Redundant Bit Vectors for Quickly Searching High-Dimensional Regions

Jonathan Goldstein, John C. Platt, Christopher J.C. Burges

Microsoft Research 1 Microsoft Way Redmond, WA 98052 USA

Abstract. Applications such as audio fingerprinting require search in high dimensions: find an item in a database that is similar to a query. An important property of this search task is that negative answers are very frequent: much of the time, a query does not correspond to any database item.
We propose *Redundant Bit Vectors* (RBVs): a novel method for quickly solving this search problem. RBVs rely on three key ideas: 1) approximate the high-dimensional regions/distributions as tightened hyperrectangles, 2) partition the query space to store each item redundantly in an index and 3) use bit vectors to store and search the index efficiently. We show that our method is the preferred method for very large databases or when the queries are often not in the database. Our method is 109 times faster than linear scan, and 48 times faster than locality-sensitive hashing on a data set of 239369 audio fingerprints.

1 Introduction

Consider the abstract search problem: given a database of items (described in some way), and a stream of queries, how can we quickly find an item in the database that is similar to a query?

This search problem can be solved by mapping the items into geometric regions (e.g., hyperspheres) or probability distributions in a high dimensional space. Queries map into points in the same high-dimensional space. To determine whether a query is similar to an item, we determine whether the query point lies within the item's geometric region, or equivalently, whether the query point was likely to be generated by the item's probability distribution.

A simple method for executing the search is to perform a linear scan: for each item, determine whether the item includes the query. For a large number of items, this linear scan becomes very slow. Therefore, we use *indexing*: a set of precomputations whose results are stored in addition to the items. The index should assist in finding overlapping items more quickly than linear scan, but without using an excessive amount of extra memory.

This paper introduces Redundant Bit Vectors (RBVs): an indexing technique that allows us to quickly perform high-dimensional search. We show that RBVs excel at applications where many of the queries do not correspond to database items.

1.1 High-Dimensional Search in Audio Fingerprinting

We created RBVs for an important application: audio fingerprinting. A streaming audio fingerprinting system recognizes audio clips in an audio stream [1]. For example, such a system could recognize songs playing on the radio or in a bar. Audio recognition must be robust to distortions and noise in the audio stream: for example, radio stations compress broadcast songs in order to fit in more advertisements.

Audio fingerprinting can be mapped into high-dimensional search by converting a segment of audio into a vector of features that form the high-dimensional space. A database of known audio clips then get mapped into points in that high-dimensional space. For each clip, there is a sphere of acceptable distortions around each point: common audio distortions will perturb the point by a known amount. When a stream is being recognized, the sliding windows of the audio are continually being converted into high-dimensional queries. This stream of queries is compared to the database. When a query approximately matches a clip, the stream is identified.

Note that the vast majority of queries to the audio clip database are negative: there is no match in the database. That is because the database only stores a few samples from each song, to save space in the database. The stream away from those samples do not match anything in the database, and hence are negative queries.

It is important that the audio fingerprinting database be efficient in both time and space: it must store millions of songs in a database, and recognize queries for thousands of simultaneous users. Therefore, any indexing of the database must be smaller than the original database, while simultaneously speeding up search dramatically.

RBVs are applicable to audio fingerprinting, but are a generic technique. When machine learning is applied to signals, such as audio, images, or video, large fractions of the input are not relevant. When these irrelevant inputs are compared to a database (for detection or recognition), they should be rejected. We thus believe that high-dimensional search with negative queries should be common in applications of machine learning to signals. Fragment-based recognition of objects in images is another example of an application of such high-dimensional search [2].

1.2 Structure of Paper

Section 2 starts with a definition of the high-dimensional search problem, and Section 2.1 transforms that problem into a more tractable problem. Section 3 describes our RBV algorithm, including pseudo-code described in Section 3.3. Section 4 puts RBVs in the context of previous work. Section 5 compares RBVs to the best of the previous algorithms, on artificial data (Section 5.1) and real audio fingerprinting data (Section 5.2). We discuss extensions and future work in Section 6 and conclude in Section 7.

2 Definition of Problem

High-dimensional geometric search problems can be generated in several different ways. Raw data items can be extremely high-dimensional or not even lie in a vector space: they typically need to be pre-processed before they can be stored in a database. This pre-processing can be carried out by random projection [3][4], by principal components analysis, or by oriented principal components analysis [5]. If the data does not lie in a vector space, it can still get mapped to a vector space by kernel PCA [6] or multi-dimensional scaling [7].

Once the input space is chosen, the search problem can be formalized. Let i identify the items in the database ($i \in \{1..N\}$). Let $\boldsymbol{x} \in R^d$ be the location in the input space of a point with unknown label. Call this point with unknown label the *query*. Let $\boldsymbol{y}_i \in R^d$ be the location of the ith stored item. Let L_i be the label of the ith stored item. We wish to label $\boldsymbol{x}$ with a single label L_i for some i, or label it as "junk" (not in the database). Assume we have models for $P(\boldsymbol{x}|L_i)$ and $P(\boldsymbol{x}|\text{junk})$, with prior probabilities of label L_i and junk being $P(L_i)$ and $P(\text{junk})$. Then, simple decision theory leads us to the following search problem:

Problem 1. **Decision-theoretic search:** Find at least one i such that

$$P(\boldsymbol{x}|\text{item } i)P(i) > P(\boldsymbol{x}|\text{junk})P(\text{junk}). \tag{1}$$

Label $\boldsymbol{x}$ with L_i if it exists, or else label it junk.

Strictly, decision theory would have us find the L_i with the highest posterior. However, in practice, it is rare that a query has two labels that are more likely than junk. Therefore, for efficiency reasons, we allow the search to find one item that is more likely than junk, then allow it to stop.

Very often, we do not have a good model for "junk". We often assume that it is globally constant, or is constant in the region of high density for each item. We can use this assumption to create our first geometric search problem:

Problem 2. **Implicit surface search:** Find at least one i such that

$$f_i(\boldsymbol{x}) < c_i. \tag{2}$$

Label $\boldsymbol{x}$ with L_i if it exists, or else label it junk.

---

where f_i is a monotonic decreasing function of the likelihood, e.g., $-\log(P(\boldsymbol{x}|L_i))$.

This implicit surface search is now in the realm of geometry. Each function f_i specifies a blob in the input space. We want to find whether any blob overlaps the query point.

