Indexing High Dimensional Rectangles for Fast Multimedia Identification

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

10/28/2003

Technical Report
MSR-TR-2003-38

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Abstract

This paper addresses the problem of quickly performing point queries against high-dimensional regions. Such queries are useful in the increasingly important problems of multimedia identification and retrieval, where different database entries have different metrics for similarity. While the database literature has focused on indexing for high-dimensional nearest neighbor and epsilon range queries, indexing for point queries against high-dimensional regions has not been addressed.

We present an efficient indexing method for these queries, which relies on the combination of redundancy and bit vector indexing to achieve significant performance gains. We have implemented our approach in a real-world audio fingerprinting system, and have obtained a factor of 56 speed-up over linear scan. Furthermore, the well-known Hilbert bulk-loaded R-Trees, a technique capable of searching low-dimensional regions, are shown to be ineffective in our audio fingerprinting system, because of the inherently high-dimensional properties of the problem.

1 Introduction

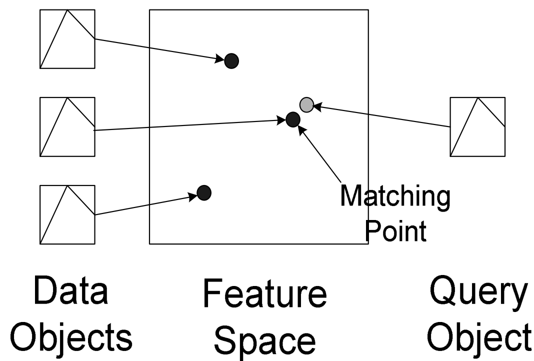

Figure 1: Performing a search

Multimedia identification and retrieval have become increasingly important problems. One important type of search problem in this domain is known as fingerprinting, or approximate searching, where a candidate object (e.g. a song) is compared to a set of objects in the database [1, 11, 12, 13]. As a result of the search, either a unique match is found, or no match exists in the data. These search problems are usually framed as some form of high-dimensional search, where data and query objects are mapped into the same high-dimensional feature space. For a particular query point, a match is discovered by finding a data point in the feature space which is close enough to the query point to be considered a match (see Figure 1).

More specifically, these approximate matching problems are usually framed in one of two ways. The traditional paradigm, which we call the fixed distance search problem, involves formulating the problem as a point query over spheres of fixed radius. The distance measure is usually some L_p metric, and the radius used is significantly smaller than the average interpoint distance (see Figure 2). This problem can then be mapped into an epsilon range search.

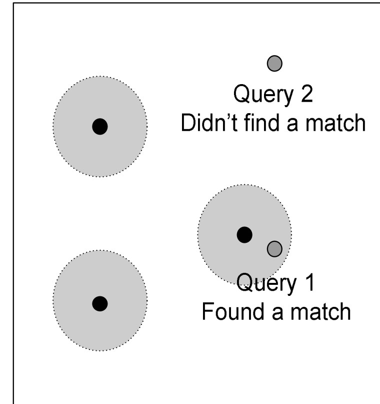

Figure 2: Fixed Distance Searching

Recent work, however, shows that large benefits are to be had by using different matching distances for different data points [1,21]. More specifically, using variable distances results in significantly improved accuracy of the overall fingerprinting system. The resulting search, which we call the variable distance search, is a point query over hyperspherical data (see Figure 3).

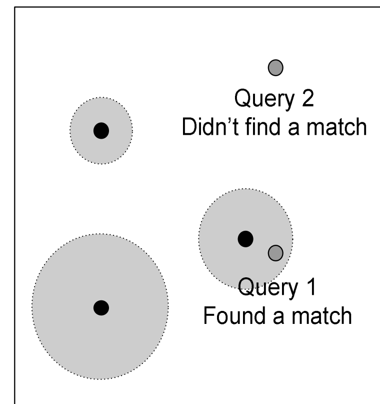

Figure 3: Variable Distance Searching

Traditional query processing strategies for solving the fixed distance search problem in high dimensionality typically suffer poor performance due to intrinsic difficulties associated with high dimensionality [2]. As a result, the most straightforward approach towards solving such problems, linear scan, usually outperforms more sophisticated approaches [3]. Alternate techniques to linear scan become even more problematic with variable distances.

In order to map the variable distance problem into an epsilon range search, we must replace all radii with the maximum radius in the database, and post-filter the results to account for variable radii. Note that enlarging the radii can produce an enormous increase in sphere hypervolume. For instance, enlarging radii by a factor of 2 in 64 dimensions increases sphere hypervolumes by a factor of 2^64. These problems associated with enlarging spheres can be avoided by framing the search problem as a point query against high-dimensional regions. Unfortunately, none of the strategies designed to handle spatial data (e.g. R-Trees [4]), have focused on high-dimensional scenarios.

This paper introduces the first technique that is capable of efficiently handling both the variable and fixed distance matching problems. While the introduced technique, which employs bit vector indexing ([6]), is linear in the number of data points, it nevertheless significantly outperforms linear scan. This is true even in situations where more traditional techniques are slower than linear scan. Furthermore, the approach presented here is faster than linear scan for both in-memory and disk resident datasets, although the primary focus of this paper is in-memory data, since the fingerprinting search problem considered here can fit easily into main memory.

