Hashing Technique: A New Index Method for High Dimensional Data

Zhexuan Song
Department of Computer Science
University of Maryland
College Park, Maryland 20742
zsong@cs.umd.edu

Nick Roussopoulos
Department of Computer Science &
Institute For Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
nick@cs.umd.edu

September 24, 1999

CS-TR-4059

Abstract

As showed in [17], when dimension goes high, sequential scan processing becomes more efficient than most index-based query. In this paper, we propose a new index method for high-dimensional data spaces. This method is based on hashing technique. The basic idea is: First find a hashing function which puts the given d-dimensional space data into a $\bar{d}$-dimensional buckets where $\bar{d} \ll d$. Then, we use existing index techniques to manage those buckets. We later define some properties of a *good* hashing function and give four hashing functions. To demonstrate the efficiency of our idea, we experimentally compared our algorithms with sequential scan and Pyramid-Techniques. The results demonstrate that this method outperforms others for skewed data set. It always beats the sequential scan by using only half of elapsed time for

range query. However if the data has uniform distribution, Pyramid-Technique is still the best method.

1 Introduction

More and more high dimensional data now appear in a variety of new database applications. For example, new database applications such as Cubetree [9] create high dimensional data when store and retrieve data warehousing views. In other area such as multimedia, content-based search [10] is an attractive new idea which is based on feature vectors. In order to support those applications, database system must be able to handle high dimensional data. The low-dimensional case (when dimensionality equals to 2 or 3) is well solved [3]. However, recent researches [11, 12] show that basically none of the querying and indexing techniques provide good results on both low-dimensional and high-dimensional data for large queries because of the so-called "curse of dimensionality".

The approach taken now is to find methods to lower the dimensionality of data and use low-dimensional index structure. For example, we have no method to handle 50-dimensional data, if we can transform those data into 2-dimensional data, we have a bunch of methods to store to retrieve them. However, as we do such transformation, some information of the original data set is lost. We have to sacrifice some query efficiency to compensate for the lost information. The cost we paid is either the inaccuracy of the result such as in SVD method, or many useless "false alarms" in the query result such as in Pyramid-technique.

In this paper, we developed a hashing based techniques. The basic idea is to find a hashing function which puts the high-dimensional data into low-dimensional buckets. When processing high-dimensional range queries, we first find the corresponding low-dimensional queries. Then we retrieve all the buckets which are inside the transformed query ranges. Finally, we check all the data in those buckets and discard the "false alarms". As showed in our experiment, like sequential scan, this method performs very well not matter how the data set is distributed. And it always outperforms sequential scan by using only half of the time. Some state-of-art methods work better when the data has uniform distribution. We believe that our method is the best choice on very skewed data set. Furthermore, our index structure does not need to be re-adjusted when the data is dynamically added.

Another advantage of our method is the fact that we can use any existing low-dimensional index structures to store the data items and take advantage of all the nice properties of those structures. The hashing based technique can easily be implemented on top of any existing DBMS.

The rest of the paper is organized as following: In section 2, we give a brief description of related work in high-dimensional indexing. Section 3 describes our algorithm and gives some discussion. Section 4 presents our experimental results. Section 5 gives the conclusions and directions for future research.

2 Related Word

Recently, many high-dimensional index structures have been proposed.

Berchtold, Keim, and Kriegel presented the X-tree [11] which is trying to adapting the algorithms of R*-trees to high-dimensional data. The basic idea of their algorithm is that they try to find an overlap-free split. If that split leads to an unbalanced split, then second, they omit that split and construct a supernode. The major drawback of the X-tree is that in order to guarantee the storage utilization, it must use the 50%-quantile when splitting a data page. As showed in [14], this is the worst case in high-dimensional indexing, because the resulting pages have an access probability close to 100%. The same thing will happen on R*-tree [4], KD-tree [2], SS-tree [12] and TV-tree [8].

The pyramid-technique [14] is a solution for the above drawback. It first splits the d-dimensional space into $2d$ parts. Then, the algorithm creates a mapping from the given space to a 1-dimensional space. Finally, a B+-tree is used to manage the transformed data. Once a range query arrives, it checks $2d$ parts separately and finds the result. The pyramid-technique outperforms the X-tree by a factor up to 2500 for range queries. The authors also extend the method for skewed data set. But in order to maintain the efficiency, the algorithm must find the median point before splitting the space. In a dynamic environment, the median is not always the same. Their solution is to re-build the whole index if the median point goes too far which is very expensive.