Computing a general implicit surface (e.g., for a mixture of Gaussians) could be very computationally expensive. Also, we frequently do not have enough data to fit a complex model for every item. Instead, let us use a single spherical Gaussian per item. Combining with the negative log function yields our primary geometric search problem:

Problem 3. **Sphere overlaps point:** Find at least one i such that

$$||\boldsymbol{x} - \boldsymbol{y}_i||_2^2 < R_i.$$

Label $\boldsymbol{x}$ with L_i if it exists, or else label it junk.

2.1 Transforming Regions into Hyperrectangles

Problems 1, 2, and 3 are difficult to index, because all of the dimensions are coupled. That is, for points near the decision boundary, an index needs to know all of the coordinates to precisely determine which side of the boundary the point lies on. In high dimensions, most of the volume (or probability mass) of an item lies near the boundary of the region. Therefore, it is difficult to use lower-dimensional coordinates to determine whether a point is inside or outside of a high-dimensional region.

Our idea is to approximate the determination of the decision boundary surface. All problems stated so far are inexact: they allow a non-zero false negative rate — some items that truly match to item i will be excluded, even with perfect indexing. Therefore, let us use a method that may introduce more false negatives, but with a tunable parameter.

The idea is to approximate problems 1, 2, and 3 by the following problem:

Problem 4. **Hyperrectangle overlaps point:** Find at least one i such that

$$\max_n \left| \frac{x_n - y_{in}}{\epsilon_{in}} \right| < 1$$

where x_n is the nth component of $\mathbf{x}$. Label $\boldsymbol{x}$ with L_i if it exists, or else label it junk.

where y_{in} is the coordinate of the ith item in the nth dimension.

For spherically symmetric distributions or regions (such as Problem 3), we can reduce the number of parameters per item to 1, with

Problem 5. **Hypercube overlaps point:** Find at least one i such that

$$\max_n |x_n - y_{in}| < \epsilon_i$$

Label $\boldsymbol{x}$ with L_i if it exists, or else label it junk.

In general, we select the parameters ϵ_{in} or ϵ_i to be "too tight." That is, we can dramatically increase the indexability of the data set by approximating the sphere, regions, etc. with hypercubes that do not enclose the entire region defined in Problems 2 or 3.

If we have access to the original probability density for item i (in Problem 1), then we can use Monte Carlo: generate points from the distribution $P(\boldsymbol{x}|L_i)$ and find a hypercube or hyperrectangle that encloses $1-\delta$ of the generated points, if the desired false negative rate due to indexing is δ.

Alternatively, if the original probability distribution is unknown and we are solving Problems 2 or 3, we can generate Monte Carlo samples from a uniform distribution over each region, and then choose a hypercube or hyperrectangle that encloses $1-\delta$ of those samples.

Using these tight hyperrectangles yields suprisingly small regions to index. This can be seen in the following two examples. First, consider Problem 1, where $P(\boldsymbol{x}|\text{item } i)$ is a Gaussian with unit variance. If we convert this problem to use hyperspheres (Problem 3), the radius of the hypersphere that encloses $1-\delta$ of the probability mass is simply

$$R = \sqrt{\left(\chi^2_d\right)^{-1}(1-\delta)}, \tag{3}$$

where $(\chi^2_d)^{-1}(1-\delta)$ is the inverse of the cumulative chi-square density function with d degrees of freedom evaluated at $1-\delta$, $d =$ the dimensionality of the input space. In contrast, the ϵ for a hypercube that encloses $1-\delta$ of the probability mass is simply

$$\epsilon = \mathcal{N}^{-1}\left(1-\frac{\delta}{d}, 0, 1\right) \tag{4}$$

where $\mathcal{N}^{-1}$ is the inverse of the cumulative Gaussian density function.

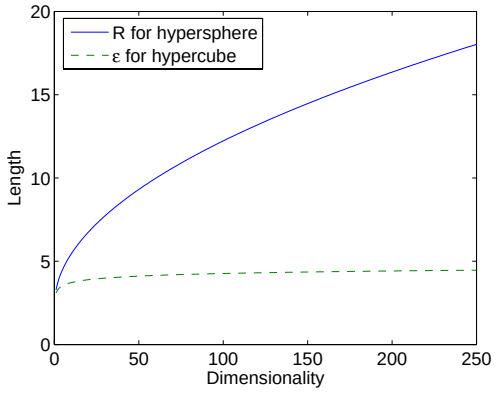

Fig. 1. R for the hypersphere and ϵ for the hypercube that encloses 99.9% of the probability mass of a unit spherical Gaussian.

The R of a hypersphere and the ϵ of the hypercube are plotted in Figure 1, for $\delta = 0.001$. As can be seen, the diameter of the sphere than mostly encloses the Gaussian grows as $\sqrt{d}$, while the sidelength of the hypercube grows very slowly, for $d > 30$.

This surprising result indicates that the vast majority of a Gaussian distribution has very low L_∞ distance to the mean. Another way of thinking about this result is that samples from a Gaussian are very likely to be have distance from the origin near $\sqrt{d}$, for large d. It is very unlikely for a sample to have one dimension near $\sqrt{d}$ while all other dimensions are near zero. Most of the probability mass of a Gaussian is in the corners, away from the coordinate axes, because there are many more corners than axes in high-dimensional space.

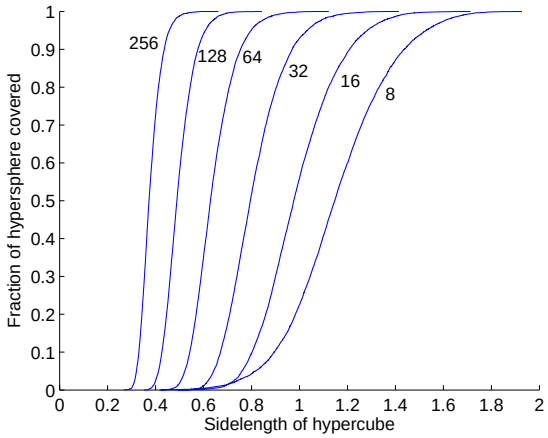

Fig. 2. Fraction of unit hypersphere covered as a function of the sidelength of the hypercube

To show that this result is not specific to Gaussians, we present Figure 2. This Figure is computed for Problem 3, where we are trying to fit a hypercube that encloses a large fraction of the volume of a hypersphere.[1]

The Figure shows that the size of the hypercube that encloses 99.9% of a hypersphere is substantially smaller than the size of the circumscribing hypercube (which would have sidelength=2), which would have a false negative rate of 0. Notice the dramatic savings by using a tight hypercube: in 256 dimensions, we can save a factor of 10^{134} in volume by using a tight hypercube instead of a circumscribing hypercube. This will both reduce the false positive rate and

[1] We generated Figure 2 by drawing uniform random samples from a hypersphere [8], then sorting them by their L_∞ distance to the origin. The x-axis of Figure 2 is then the L_∞ distance and the y-axis is the order in the list (scaled from 0 to 1).

dramatically reduce the number of hypercubes needed to be searched (as will be seen in section 5).