This paper also presents a detailed performance evaluation based on the integration of these techniques into a real audio fingerprinting system, called RARE (Robust Audio Recognition Engine) ([21]). The performance of our technique is compared to both linear scan, which was the original implemented technique for RARE, and Hilbert bulk loaded R-Trees. Hilbert bulk loaded R-Trees were selected because of their competitive page pruning characteristics for searches over *regions.* Note that the severe penalty associated with disk seeks, which motivated work such as the X-Tree([20]) were not relevant as all searches were performed in memory. In addition, since the R-Tree was bulk loaded, the problems associated with dynamic insertions into R-Trees were also not a factor.

By implementing our strategy in RARE, we improved the throughput of the overall system by a factor of ~50. In addition, because of the fundamentally high dimensional distribution of the data, the R-Tree was unable to prune a single page during the search. The key insight which explains our performance results, is that our technique makes extensive use of redundancy by partitioning the queries rather than the data, and uses bit vectors to compress the resulting tuple ID lists. Because of bit vector indices' efficient representation of tuple IDs, the total size of our structures was smaller than the original dataset.

In order to achieve these performance results , we map fixed and variable distance searching into a related problem. We call this related problem the *rectangle search problem*, which can be defined as follows:

Definition 1**:**
Given:

- A static set of rectangles **R**
- A query point **Q**

Find the set of rectangles in **R** that contain **Q**.

While there are many efficient solutions for solving this well known problem in low dimensionality (e.g. R-Trees), no work has focused on solving this problem in high (e.g. higher than 10) dimensionality. This is due primarily to the lack of motivation for solving such problems, which do not arise in typical spatial database applications.

The rest of this paper is organized into five Sections. Section 2 describes the manner in which various approximate searches can be mapped into the rectangle search problem. Section 3 describes our solutions to the rectangle search problem. Section 4 discusses a case study when our techniques were integrated into a large-scale audio fingerprinting application. Performance is evaluated relative to both linear scan and Hilbert bulk loaded R-Trees. Section 5 discusses related work, and Section 6 concludes this paper.

2 Mapping Approximate Matching Problems into the Rectangle Search Problem

2.1 Fixed distance searching

There are several geometric search problems arising from approximate matching problems, that can be mapped into rectangle search problems. The most common of these problems are fixed distance searches (see Figure 2), which are isomorphic to the more commonly found epsilon range queries. In the epsilon range formulation, each complex object is mapped into a point in a high dimensional feature space. A match is then found for a query object by mapping that query object into the high dimensional feature space that the data objects are in. An epsilon range query is then performed as follows: if a data point is found that lies within epsilon distance of the query point (usually using some L_p metric), the data point is considered a match for the query point. Otherwise no match is found. If multiple data points are within epsilon distance of the query point, then either the closest is used, or the answer is considered ambiguous, and either no match is used, or all points are

used. Note that the following process can be used to translate such epsilon range problems into rectangle search problems (see Figure 4):

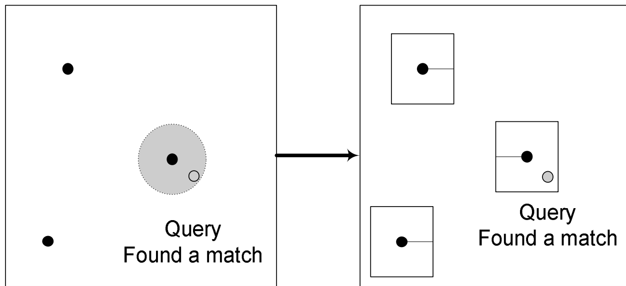

Figure 4: Conversion from Fixed Distance to Rectangle Search

Dataset transformation:

1. Replace each data point p with the smallest hypercube c that encloses the hypersphere with center point p and radius epsilon

Search transformation:

2. Instead of performing an epsilon range query over the data points, perform a rectangle search using the original query point over the hypercubes. Note that performing such a search is conservative, and can lead to false positives.
3. After performing the rectangle search, post-process the results to eliminate false positives.

Note that in high dimensional scenarios, it can be very useful to use hypercubes in step 1 smaller than described if a small number of false negatives (i.e. results incorrectly pruned) are allowed. This issue is explored both generally in Section 2.3 and with real data in the performance study in Section 3.

2.2 Variable distance searching

Some versions of epsilon range queries use different epsilons to match different data points [1,21]. The resulting problems are isomorphic to the variable distance searches described in Figure 3. These problems can be transformed to rectangle searches in a manner very similar to static epsilon range queries. The only difference is in step 1, where the size of hypercube c is dependent on the location of the data point. Thus step 1 becomes:

1. Replace each data point p with the smallest hypercube c that encloses the hypersphere with center point p and radius $epsilon_p$, where $epsilon_p$ is the value of epsilon at point p.

The results are shown in Figure 5.