Another approach is called Singular Value Decomposition (SVD) method [6, 7]. The basic idea is to condenses most of the information in a data set to a few dimensions by applying SVD. The data in the few condensed dimensions are then indexed to support fast retrieval. The major drawback to this approach is that SVD is very expensive to compute, once new data are added into the dataset, the whole transformation must be re-computed. Therefore, it is not readily applicable to dynamic databases either. The other thing is that SVD method can not give the 100% correct result, because some information is lost due to the dimensionality reduction.

To overcome the first drawback, Kanth, Agrawal and Singh recently proposed an efficient way to incorporate the recomputed SVD-transform in the existing index structure [15]. However, their method is based on SVD, the second drawback can not be overcome.

3 The Hashing Technique

The basic idea of the Hashing Technique is to put the d-dimensional data points into $\bar{d}$-dimensional buckets by using a hashing function H where $\bar{d} \ll d$. Then use an efficient index structure such as B+-tree [1] (when $\bar{d} = 1$) and R*-tree [4] (when $\bar{d} > 1$) to store and access the buckets. Basically, any efficient low-dimensional index methods can be used. In this paper, we use R*-tree and B+-tree. Figure 1 gives the overview of the method:

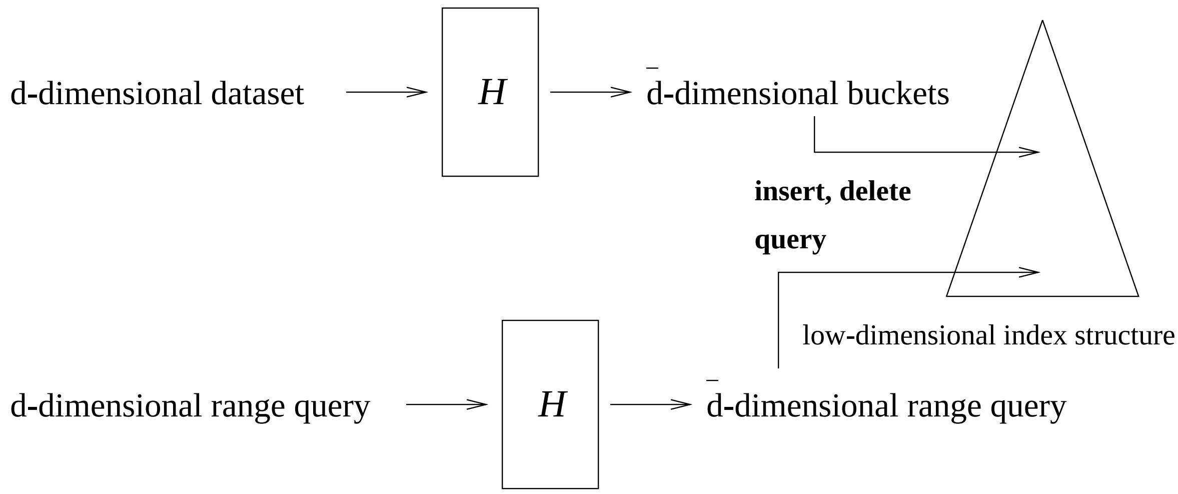

Figure 1: Overview of the Hashing Technique

In the leaf of the low-dimensional index structure, we store the d-dimensional information along with the $\bar{d}$-dimensional key so we do not need to provide an inversed transformation. Now the problem remained is how to find a good hashing function.

3.1 Hashing Functions

In order to find a good hashing function H, some criteria must be met.

First of all, in order to guarantee the correctness, if point p is in the range R, after transformation, $H(p)$ must be in transformed range $H(R)$.

Property 3.1 *To any point p and range R, p is in R, then $H(p)$ is in $H(R)$.*

We do not need the inversed side to be satisfied. Actually, if we want to lower the dimension, it is impossible to guarantee the inverse to be true.

The second property is set to guarantee the efficiency of the hashing function. If two point p_1 and p_2 which are close in d-dimensional space, $H(p_1)$ and $H(p_2)$ should be close in $\bar{d}$-dimensional space. Otherwise, suppose we have a range query R which contains p_1 and p_2, after transformation, in order to satisfy the first property, $H(R)$ must contain $H(p_1)$ and

$H(p_2)$. If $H(p_1)$ and $H(p_2)$ are very far away in $\bar{d}$-dimensional space, $H(R)$ now becomes a very large query range. This will decrease the efficiency of the method.