3 Redundant Bit Vectors

The most straightforward way to solve Problems 4 or 5 is by a brute force method called "linear scan". In a linear scan, the bounds of each data item's hyperrectangle are checked to determine if the query point $\boldsymbol{x}$ lies within the hyperrectangle. Linear scan potentially involves performing $2N$ comparisons.

As an alternative, this paper introduces a technique that, while still linear in the number of operations, performs the search much faster than a linear scan. This is accomplished through the careful use of redundancy and precomputation.

3.1 Partition the Query Space

Non-redundant techniques for searching high dimensional data fail because they uniquely assign data entries to buckets/pages. Since, for truly high dimensional data, locality properties of data are weak [9], there is no one way of grouping like data entries together in a way that is frequently helpful during searching. Recent strategies to cope with this situation involve using redundancy to group data entries together in multiple ways such that a useful grouping frequently exists when the structure is queried [10, 4].

In this tradition, we propose a way of generating redundancy that is particularly suited to high dimensional point queries over hyperrectangles. More specifically, we propose a redundancy strategy based on partitioning the query space rather than the data space.

In each dimension, we can compare the coordinate of the query point to intervals spanned by each of the stored item hyperrectangles. Per dimension, only a subset of the hyperrectangles overlap the query point. Now, notice that if we move the query point by a small amount, the hyperrectangles that overlap it stay largely constant: perhaps a few are added or dropped.

We can exploit this spatial structure by creating m bins (or intervals) of possible query points per dimension and pre-compute which hyperrectangles can overlap each query bin. These precomputed hyperrectangles are then stored in a bucket associated with the interval. Our index consists of the set of dm stored bins and buckets. For any query that lands in a bin, the set of hyperrectangles stored in the bucket is a superset of the hyperrectangles that overlap the query. We will use the buckets to systematically exclude possible items from our search. Thus, the grouping of queries into bins does not introduce false negatives.

Creating an index out of precomputed overlaps is a method for creating redundancy. To understand why, consider a single item and a single dimension. The hyperrectangle for that item appears redundantly in many buckets for that dimension. A query only accesses one bucket, so the algorithm does not need to backtrack to find all the hyperrectangles that may overlap it.

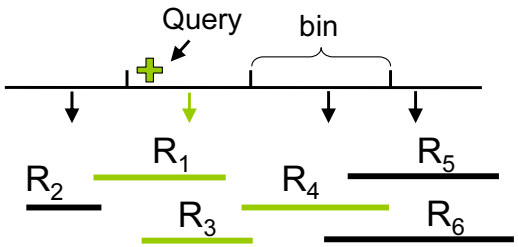

Fig. 3. This figure shows one dimension. In each dimension, the location of the query (the cross) gets assigned to one bin. Knowing the query lands in this partition eliminates certain rectangles: R_2, R_5, R_6.

For example, in Figure 3, the dimension shown is partitioned into 4 bins. There are six hyperrectangles in the dataset labeled $R_1 - R_6$. Each bin is associated with a set of hyperrectangles that represent all possible answers to queries that land in the bin. Therefore, the bucket associated with the second bin, which corresponds to the highlighted arrow in the drawing, contains the three rectangles R_1, R_3, and R_4. Note that the first bin also contains R_1, which means that R_1 is represented twice (once in each bucket). Of course in this figure, the partition boundaries are given. In general, we will choose these partition boundaries using a fast heuristic described in Section 3.3.

Now that we have our index, when we run a query, for each dimension, we find the bin that contains the projected query point (e.g. the highlighted bin in Figure 3). This can be done with either a linear or binary search of bin boundaries. We then select the associated bucket of hyperrectangles, which is guaranteed to contain a superset of the correct hyperrectangle answers. Then we perform set intersection on all the selected hyperrectangle buckets, producing a smaller superset of the correct answer. Note that the result of the set intersection is all possible hyperrectangles that overlap a hyperrectangle in query space.

Even though each dimension may not be very selective, the intersection of the selected dimensions is quite small, since the selectivity of the final intersection is close to the product of all the individual dimensions' selectivities. For instance in our audio fingerprinting application, individual dimension selectivities ranged from $50-90\%$ (see Figure 4) while the selectivity after the intersection was much less than 1%.

For each query, this procedure leaves us with a superset of the correct answer. Since the superset is small compared to the original dataset, we can compute, for each candidate, whether the query point is actually contained in each candidate hyperrectangle without incurring the cost of a linear scan.

3.2 Indexing with Redundant Bit Vectors

While the above strategy for using the index to generate and prune candidate answer sets may sound reasonable at first glance, careful examination yields some

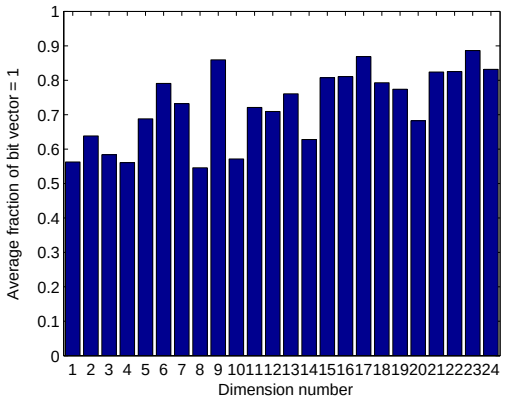

Fig. 4. For the audio fingerprinting task, average fraction of "1" bits in each bit vector, per dimension, weighted by bin selection frequency.

problems. For instance, assume the buckets of hyperrectangles are represented using arrays of 4 byte data IDs. Given that most buckets found will contain most of the data entries, the amount of data that must be sorted and combined for set intersection is close to the amount of data in the dataset. Since sorting is a superlinear operation, this procedure will almost certainly be slower than a linear scan of the data.

In order to deal more effectively with these buckets, we must use a representation for the sets that is both more compact, and allows for linear time set intersection. Fortunately, bit vector indices [11] provide us with such a representation.

Instead of representing each bucket using an array of IDs, each bucket will be represented using a string of N bits. Each bit represents the presence of an item in the bucket. More precisely, the kth bit of the string is a 1 iff the item with ID k is in the set.

Figure 5 shows how bit indices are used in our example from Figure 3 to represent the buckets associated with the given bins. Note that, as mentioned earlier, R_1 is in both the first and second bucket. As a result, the first bit in the associated bit indices are both set to 1.