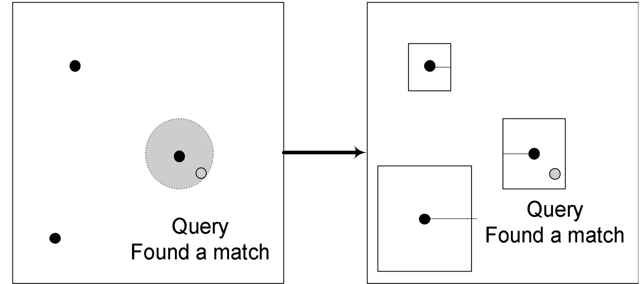

Figure 5: Conversion from Variable Distance to Rectangle Search

As in Section 2.1, it can be very useful to use hypercubes in step 1 smaller than the circumscribing hypercube if a small number of false negatives (i.e. results incorrectly pruned) are allowed. This can be a very important part of the mapping and is discussed in Section 2.3. We also make use of this observation for a particular application in Section 4. More specifically, the rectangle sizes are determined from examples of acceptable distortion along with a false negative objective.

2.3 Approximating Hyperspheres with Hypercubes

In Sections 2.1 and 2.2, we approximate hyperspheres with noncircumscibing hypercubes. We now show that this approximation is harmless in high dimensions.

Substituting a noncircumscibing hypercube for a hypersphere is useful and harmless if the hypercube contains almost all of the volume of the hypersphere and is substantially smaller that the circumscribing hypercube.

Figure 6 shows the fraction of volume of a unit hypersphere that is covered with hypercubes of differing side lengths L, for 6 settings of dimensionality. This graph shows that, for high dimensional search spaces, L can be substantially smaller than 2, which is the side length of the circumscribing hypercube. For example, in 256-dimensional space, a hypercube with L = 0.6 covers virtually all of the hypersphere, while being a factor of $0.3\^{}256 = 10^{134}$ times smaller than the circumscribing hypercube. This savings only relies on the properties of high-dimensional geometry, not on any specific application or data set.

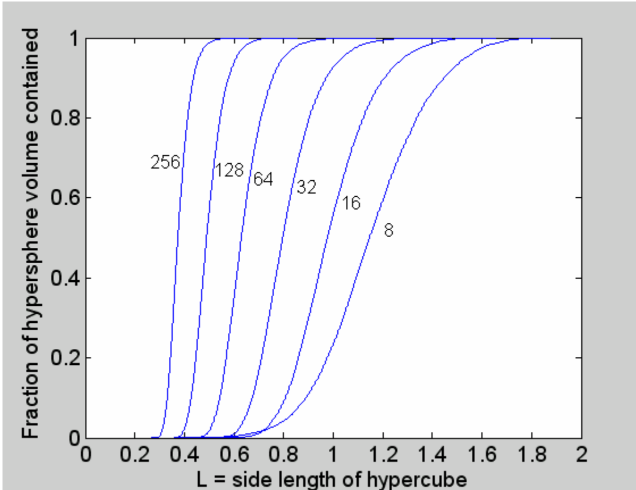

Figure 6: Fraction of unit hypersphere contained within hypercube with side length L.

Figure 6 was calculated using a Monte Carlo estimation technique. More specifically, the graph was was formed by drawing a sample of 10000 points uniformly distributed through a unit hypersphere centered on the origin [22]. The maximum coordinate for each of the 10000 points is one-half the side length of the hypercube that would enclose that point. Therefore, we sort the points by maximum coordinate, plot twice the max coordinate on the x-axis and the fractional position in the sorted list on the y-axis.

3 Efficiently Solving the Rectangle Search Problem

3.1 Problem setup

We begin this section by discussing the brute force method of solving the rectangle search problem, which is found in Definition 1.

The most straightforward way to solve this problem is to perform a linear scan. In a linear scan, the bounds of each data rectangle are checked to determine if **Q** lies within the rectangle. Linear scan potentially involves performing a compare for each bound value in **R**, and requires at least |**R**| compares.

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 aggressive use of precomputation in the form of bit vector indices.

3.2 Bit vector indices

Bit vector indices are typically used on relational data in situations where the data is either static or append mostly, such as data warehousing and OLAP applications [6], and are typically defined as follows:

Given:

- Every tuple **d** in a table **S** has an assigned unique identification number such that the ID numbers used are all integers between 1 and |**S**| (this is easy to maintain when the data is either static or append mostly).
- A boolean predicate **p** that can be applied to any tuple **d** in **S** (e.g. **p** is true if and only if the value of the first column in **d** is less than 40).

The bit vector index **b** over table **S** using predicate **p**, referred to as **b**=bitvec(**S**,**p**) is a list of binary digits such that the **n**th value of the list is 1 if and only if **p** holds for the tuple **d** with ID **n** in **S**. These bit vectors are typically packed into word arrays for fast linear access. For example, see Table 1, which shows how the bit vector index is computed over a table **S** using the predicate: Age<40. This index can then be used to find the tuple IDs that satisfy predicates in queries. For instance, assume the query is: "Find all tuples in **S** that satisfies the predicate: Age < 40 AND Salary > 45000". We can answer this query by doing the following:

1. Obtain the set of tuples associated with set bits in **b**.
2. Remove all tuples whose salary is less than or equal to 45000 from the remaining set.