Property 3.2 *To any $\varepsilon > 0$, $\exists \delta$, to any p_1, p_2, if $|p_1 p_2| < \varepsilon$, then $|H(p_1)H(p_2)| < \delta$.*

If we can not find such a small δ for ε, that means we can not give an upper bound of size of the transformed query range. Of course, the lower δ is, the more efficient the function is. Here "lower" means the size of δ comparing to the whole low-dimensional data space.

We called the hashing functions which satisfy the above properties *good* hashing functions.

3.2 Distance mapping

First, we use distance function as our hashing function. The idea is that first find a point p in d-dimensional space. To any data point p', find the distance between p and p'. Define $H(p') = |pp'|$. Use $H(p')$ as a key and save the data into a B+-tree. Here the point p can be randomly selected. It is not necessary to pick a data point.

Once we have a query range R. The remaining problem is to find $H(R)$. Define D_{min} and D_{max} as the minimum and maximum distance of any point in the range R to p. Then $H(R) = [D_{min}, D_{max}]$. D_{min} and D_{max} can be easily found from the following lemma:

Lemma 3.3 *Given a hyper-rectangle $R = [x_{1_{low}}, x_{1_{high}}], \ldots, [x_{d_{low}}, x_{d_{high}}]$ and $p = \{y_1, \ldots, y_d\}$, $D_{min} = \sqrt{\sum_{i=1}^{d} D_i^2}$ where*

$$D_i = \begin{cases} 0 & : \quad x_{i_{low}} \leq y_i \leq x_{i_{high}} \\ x_{i_{low}} - y_i & : \quad y_i < x_{i_{low}} \\ y_i - x_{i_{high}} & : \quad x_{i_{high}} < y_i \end{cases}$$

and $D_{max} = \sqrt{\sum_{i=1}^{d} D_i^2}$ where $D_i = max\{|x_{i_{low}} - y_i|, |y_i - x_{i_{high}}|\}$

Proof: We will only prove the correctness of D_{min} because D_{max} is almost the same. In each dimension, D_i is the minimum possible value of distance of any points in R to p on dimension i, so D_{min} is the minimum value. The important thing here is that we can find such a $p' = \{y'_1, \ldots, y'_d\}$ in R which satisfies $H(p') = D_{min}$.

Define

$$y'_i = \begin{cases} y_i & : \quad x_{i_{low}} \leq y_i \leq x_{i_{high}} \\ x_{i_{low}} & : \quad y_i < x_{i_{low}} \\ x_{i_{high}} & : \quad x_{i_{high}} < y_i \end{cases}$$

It is very obvious that p' is in R. $\square$

Now, we can transform a d-dimensional hyper-rectangle query to a 1-dimensional range query. For each query range R, first we find D_{min} and D_{max}, then retrieve all the buckets in the range from B+-tree. Check the points in the buckets and filter the "false alarms", we have the final result.

Theorem 3.4 *Distance mapping is a good hashing function. Here we define $H(p') = |pp'|$.*

Proof: First of all, if p' is in R, from the lemma above we know that $D_{min} \leq |pp'| \leq D_{max}$, that means $H(p') \in [D_{min}, D_{max}]$. Secondly, to any $\varepsilon > 0$, define $\delta = \varepsilon$, to any p_1, p_2, if $|p_1p_2| < \varepsilon$, then according to triangular inequality, $|H(p_1) - H(p_2)| \leq \delta$. $\square$

The detail algorithm is listed in figure 2.

3.3 Multi-distance mapping

Since there are many great index methods for low-dimensional data, we further extend our method in order to increase the efficiency.

```
// p is the base point in d-dimensional space
Point_Set rangeQuery_1 (range R, Point p) {
    Point_Set res;
    find Dmin and Dmax using p; // using Lemma
    cs = btreeQuery (Dmin, Dmax);
    for (c = cs.first; cs.end; cs.next) {
        if (inside (R, c))
            res.add (c);
    }
}
```

Figure 2: Range query algorithm 1

This time we select a set of points $\{p_1, p_2, \ldots, p_{\bar{d}}\}$, for each point p in the d-dimensional space, find the distance from p to each point of the set. Now we have a $\bar{d}$-dimensional vector $< d_1, d_2, \ldots, d_{\bar{d}} >$. Obviously, $\bar{d}$ must be a small number, otherwise, we do not have any improvement here. Use that $\bar{d}$-dimensional vector as a key and save the point into a R*-tree. The insert, delete and update operations are quite easy.