A nice consequence of using bit indices to represent the hyperrectangle buckets is that set intersection now becomes a linear bitwise intersection of the associated bit strings (e.g. logically AND all the bit strings together). Note that these linear bitwise operations are very efficient since given a CPU with 4 byte registers, set intersection between 2 sets for 32 data items is performed with 1 CPU operation. A 64-bit processor can process set intersections twice as quickly. Also, given the ordered nature of performing set intersection, excellent memory subsystem performance is achieved. Memory accesses become linear in nature and cache misses are relatively rare, leading to very high bandwidth from the

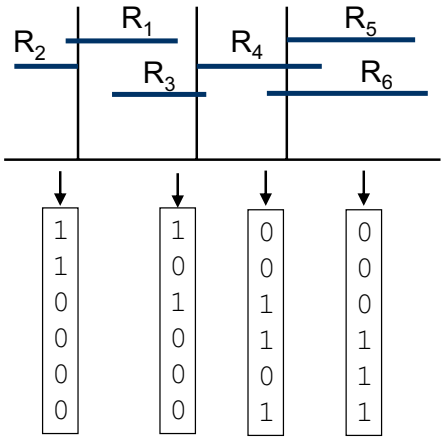

Fig. 5. For each bin, store whether each rectangle overlaps that bin in a bit vector: the ith bit represents whether the hyperrectangle associated with the ith item in the database overlaps this query bin.

memory subsystem. Finally, for very dense sets, the individual sets use approximately 1/32 of the space of the ID array approach. All these properties cumulatively work to produce an algorithm for set intersection that, while linear in the number of data entries, is extremely efficient.

The final algorithm for performing a query using our index is shown in Algorithm 1. The first *for* loop loads the appropriate bit index from the first dimension into C. Note that the arrays lo and hi will be explained in more detail later: the algorithm functions as if $lo = 1$ and $hi = N$. The second loop iterates over each dimension beyond the first. For each dimension, the appropriate bit index is selected and intersected with C. The final loop iterates over C, which contains the result of all the index intersections, and performs a final test for each remaining item.

The bit vector representation highlights the redundancy that arises from query partitioning. If we examine the bits for a single item across all bins for one dimension, they look like

$$\underbrace{0000000}_{\text{item above query}}\underbrace{1111111111111}_{\text{item overlaps query}}\underbrace{000000}_{\text{item below query}}$$

Thus, the span of the item hyperrectangle (relative to the bin boundaries) is stored in a redundant binary code. Only one bit of this binary code need be accessed for each query.

Algorithm 1 Querying the Redundant Bit Vector index

Require: Database of N items/distributions, Bit vector index B, Bin edge array E, hi and lo arrays, number of indexed dimensions I, query vector x
  j = smallest index such that $x_1 < E_{1j}$
  for i =lo[j] to hi[j] **do**
    $C_i = B_{1ji}$
  end for
  for $k = 2$ to I **do**
    j = smallest index such that $x_k < E_{kj}$
    for i =lo[j] to hi[j] **do**
      $C_i = C_i \& B_{kji}$ (once every machine word)
    end for
  end for
  for all i such that $C_i = 1$ **do**
    if $x \in$ item$_i$ **then**
      Finished
    end if
  end for
  If no item found, return empty

3.3 Details and Notes

We have not yet described how the partition boundaries are determined. We employ a heuristic that is designed for a particular dimension, to evenly spread the selectivity amongst all indices for that dimension. This uniform selectivity is accomplished by keeping the Hamming distance between adjacent bit indices roughly constant. More precisely, our heuristic first sorts the interval boundaries of the projected data, resulting in a sorted list of $2N$ interval boundaries. Given a user defined number of partitions Q per dimension, we divide the sorted list into Q maximally equal sized pieces and choose as the partition boundaries the last elements of the first $Q - 1$ pieces.

We are now ready to present the complete index-building algorithm shown in Algorithm 2. The first *for* loop calculates and stores the partition edges as described previously. The space for the bit vectors is then allocated to ensure that the individual bit vectors are packed linearly in memory. The next *for* loop then calculates the actual bit vectors. Finally, the last loop sets *lo* and *hi*.

lo and *hi* are used to reduce the portion of the bit indices that have to be "*AND*"ed during query time. If, while querying, the first bit vector selected begins with 800 "off" bits (100 bytes of 0 values), there is no point in using the first hundred bytes of any of the indices for this query, since the first index will ensure that the first 100 bytes of the result is all zeroes. Therefore, for the first dimension, for each bit index, we keep track of the leading and trailing number of "off" bits and use this to reduce the work that must be done during querying. This technique is made most effective by always choosing the most selective dimension for the first dimension, and by sorting the dataset by the first dimension. The result is that the "on" bits tend to cluster together somewhere

Algorithm 2 Building the Redundant Bit Vector index

Require: Database of N $d-$dimensional vectors x_{ij}, hypercube half-sidelength ϵ_i for each vector, number of bins per dimension Q, number of indexed dimensions I.
  Sort database by increasing value of most selective dimension
  Initialize temp array $t[2N]$
  for $k = 1$ to I **do**
    for $j = 1$ to N **do**
      Append $x_{jk} \pm \epsilon_j$ to t
    end for
    Sort t
    for $j = 1$ to $Q - 1$ **do**
      Bin edge array $E_{kj} = t_{\lceil 2jN/Q \rceil}$
    end for
  end for
  Initialize bitvector $B[I, Q, N]$ (last dimension packed into machine words)
  for $k = 1$ to I **do**
    for $j = 1$ to N **do**
      Initialize B_{k*j} to all 1
      for all i such that $x_{jk} + \epsilon_j \leq E_{ki}$ **do**
        $B_{k,i+1,j} = 0$
      end for
      for all i such that $x_{jk} - \epsilon_j \geq E_{ki}$ **do**
        $B_{kij} = 0$
      end for
    end for
  end for
  for $j = 1$ to Q **do**
    lo[j] = index of first "on" bit in B_{1j} (rounded down to word boundary)
    hi[j] = index of last "on" bit in B_{1j} (rounded up to word boundary)
  end for

in the bit index. For instance, the first bit index of the first dimension has many trailing zeroes while the last bit index of the first dimension has many leading zeroes.

4 Previous Work

There is an extensive literature in speeding up high-dimensional search. In order to understand the related work, we must transform Problem 3 into one of two related problems. First, Problem 3 is equivalent to the following problem, if all of the spheres' radii are the same:

Problem 6. **ε-range search** Find at least one i such that

$$||\boldsymbol{x} - \boldsymbol{y}_i||_2^2 < R$$

Label $\boldsymbol{x}$ with L_i if it exists, or else label it junk.