Tuple ID/ Bit position	Tuple in **S**: (Age, Salary)	**p**=(Age<40)	Bit Vector Index Value
1	(18, 20000)	True	1
2	(43, 65000)	False	0
3	(35, 50000)	True	1
b=bitvec(**S**, **p**) = 101			

Table 1: Example of a bit vector index

In addition, multiple indices can be bitwise ANDed and ORed together when queries with complex predicates can be expressed in terms of the predicates used to build the bit vector indices. For instance, assume we have two bit vector indices, one of which is the index displayed in Table 1, called **b**. The other bit vector index is called **a**, and like **b**, is over table **S**, but uses the predicate **p** = Salary > 45000. As a result, the indices themselves are **b**=101 and **a**=011. The query "Find all tuples in S that satisfy the predicate: Age < 40 AND Salary > 45000" can be executed by

obtaining the set of tuples associated with set bits in the bitwise conjunction of **b** and **a**. Note that bitwise logical operands like AND and OR are extremely efficient as each CPU operation performs the logical operator for many tuples in a single CPU instruction (since CPU instructions operate on a whole word at a time). In addition, these strategies tend to have excellent cache performance.

Rect ID/ Bit pos	Rectangle in **S**: (X1,Y1)-(X2,Y2)	**p**=Overlaps[(X1,Y1)-(X2,Y2), (3,4)-(8,5)]	Bit Vector Index Value
1	(0, 0)-(10, 10)	True	1
2	(4,2)-(7,5)	True	1
3	(8,1)-(9,2)	False	0
b=bitvec(**S**, **p**) = 110			

Table 2: Example of a spatial bit vector index

The definition of bit vector indices above, which is the standard definition used today, applies only to relational data. We relax this requirement by allowing data values of **S** to be hyperrectangles, and we allow **p** to contain the hyperrectangle specific predicate overlaps. As a result, we call such indices spatial bit vector indices. An example of spatial bit vector indices are shown in Table 2. **p** in this case is an overlaps test between the constant rectangle (3,4)-(8,5) and a data rectangle**.**

3.3 Solution overview

The way we use the spatial bit vector indices described above to efficiently solve the rectangle search problem is outlined as follows:

- For each dimension, create **m** spatial bit vector indices over rectangle data.
- When a point query is performed with point **q**:
    - Using **q**, select exactly one spatial bit vector index for each dimension.
    - AND together the selected spatial bit vector indices from all dimensions, producing **c**.
    - Obtain the set of rectangles **T** associated with set bits in **c**. Because of the manner in which the bit vectors were created, and because of the selection criteria, there is a guarantee that all members of the answer set are in **T**.
    - Remove any extra rectangles in **T** by performing a linear scan of the contents of **T**. Assuming that |**T**| is significantly smaller than |**S**|, this is much faster than a linear scan over the whole dataset. Note that this step may be replaced by a scan over hyperspheres if the hyperrectangle search is a mapped hypersphere search..

Note that there are two aspects in the above strategy that need to be specified. The first is how the **m** indices per dimension are created. The second is how an index for a particular dimension is selected during search.

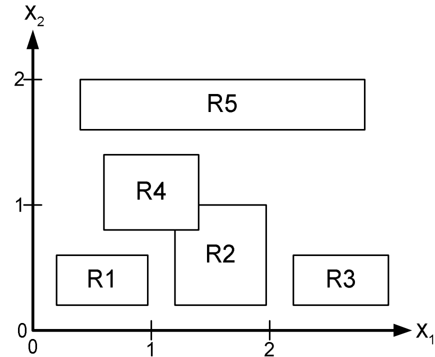

Figure 7: Rectangle Data Example

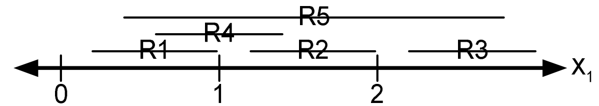

Figure 8: Projected rectangles

3.4 Creating and selecting the indices

The spatial bit vector indices are created by considering the projection of the data rectangles onto each individual dimension. For instance, consider the two-dimensional example depicted in Figure 7, and Figure 8, which shows the projection of those rectangles onto the x_1 axis. Figure 9 shows **m**=3 disjoint intervals on the x_1 axis that cover the entire axis.

For each interval, we construct a bit vector index such that the predicate is an overlap test between the interval and the projection of each data rectangle onto that axis. As a result, the bits associated with all data rectangles that intersect the interval are set to 1, and all others are set to 0 (see Table 3). Thus, each spatial bit vector index selects only the data rectangles that cannot be ruled out using the projected dimension alone for a query whose value lies in the specified interval. As a result, the selection criteria for each dimension is to select the bit vector index corresponding to the partition that includes the query point. Geometrically, the result of ANDing all the bit vectors selected in this way is that the query becomes a hyperrectangle that contains the

query point, which is why a post-processing step is necessary to remove the extra rectangles.

Our technique partitions the queries per dimension, such that each partition contains all rectangle IDs of hyperrectangles that overlap the query partition description. The bit vector indexes are just a way of compressing those rectangle ID lists so that the lists can be intersected efficiently and easily, analogous to [6].

