULOS, N., KELLEY, S., AND VINCENT, F. 1995. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 71–79.

SHAMOS, M. I. AND HOEY, D. 1975. Closest-point problems. In *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science (FOCS'75)*. 151–162.

TAO, Y., YI, K., SHENG, C., AND KALNIS, P. 2009. Quality and efficiency in high dimensional nearest neighbor search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 563–576.

WEBER, R., SCHEK, H.-J., AND BLOTT, S. 1998. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the International Conference on Very Large Databases (VLDB'98)*. 194–205.

WONG, R. C.-W., TAO, Y., FU, A. W.-C., AND XIAO, X. 2007. On efficient spatial matching. In *Proceedings of the International Conference on Very Large Databases (VLDB'07)*. 579–590.