There is an extensive literature in approximately solving Problem 6 (e.g., [12]). This Problem has been more extensively studied than Problem 3, because it permits the database to store $\boldsymbol{y}_i$ as points, and consider the query as a sphere around the point $\boldsymbol{x}$. Thus, only points need to be indexed, not regions.

The research into Problem 6 has culminated in Locality-Sensitive Hashing (LSH) [4], which has sublinear time performance. We compare RBV to LSH in Section 5, below.

LSH operates by randomly projecting an input space into a number of dimensions. This random projection can be accomplished by taking a dot product with a vector drawn from a spherical Gaussian distribution with unit variance. Each projected dimension is divided into randomly-offset bins, whose width is proportional to the radius of the sphere in the original space. The bin number is then a hash function. A number of these hash functions, k, are then grouped together to form a key in a hash table: the k integers hashed together with an additional hash function is the key. The index of all items that match that hash table key are then associated with the key.

LSH maintains a redundant data structure by constructing l different hash tables in this way, each with their own random projections and random bin offsets. All l of these hashtables are checked, until either one item that solves Problem 6 is found, or else all hashtables are searched.

4.1 Nearest neighbor search

If we start to solve Problem 6 for small r, and then gradually increase r if no points are found, then we are solving:

Problem 7. **Nearest neighbor search**

$$c = \arg\min_i ||\boldsymbol{x} - \boldsymbol{y}_i||_2^2$$

Label $\boldsymbol{x}$ with L_c.

Again, there is an extensive literature in exactly or approximately solving nearest neighbor search in logarithmic time [13][14][15][16][17]. However, these techniques always force the query to map to one item, even if the item does not match well. Thus, the methods can spend a lot of index time and space finding the nearest neighbor, even when the query is clearly junk. Also, these methods often have exponential dependence on dimensionality. Therefore, we do not test these methods in this paper.

5 Speed and Memory Comparisons

In order to test the effectiveness of RBVs, we tested them versus LSH and linear scan. We perform two major tests: first, we test on an artificial dataset, which can be parameterized to explore dependency on the number of database

items, dimensionality, and the radius of the sphere; second, we test on real audio fingerprinting data, to check the effectiveness on a problem that we care about.

Both RBV and LSH have a tuning parameter that performs a memory vs time trade-off. For RBVs, the number of indexed dimensions can affect the speed, while for LSH, the number of hashtables can affect the speed. For both RBV and LSH, the parameter was tuned to optimize speed and the optimal speed was reported.

All experiments were performed on an unloaded Xeon 2.4 GHz processor running Windows Server 2003 with 1.5 GB of physical memory. All real numbers were stored as 4-byte floating point. Because the database of items must be loaded in RAM, we limited the size of any index to be 800MB. In the figures below, you can see the LSH memory consumption curves saturating at that level, which causes upward kinks in the LSH time curves.

In order to ensure a fair comparison, we optimized an LSH implementation. The most important implementation detail was to first build the hash tables for LSH using chaining, then copy the hash tables into a single array using linear probing, where the length of each chain is stored in a separate array. Since the index is only built once, this saves a large amount of memory: each entry uses only 16 bytes of memory (in contrast to [4], which used 60 bytes of memory per entry). As suggested by [4], we used a second hash function to disambiguate collisions in the main hash table (rather than checking for equality on the separate components of the first hash function). Following [18], we set the bin size on the output of each projection to be 4 times the radius of the sphere in the input space. Finally, as in the RBV linear scan, we halt the search through the hashtables when a match is found.

5.1 Artificial Data

To test indexing under various conditions, we generated an artificial training set, which consists of random deviates drawn from a spherical Gaussian distribution of unit variance. Each point becomes the center of a sphere. All spheres have equal radius.

We created two kinds of tests: "positive queries" and "negative queries." Positive queries are those which are very likely to have a match in the database, while negative queries are very unlikely to have a match.

We generated positive queries (following [4]) by adding Gaussian random noise to 1000 randomly selected points in the database. The variance of the Gaussian noise was selected to introduce a false negative rate of 10^{-3} (given a previously chosen radius).

We generated negative queries by generating 1000 randomly selected points from the unit spherical Gaussian distribution. The radius of the spheres was chosen to produce a false positive rate of 10^{-10}. Given this low false positive rate, it is very unlikely that new draws from the same distribution will end up inside any of the spheres.

The false positive and false negative rates were chosen to be realistic for a fingerprinting application: the false positive rate (per database item) must

be very low, because there can be tens of millions of items in a fingerprinting database.

We also tuned the RBV and LSH algorithms to produce a false negative rate of 10^{-3}, to be compatible with the underlying false negative rate of the problem. The false negative rate of the RBV method is tuned by choosing the size of the hypercubes. This is done by examining Figure 1 and scaling the unit Gaussian of the Figure to match the variance of the Gaussian used to generate the noise. For LSH, the false negative rate is tuned by the number of hashtables [19] via the formula

$$l = \lceil \frac{\log(\delta)}{\log(1 - p^k)} \rceil, \tag{5}$$

where l is the number of hashtables, δ is the desired false negative rate, k is the number of hash functions that are used to generate a key, and p is the probability that two input vectors that are the sphere radius R apart will collide under the same hash function. Since we are using Gaussian random projections in LSH, $p = 0.8$.

Vary database size, fix dimensionality, positive queries For the first experiment, we fix the dimensionality of the Gaussian to be 64, and the radius of all spheres to be $R = 5.6239$, to yield a false positive rate of 10^{-10}. For the RBV, we used a hypercube sidelength of 4.5767. We added Gaussian noise of variance 0.3020 to items in the database to generate queries. As we add items to the database, we measure how much time and memory each of the methods consume.

Our first experimental results are for positive queries, shown in Figure 6. Both RBV and LSH beat linear scan by a substantial margin. RBV grows linearly in time and space, being approximately 46 times faster than linear scan and requiring an average of 53% of the size of the database items to store the bit vector indices. RBVs reduced the number of items for final linear scanning by approximately a factor of 700 (compared to the total size of the database).

These positive queries are similar to the experiments performed in [4], which show sublinear time for LSH. This can be seen in the slope of the LSH line performance, which corresponds to a scaling of $O(N^{0.60})$. The speed of LSH surpasses BVs at approximately 300,000 items. Notice that, at 1,000,000 items, the memory required by LSH reaches the 800 MB limit, which causes LSH to be unable to use the optimal number of hash functions and tables, and therefore slow down.