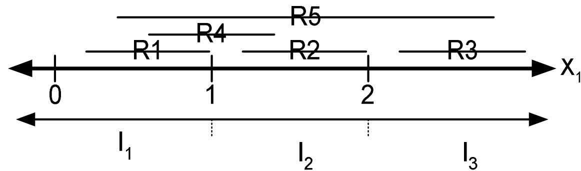

Figure 9: Dividing the X_1 Axis into 3 intervals

Of course we still need to specify how this partitioning is created. We select our **m**-1 dividers along the axis from amongst the boundary values of the rectangles. More precisely, we create those dividers every (2*|S|)/m boundaries, with any leftover boundaries spread out amongst the first intervals. The partitioning shown in Figure 9 was created with this heuristic, since each partition has (2*5)/3=3 boundaries with 1 left over. Thus we make our divisions on the (3+(1 left over))=4th boundary and the (4+3)=7th boundary.

Rect ID/ Bit pos	Rect in[] **S**	$\mathbf{P_1}$= Overlaps $[R_i,I_1]$	$\mathbf{p_2}$= Overlaps $[R_i,I_2]$	$\mathbf{p_3}$= Overlaps $[R_i,I_3]$
1	R_1	True	False	False
2	R_2	False	True	False
3	R_3	False	False	True
4	R_4	True	True	False
5	R_5	True	True	True
$\mathbf{b_1}$ = bitvec($\mathbf{S}$,$\mathbf{p_1}$)= 10011		$\mathbf{b_2}$ = bitvec($\mathbf{S}$, $\mathbf{p_2}$) = 01011	$\mathbf{b_3}$ = bitvec($\mathbf{S}$, $\mathbf{p_3}$)= 00101	

Table 3:Bitmap indices for dimension X_1 from Figure 9

There is one special case, however that we treat slightly differently. We may know *a priori* that all, or the great majority, of queries lie within some interval narrower than the whole axis. This can happen from either workload profiling, or because of the way the problem is set up (see Section 4). In these cases, when we generate our partitioning, we only consider rectangle boundaries that lie within the query interval, and completely ignore all other rectangle boundaries.

3.5 Inserts and Deletes

The strategy we recommend for handling inserts and deletes involves a quick update that ensures that all subsequent answers are correct, but gradually degrades the quality of the bit vector index.

When inserting a new data value, the ID given to that value is always one more than the last value already in the indices. This results in adding a bit to the end of every bit vector index. In order to achieve acceptable I/O performance for disk based searching, or good cache performance for in memory based searching, memory is added to the end of the bit vector indices in large chunks, and then gradually filled in by subsequent inserts. Note that in the disk-based scenario, it is not necessary to flush the modified pages to disk since a log can be used to aggregate all the changes and stream them out efficiently. This avoids many forced random I/Os (one for each index) associated with inserts. This is a known technique for inserting new tuples in bit vector indexes.

When deleting a data value, we rely on the fact that one index is guaranteed to be used for each dimension. As a result, we can pick a single dimension, and change any 1 to a 0 for all indexes for that dimension for the tuple ID to remove. This ensures that the appropriate row is always set to 0 during a search. Note, however, that over time, this can result in a high concentration of unused tuple IDs that take space in the bit vector index. As a result, in a workload where deletes are significant compared to inserts, occasional reorganizations will be needed to remove the deleted IDs from the indices.

4 Case Study: Audio Fingerprinting

4.1 System overview

This section describes an implementation of our techniques in the context of an existing audio fingerprinting system, RARE ([1,21]). In this system, 239,369 songs were fingerprinted. Each fingerprint consisted of a single 64 dimensional vector of floats plus an L_2 matching distance. As a result, the entire database is approximately 61 MB and can fit comfortably in memory. Each fingerprint is constructed from approximately 6 seconds of audio approximately 30 seconds into the clip.

When an audio clip is playing on a client, traces (generated in the same way as fingerprints) are generated every 186 milliseconds and submitted to the server, which matches the traces against the fingerprints in the database (see Figure 10). If the 64-dimensional Euclidean distance between the fingerprint in the database and the trace is within the L_2 matching distance associated with the database fingerprint, we have a tentative match, and a future fingerprint is used to confirm that we found an actual match. The system is highly accurate and achieves a false positive rate of 8×10^{-6}, per fingerprint, per clip, at a false negative rate of 0.8%, even in the presence of quite severe audio distortions ([21]). The great majority of time (> 99%) in the server is spent searching the 239,369 audio clips for an initial match.

The L_2 distances between incoming traces and stored fingerprints are measured in a feature space, where the map from the input audio to the feature space is learned using examples of distortion (the system is generally robust, however; the accuracy numbers mentioned above are for distortions that were not used for training). The per-point epsilon values (Section 2) were computed using a validation set, where the epsilon value is chosen to be a fixed fraction of the mean distance of the database point to the validation set.

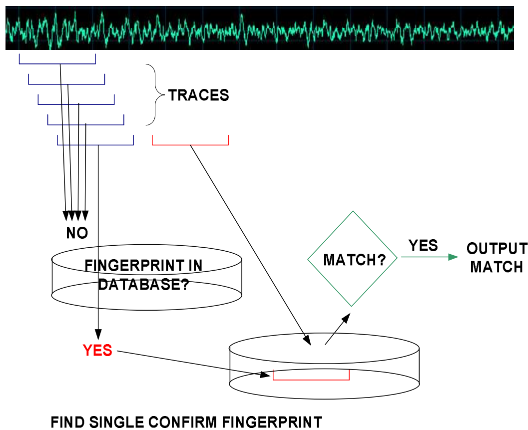

Figure 10: RARE Architecture

4.2 Searching the Fingerprints

Since the server spends the great majority of its time searching the in-memory database of fingerprints, we focus on that search. The search is best formulated as a point query (an individual trace) over a set of hyperspheres, where each hypersphere has a fingerprint as the center and an individual match radius. The linear scan done by the fingerprinting system has an early bailout. More specifically, as dimensions are added to the sum in the Euclidean squared distance, the current sum is compared to the smallest (squared) Euclidean distance found so far. If the sum is greater than this value, we know that the point currently being examined is not the closest, and the computation can skip to the next point. This early bailout results in approximately a factor of 2 speedup over the full distance computation in the linear scan.

The fingerprint centerpoints themselves are distributed in a truly high dimensional fashion, as is demonstrated by the fact that they are all approximately equidistant. For instance, the distribution of distances about a typical centerpoint is such that the median centerpoint is less than a factor of two further than the closest centerpoint. In addition, the radii are quite large when projected onto individual dimensions. Figure 11 shows, for each dimension, the percentage of centerpoints that overlap a typical hypersphere when projected onto that dimension. The dimensions are listed in increasing order of overlap. Obviously, there is a great deal of overlap in the projections, although there is almost no overlap in the 64 dimensional space.

Instead of performing a full linear scan, we perform a faster but less discriminating search before performing the linear scan with bailing. For the faster search, we first map the problem into a hyperrectangle search problem, and then measure a variety of approaches. These approaches include linear scan over the hyperrectangles, bulk-loaded R-Trees, and our spatial bit vector index based approach.

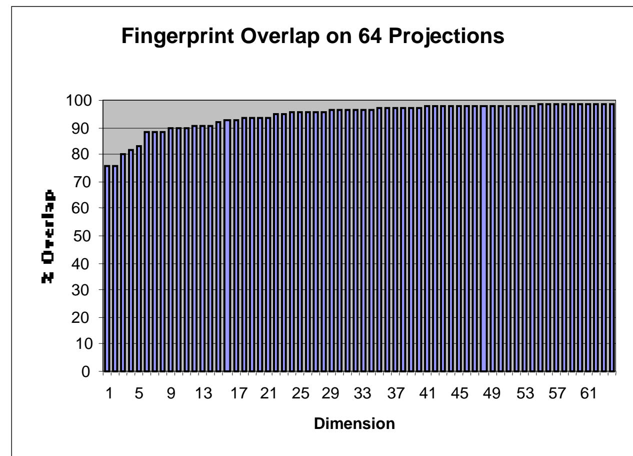

Figure 11: Percentage of centerpoints that overlap a typical hypersphere

4.3 Mapping the Fast Search

We derive the hyperrectangle search problem in the fashion described in Section 2.2. More specifically, for each fingerprint hypersphere, we create a corresponding hyperrectangle with the same centerpoint as the

hypersphere. If we wanted to be completely conservative, the side length of the hyperrectangle would be double the radius of the hypersphere. But since a very small number of false negatives are acceptable (in line with the overall false negative rate of the system to begin with), and because of the distribution of acceptable distortions, we can use a substantially smaller side length, as in Section 2.3.

After performing the dataset transformation described above, we have a set of hyperrectangles instead of a set of hyperspheres. The distribution of the centerpoints of these hyperrectangles is identical to that of the hyperspheres, but the projections of those hyperrectangles on individual dimensions is quite different due to the non-conservative approximation. Nevertheless, there is still significant overlap in the dataset, which is shown in Figure 12, which shows, for each dimension, the percentage of hyperrectangle centerpoints that overlap a typical hyperrectangle when projected onto that dimension. It is easy to see that while overlap is significantly improved, it is still quite high.

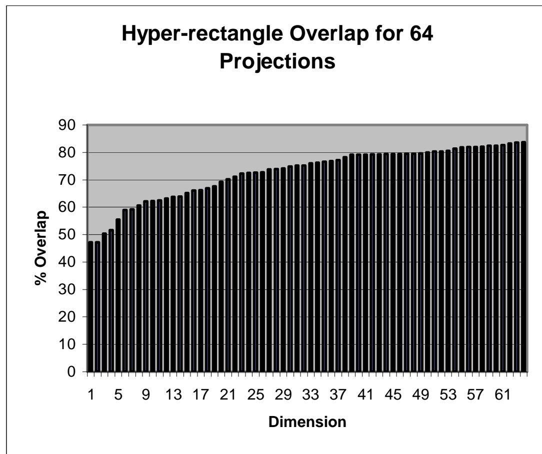

Figure 12: Percentage of centerpoints that overlap a typical hyperrectangle

Since we know we are going to remove false positives afterward, and since the first 14 dimensions yield the most selectivity (due to the process used to generate the features [1]), we use only the 14 dimensions with the lowest overlap and use post-processing to remove any false positives that come from not considering more dimensions. Note that the post-processing may involve filtering out the extra hyperrectangles, followed by filtering out the extra hyperspheres, or just directly filtering out the false positives using the hyperspheres. This is a performance consideration that we measure, below.