Unlike RBV, the memory required by LSH grows superlinearly with database size. Fitting a line to the points in Figure 6 yields a memory requirement that scales as $O(N^{1.43})$. This can become an issue when scaling up to large databases: a 40 million item database would require 350 GB of RAM to store the hashtables, while the RBV method would require only 5 GB of RAM.

In this experiment, we test LSH in another way: we chose the number of hash functions k (hence the number of hashtables l) so that LSH takes up as much memory as the items in the database, then tested LSH's speed. Interestingly,

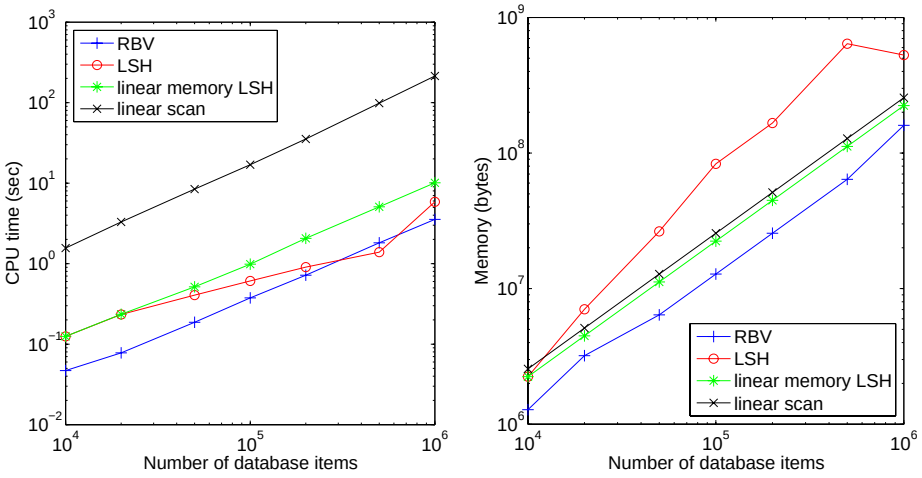

Fig. 6. Positive queries: Log-log plot of CPU time for 1000 queries (left) and memory (right) vs. database size, for RBVs, LSH, LSH limited in size to database size, and linear scan. In the memory plot, all schemes show the size of the index, except for linear scan, where the size of the database is plotted.

in this experiment, forcing LSH to take a linear amount of memory caused it to have linear time performance: RBVs are an average of 2.8 times faster than linear-memory LSH.

Thus, for situations where memory is constrained (as in very large databases), RBV is preferable to both LSH and linear scan.

Vary database size, fix dimensionality, negative queries Our next experiment is shown in Figure 7. Here, we present the more realistic scenario for fingerprinting-like search, where the vast majority of queries do not match in the database. For negative queries, RBV still maintains its superiority over linear scan. Here, the index requires as much memory as storing all the database items, and is 38 times faster than linear scan. RBVs reduced the number of items for final linear scanning by approximately a factor of 200.

For negative queries, LSH becomes much slower. Part of the reason why LSH is fast on positive queries is because it stops searching the hashtables when it finds a match. This is not possible on negative queries, so LSH must slow down. Performing a line fit to the first 4 points in Figure 7, we see that the time for LSH is still sublinear in time ($O(N^{0.65})$), and is still superlinear in memory ($O(N^{1.5})$), but with a much worse constant than RBV. Performing a possibly inaccurate extrapolation, we would expect LSH to be as fast as RBV for a database of 72 million items, but require 8 terabytes of RAM, while RBV would require 18 gigabytes. Clearly, LSH is not feasible in this situation. Thus, RBV is clearly

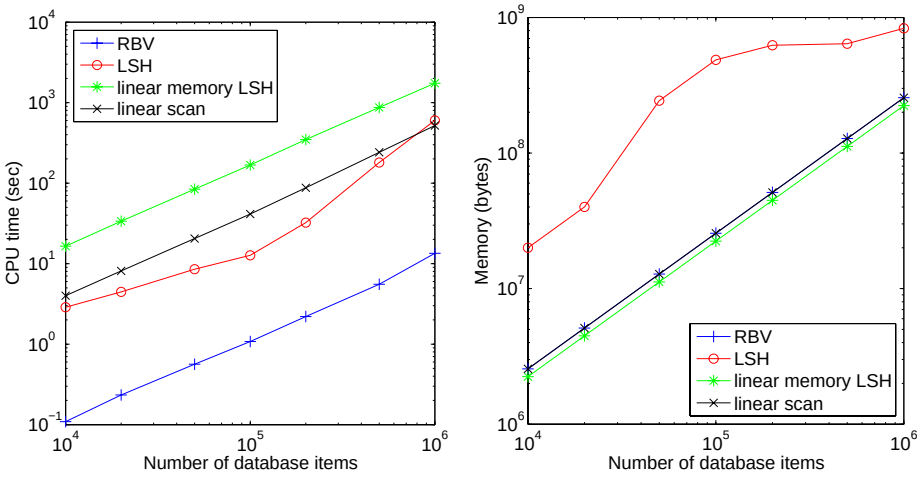

Fig. 7. Negative queries: Log-log plot of CPU time for 1000 queries (left) and memory (right) vs. database size, for RBVs, LSH, LSH limited in size to database size, and linear scan. Note that RBV and linear scan consumes the same amount of memory in this experiment.

the method of choice when negative queries make up even a small fraction of the input.

As in the positive query case, if we force LSH to have linear growth in memory, it has linear growth in time: RBV is 148 times faster than LSH with linear memory.

Fix database size, vary dimensionality To test the behavior of the algorithms in high dimension, we varied the dimensionality of the Gaussian that generated the items in the database while keeping the number of database items at 200,000. There are multiple ways to set the sphere radii as the dimensionality rises. We chose to keep the false positive rate of the system constant at 10^{-10}, and the false negative rate constant at 10^{-3}. These choices yielded parameters as shown in Table 1.

This choice forces the sphere radius to grow as a fraction of the mean interpoint distance, because the distribution of the interpoint distances are approaching a delta function. This growing radius makes the projections of the hypercubes ever larger, which interferes with efficient indexing, as can be seen in Figure 8. Here, linear scan is as efficient as LSH for $d = 128$, and as efficient as RBV for $d = 256$.

Under these severe conditions, all indexing techniques break down in high enough dimension. However, real high-dimensional problems may have more benign conditions than those in table 1: the techniques need to be tested to see if they are effective on a particular data set.

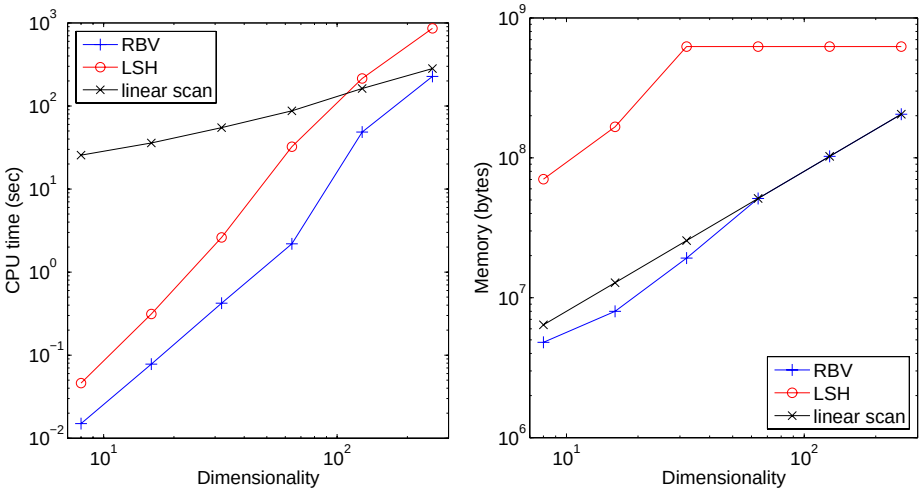

Fig. 8. Log-log plot of CPU time for 1000 queries (left) and memory (right) vs. dimensionality, for RBVs, LSH, and linear scan. Results shown for negative queries. False positive and false negative rates were held constant across dimensionality.

Dimensionality	Sphere Radius	Hypercube Sidelength
8	0.1674	0.2399
16	0.9313	1.1405
32	2.6768	2.7111
64	5.6239	4.5767
128	10.0834	6.4372
256	16.5662	8.1340

Table 1. Parameters used in dimensionality experiments

Fix database size, vary sphere radius As a final experiment on artificial data, we fixed the database size to be 200,000 items and the dimensionality at 64. We shrink the radius of the hyperspheres to see if RBV maintains its superiority over LSH, even for easy cases (at very low false positive rates). We maintain the indexing false negative rates at 10^{-3}, with a RBV sidelength of 0.5033 the sphere diameter.

The results of this experiment are shown in Figure 9, which shows that, for negative queries, all indexing techniques get faster when the false positive rate tends to zero. RBV maintains its superiority over both variants of LSH, although LSH starts to approach the RBV performance for small sphere radii: RBV is 14 times faster than LSH at radius 5.659, but only 3 times faster at radius 1.1314.

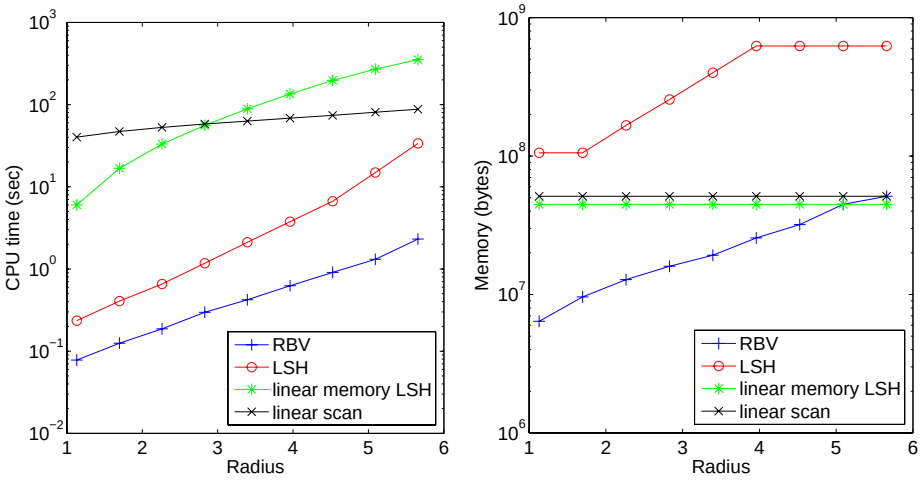

Fig. 9. Log-log plot of CPU time for 1000 queries (left) and memory (right) vs. radius of spheres, for RBVs, LSH, and linear scan. Results shown for negative queries.

5.2 Real Audio Fingerprinting Data

As a true test of the RBV method, we applied RBV to the audio fingerprinting task described in [1] and in Section 1.1. In this task, 6 seconds of audio are represented as a 64-dimensional vector. The database consists of 239369 items, each item taken from a unique song. An item is compared to a query with Euclidean distance.

For this test, we used realistic query data: 1000 queries taken from sequential frames of a single song.

Audio fingerprinting adds additional complexity to indexing methods: the radii of the spheres are not identical. According to [1], each radius is chosen to be 0.374 of the average distance of the item to the closest vector taken from 100 randomly chosen songs. This ensures that the false positive rate is more uniform across the database of items: the radii vary by a factor of 3. The false positive rate reported in [1] is roughly 10^{-6}.

RBVs can handle variable sphere radii in Algorithms 2 and 1. The LSH algorithm needs a fixed radius to compute the hash functions: we use the largest radius in the database for that radius.

As in the artificial data case, we set the false negative rate for 10^{-3} for both RBV and LSH. For RBV, this yields hypercube sidelength S_i:

$$S_i = 0.5033 D_i \tag{6}$$

where D_i is the diameter of the ith hypersphere.

Figure 10 shows the performance of RBVs, linear scan, and LSH on the real audio fingerprinting database. Because the audio fingerprinting data is not

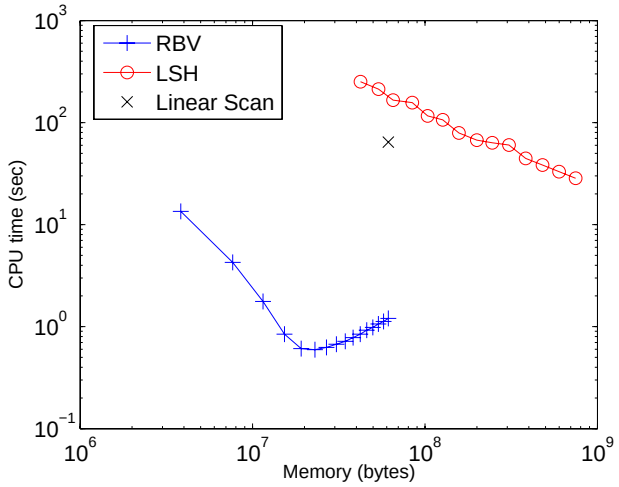

Fig. 10. Log-log plot of CPU time for 1000 queries vs. memory for RBVs, LSH, and linear scan (one point). Curves are generated by changing the number of indexed dimensions (for RBVs) and the number of hashtables (for LSH).

parameterized, we show the performance of RBVs and LSH as a function of their speed/memory tradeoff parameters: sweeping the number of indexed dimensions for RBV, and the number of hash functions used to construct a key in LSH.