4.4 Fast Search Results

This section presents the various alternatives for performing the hyperrectangle search over the mapped audio fingerprint data. The total size of the original (hyperspherical) data is ~61MB. The size of the 14 dimensional mapped (hyperrectangular) data is ~27MB, while the total size of the bit vector indices is ~7MB, ~14MB, ~27MB, and ~54MB for the 4 settings of indices per dimension that we used. The search alternatives include a linear scan of the rectangles, searching a bulk loaded R-Tree, and our bit vector index based solution. GIST ([7]) was used for the R-Tree implementation, and Hilbert ordering([8]) based bulk loading was used. For our bit vector index based solution, we used 16, 32, 64, and 128 partitions per dimension. In order to ensure that our bit vector based approach produced the same answer as the alternatives, we eliminated false positives from the result set by scanning the candidate hyperrectangles. Note that in the audio application, however, we would scan the hyperspheres associated with the candidate hyperrectangles rather than scanning the hyperrectangles themselves.

For all experiments, after the indexes were created, 1000 queries were run and the wall clock time was measured for the 1000 query run. The minima and maxima for the hyperrectangles were represented as 4 byte floats. Since all structures fit in memory, there was not a significant amount of disk I/O. The results are shown in Table 4

Technique	Time for 1000 Queries(seconds)	Speedup Over Linear Scan
Linear Scan	29.5	1.0
Bit Vector Indices 16 Ind Per Dim	1.027	28.7
Bit Vector Indices 32 Ind Per Dim	0.7598	38.8
Bit Vector Indices 64 Ind Per Dim	0.6687	44.1
Bit Vector Indices 128 Ind Per Dim	0.631	46.75
Hilbert R-Tree	103	0.2864

Table 4: Hyperrectangle search performance

First of all, note that all R-Tree times are substantially more than linear scan times. This was due to predicate evaluation overhead in GIST. Nevertheless, not a single page was pruned during any of the 1000 queries, which is unsurprising given the extreme overlap in the projections of the data.

The bit vector indices based approach easily outperformed the alternatives. It is easy to understand the vast

improvement over linear scan when one considers the amount of data processed in each case. For the linear scan, the number of bytes examined per query was a function of the number of tuples (239,369), the size of a float (4 bytes), the number of dimensions (14), and the fact that both a maximum and minimum is examined per dimension. All together that is 239,369*4*14*2=26.81MB. The bit vector index based approach, however, scanned 14 indices, where each index contained 239,369/8 bytes. The total was, therefore, only 14*239,369/8=418,895 bytes. Therefore, the speedup in memory bandwidth was a factor of 64. Of course this improvement was diluted somewhat by the post-filtering step. Therefore the total speedup, for 128 indices per dimension was a factor of ~47. Also, notice that as we increase the number of indices per dimension, performance improves. This is due to the reduction in false positives, which ultimately leads to a smaller set to post-process.

Of course these timings, including linear scan, are for solving only the rectangle search problem. When our indexing technique is put in the context of the audio fingerprinting system, the post-processing is based upon a scan over the hyperspheres associated with the remaining hyperrectangles. Also, the linear scan performance is over hyperspheres rather than hyperrectangles. The results of running the modified bit vector based approach compared to a linear scan over the hyperspheres is given in Table 5. The overall speedup for searching the fingerprints is ~56.

Technique	Time for 1000 Queries(seconds)	Speedup Over Linear Scan
Linear Scan	32.5	1.0
Bit Vector Indices 16 Ind Per Dim	0.8791	37.0
Bit Vector Indices 32 Ind Per Dim	0.6693	48.6
Bit Vector Indices 64 Ind Per Dim	0.5992	54.2
Bit Vector Indices 128 Ind Per Dim	0.577	56.33

Table 5: Fingerprint search performance

5 Related Work

Many strategies have been proposed for indexing similarity/fingerprinting problems [14, 15, 16, 17, 18], these strategies can typically be used in two ways: nearest neighbor searching, and epsilon range searching. For the type of fingerprinting presented here, neither of these approaches are appropriate.

Nearest neighbor based solutions would suffer from poor performance due to the requirement that an answer be returned even when there are no points nearby (relative to other points in the dataset). This is backed up by the observation that for the dataset used in our case study, all fingerprint centerpoints are approximately equidistant, a scenario known to produce poor performance [2].

Epsilon distance based solutions are acceptable in situations where comparable neighborhoods around all fingerprints are used for matching. Unfortunately the data presented in our case study does not have this property.

There is a plethora of work on indexing low-dimensional regions. A good overview of this work can be found in [5]. This work is typically motivated by GIS and CAD applications, and is targeted towards two or three dimensions. As we saw in Section 1, it is unreasonable to expect such techniques to be effective. Of perhaps more interest is the work on one dimensional interval data (e.g. [9, 10]) as one could conceivably use these techniques on projections of the data in a manner similar to our bit vector based approach. Unfortunately, the overlap in the projection is so high as to make techniques without redundancy useless, where redundancy is defined as data items not assigned to a unique location in the index. Even if one of these techniques were modified to use clipping, which involves breaking up the data intervals into smaller intervals based on page boundaries [5], the number of tuple IDs in each page would typically be more than half the dataset, making the need for bit vector representation of tuple IDs such as that used in our approach absolutely necessary.

Of course there has been a great deal of work on bit vector indexing. A good summary of bit vector indices can be found in [6]. Bit vector indices have typically been used in OLAP systems like Sybase IQ [6] and model 204 [19]. The main contribution of our work over the existing literature in bit vector indexing is the particular scheme, including our predicates and projections, we use to achieve high performance searching over high-dimensional regions. The application of bit vector indexing to multimedia indexing problem is also novel.

6 Conclusions

This paper presents a method for quickly performing point queries against non-uniform high-dimensional regions. Such queries are motivated by multimedia retrieval and matching applications, such as audio fingerprinting. For highest accuracy, these applications can require per-data-point thresholds for accepting a match in a high-

dimensional space. Therefore, neither nearest neighbor search nor epsilon range searching are appropriate.

Rather, we map such queries into high-dimensional hyperrectangle searching with postprocessing. The proposed solution is based on bit vector indexing, and has complexity that is linear in the size of the dataset. Inserts and deletes are relatively cheap as long as the number of deletes is small compared to the number of inserts. Our techniques are applicable to both in memory and disk based applications.

Our solution was applied to an actual audio fingerprinting application and was compared to both linear scan and Hilbert bulk loaded R-Trees. Our scheme vastly outperformed the alternatives, and improved the overall performance of the search by a factor of 56 (compared to linear scan). The R-Tree was completely ineffective at indexing/partitioning the data, which was unsurprising given the properties of the data. These results were achieved through the use of redundancy. This redundancy resulted from partitioning the queries rather than the data, and using bit vector indexing to compress the results. Because of bit vector indexes' efficient representation of tuple IDs, the total size of our structures was smaller than the original dataset.

7 References

1. C. Burges, J. Platt, S. Jana: Extracting Noise-Robust Features from Audio. In Proc. IEEE Conference on Acoustics, Speech and Signal Processing, 2002.
2. Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, Uri Shaft: When Is "Nearest Neighbor" Meaningful?, ICDT 1999
3. Roger Weber, Hans-Jörg Schek, Stephen Blott: A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces., VLDB 1998
4. Antonin Guttman: R-Trees: A Dynamic Index Structure for Spatial Searching. SIGMOD Conference 1984
5. Volker Gaede, Oliver Günther: Multidimensional Access Methods. ACM Computing Surveys 30(2) 1998
6. Patrick E. O'Neil, Dallan Quass: Improved Query Performance with Variant Indexes. SIGMOD Conference 1997: 38-49
7. Joseph M. Hellerstein, Jeffrey F. Naughton, Avi Pfeffer: Generalized Search Trees for Database Systems. VLDB 1995
8. Ibrahim Kamel, Christos Faloutsos: Hilbert R-tree: An Improved R-tree using Fractals. VLDB 1994
9. Herbert Edelsbrunner, Hermann A. Maurer: On the Intersection of Orthogonal Objects. Information Processing Letters 13(3): 177-181 (1981)
10. Hans-Peter Kriegel, Marco Pötke, Thomas Seidl: Object-Relational Indexing for General Interval Relationships. SSTD 2001
11. Y. Rui, T. S. Huang, S.-F. Chang: Image retrieval: current techniques, promising directions and open issues, J. Visual Communication and Image Representation, Vol 10, no. 4, 1999
12. J. Bromley, I. Guyon, Y. LeCun, E. Sackinger, R. Shah: Signature Verification using a "Siamese" Time Delay Neural Network, Advances in Neural Information Processing Systems, vol 6, 1994
13. T. Funkhauser, P. Min, M. Kazhdan, J. Chen, A. Halderman, D. Dobkin, D. Jacobs: A Search Engine for 3D Models, ACM Transactions on Graphics, to appear 2003
14. Gisli R. Hjaltason, Hanan Samet: Ranking in Spatial Databases. SSD 1995
15. Stefan Berchtold, Daniel A. Keim, Hans-Peter Kriegel: The X-tree : An Index Structure for High-Dimensional Data. VLDB 1996
16. Jonathan Goldstein, Raghu Ramakrishnan: Contrast Plots and P-Sphere Trees: Space vs. Time in Nearest Neighbour Searches. VLDB 2000
17. Bernd-Uwe Pagel, Flip Korn, Christos Faloutsos: Deflating the Dimensionality Curse Using Multiple Fractal Dimensions. ICDE 2000
18. Norio Katayama, Shin'ichi Satoh: The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. SIGMOD Conference 1997
19. Patrick E. O'Neil: Model 204 Architecture and Performance. HPTS 1987
20. Stefan Berchtold, Daniel A. Keim, Hans-Peter Kriegel: The X-tree : An Index Structure for High-Dimensional Data. VLDB 1996: 28-39
21. C. Burges, J. Platt and S. Jana: Distortion Discriminant Analysis for Audio Fingerprinting, to appear in IEEE Transactions on Speech and Audio Processing, 2003.
22. D.M.J. Tax, R.P.W. Duin: Uniform Object Generation for Optimizing One-class Classifiers, Journal of Machine Learning Research, Vol 2, 155-173, 2002. Note: step 4 of their uniform generation algorithm has an error --- 1/d should be substituted for 2/d.