For the audio fingerprinting task, RBV is much faster than other techniques. RBV is 109 times faster than linear scan and requires 3/8 of the memory to store the database items. RBV is 48 times faster and requires 34 times less memory than LSH.

We have previously tested SS-trees [15], and R-trees for this task [20], and found them to be worse than linear scan: we do not present those results here.

6 Extensions and Future Work

So far, this paper has assumed that the query is a point and that we want to find all data regions that contain the query point. We may also want to test which data regions spatially overlap a given query region. The query region can be mapped to a hyperrectangle in the manner described previously for mapping items. We now are testing overlap between a query hyperrectangle and all item hyperrectangles.

This test can be accomplished by building two sets of redundant bit vector indices. The first set divides the query space on the lower bound of a query hyper-rectangle while the second divides the query space on the upper bound. In other words, the answer sets associated with the lower bound index contain all data hyper-rectangles that may spatially overlap a query rectangle whose lower bound falls into the partition associated with that set. Similarly, the answer

sets associated with the upper bound index contain all data hyper-rectangles that may spatially overlap a query rectangle whose upper bound falls into the partition associated with that set. When a query is performed, 2 indices are picked per dimension; one for the lower bound of the query hyper-rectangle and one for the upper bound. As with previous strategies, all bit indices are "AND"ed together to generate a candidate list.

Possible future work also includes comparing redundant bit vector indices and LSH to R-trees and approximate kd-trees. A particularly interesting question is in the cases where LSH is competetive or better than redundant bit vector indices (for positive queries), do other techniques like R-Trees or kd-trees outperform both? This may be possible, since they are also sublinear techniques when applied to easier problems.

7 Conclusions

This paper has introduced Redundant Bit Vectors (RBVs), a method guaranteed to have linear time and memory complexity.

RBVs are built on three key ideas:

1. *High-dimensional geometry* — We use the fact that we can approximate hyperspheres (or more general regions) with hypercubes that are substantially tighter. This improves the selectivity of the indexing.
2. *Partition the queries, not the data* — We avoid the difficult task of partitioning the data by partitioning the queries into bins per dimension. These bins become "coarse queries," which can be precomputed and stored. These precomputations form a redundant binary code for the location and size of the objects. This redundant code means we do not need to backtrack a data structure to find objects.
3. *Bit vectors* — We store the precomputed indicies in bit vectors, which gives two advantages: first, we can become much faster than linear scan by exploiting the innate 32-way or 64-way bit-wise parallelism of modern CPUs. Second, our index is extremely compact and takes up less memory than the database of objects.

Realistic applications, such as audio fingerprinting, can have a large fraction of negative queries. RBVs are the first high-dimensional indexing scheme that is much faster than linear scan with a large fraction of negative queries. We showed that the best previous known algorithm (LSH) is slower than RBVs and requires an unrealistic amount of memory in this situation.

Acknowledgements

We would like to acknowledge Shimon Ullman, for suggesting that RBVs could be applied to his fragment-based object recognition task.

References

1. Burges, C.J., Platt, J.C., Jana, S.: Distortion discriminant analysis for audio fingerprinting. IEEE Transactions on Speech and Audio Processing **11** (2003) 165–174
2. Ullman, S., Sali, E., Vidal-Naquet, M.: A fragment-based approach to object representation and classification. In: Proc. of the 4th Intl. Workshop on Visual Form. Volume 2059 of Springer-Verlag Lecture Notes In Computer Science. (2001) 85–102
3. Achlioptas, D.: Database-friendly random projections. In: Proc. of the 20th Ann. Symp. on Principles of Database Systems. (2001) 274–281
4. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: Proc. of the 20th Ann. Symp. on Computational Geometry. (2004) 253–262
5. Diamantaras, K., Kung, S.: Principal Components Neural Networks. John Wiley (1996)
6. Schölkopf, B., Smola, A.: Learning with Kernels. MIT Press (2002)
7. Cox, T.F., Cox, M.A.A.: Multidimensional Scaling. Chapman and Hall (1994)
8. Tax, D.M., Duin, R.P.: Uniform object generation for optimizing one-class classifiers. Journal of Machine Learning Research **2** (2001) 155–173
9. Beyer, K.S., Goldstein, J., Ramakrishnan, R., Shaft, U.: When is "nearest neighbor" meaningful? In: Proc. of the 7th Intl. Conf. on Database Theory. Volume 1540 of Springer-Verlag Lecture Notes in Computer Science. (1999) 217–235
10. Goldstein, J., Ramakrishnan, R.: Contrast plots and P-Sphere trees: Space vs. time in nearest neighbour searches. In: Proc. of the 26th Intl. Conf. on Very Large Databases. (2000) 429–440
11. O'Neil, P., Quass, D.: Improved query performance with variant indexes. In: Proc. of the 1997 ACM SIGMOD Intl. Conf. (1997) 38–49
12. Arya, S., Mount, D.M.: Approximate range searching. In: Proc. of the 11th Ann. Symp. on Computational Geometry. (1995) 172–181
13. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Communications of the ACM **11** (1979) 397–409
14. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proc. of the 1984 ACM SIGMOD Conf. (1984) 47–57
15. White, D.A., Jain, R.: Similarity indexing with the SS-tree. In: Proc. of the 12th Intl. Conf. on Data Engineering. (1996) 516–523
16. Berchtold, S., Keim, D., Kriegel, H.P.: The X-tree: an index structure for high-dimensional data. In: Proc. of the 22nd Intl. Conf. on Very Large Databases. (1996) 28–39
17. Pagel, B.U., Korn, F., Faltusos, C.: Deflating the dimensionality curse using multiple fractal dimensions. In: Proc. of the 16th Intl. Conf. on Data Engineering. (2000) 589–598
18. Andoni, A., Indyk, P.: E2LSH 0.1 user manual. Technical report, Massachusetts Institute of Technology (2004) http://web.mit.edu/andoni/www/LSH/manual.pdf.
19. Cohen, E., Datar, M., Fujiwara, S., Gionis, A., Indyk, P., Motwani, R., Ullman, J.D., Yang, C.: Finding interesting associations without support pruning. IEEE Transactions on Knowledge and Data Engineering **13** (2001) 64–78
20. Goldstein, J., Platt, J.C., Burges, C.J.: Indexing high-dimensional rectangles for fast multimedia identification. Technical Report MSR-TR-2003-38, Microsoft Research (2003)