For any range query R, we find $D_{i_{min}}$ and $D_{i_{max}}$ for each of $\bar{d}$ dimensions using the same method in section 3.2. Now we have a new range R' in our low-dimensional space. Run the query and filter the "false alarm", we have the result.

Theorem 3.5 *Multi-distance mapping is a good hashing function. Here we define* $H(p') =< |p_1p|, |p_2p|, \ldots, |p_{\bar{d}}p| >$.

Proof: First of all, if p is in R, we know that $D_{i_{min}} \leq |pp_i| \leq D_{i_{max}}$, that means $H(p)_i \in [D_{i_{min}}, D_{i_{max}}]$ for each $i \in [1, \ldots, \bar{d}]$. So $H(p)$ is in $H(R)$. Secondly, to any $\varepsilon > 0$, define $\delta = \sqrt[\bar{d}]{\bar{d}}\varepsilon$, to any p_1, p_2, if $|p_1p_2| < \varepsilon$, then according to triangular inequality,

$|H(p_1)_i - H(p_2)_i| \leq \varepsilon$ for each $i \in [1, \ldots, \bar{d}]$. So $|H(p_1)H(p_2)| \leq \delta$. $\square$

The detail algorithm is listed in figure 3.

```
// d_bar is the number of points we selected
// p is the array of base point in d-dimensional space
Point_Set rangeQuery_2 (range R, int d_bar, Point p[])
{
    Point_Set res;
    // transform the range query
    for (i = 0; i < d_bar; i++)
        find Dmin[i] and Dmax[i] using p[i];
    cs = rtreeQuery (Dmin, Dmax, d_bar);
    for (c = cs.first; cs.end; cs.next) {
        if (inside (R, c))
            res.add (c);
    }
}
```

Figure 3: Range query algorithm 2

3.4 Max-dimension and Multi max-dimension functions

The distance function is not very efficient. Look at the following example in a 2-dimensional space (Figure 4).

Here p is a point we selected as the base point. For range R, D_{min} and D_{max} are showed as the minimum and maximum distance of points in R to p. All the points between D_{min} and D_{max} will be retrieved. Since R is a rectangle, D_{min} and D_{max} are two circles. Wherever R is, there is always some waste space.

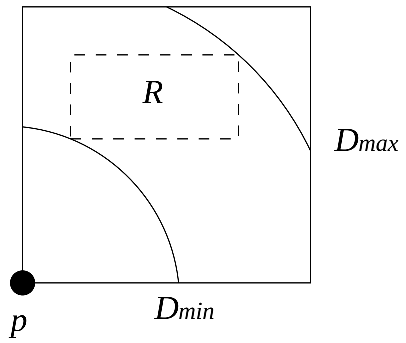

Figure 4: Example of a distance mapping

In order to solve the problem, we define the following function.

Definition 3.6 *Suppose* $p_1 = \{x_1, x_2, \ldots, x_d\}$ *and* $p_2 = \{y_1, y_2, \ldots, y_d\}$ *are two d-dimensional points. Define:* $Dist(p_1, p_2) = max\{|x_i - y_i|\}$*, where* $i \in [1, \ldots, d]$.

The meaning of function $Dist(p, p')$ can be view as following: Generate a series of hypercubes and p is their center. There must exist a hypercube C which passes p'. The distance between p and any side of C equals to $Dist(p, p')$. Here $Dist(p, p') = Dist(p', p)$.

Select a point p in a d-dimensional space, for each data point p' in the space, find $Dist(p, p')$ and use it as a key to save p' into a B+-tree.

Like the distance mapping, for any range R, we need to find D_{min} and D_{max}. The method is almost the same:

Lemma 3.7 *Given a hyper-rectangle* $R = [x_{1_{low}}, x_{1_{high}}], \ldots, [x_{d_{low}}, x_{d_{high}}]$ *and* $p = \{y_1, \ldots, y_d\}$*,* $D_{min} = max\{D_i\}$ *where*

$$D_i = \begin{cases} 0 & : \quad x_{i_{low}} \le y_i \le x_{i_{high}} \\ x_{i_{low}} - y_i & : \quad y_i < x_{i_{low}} \\ y_i - x_{i_{high}} & : \quad x_{i_{high}} < y_i \end{cases}$$

and $D_{max} = max\{D_i\}$ *where* $D_i = max\{|x_{i_{low}} - y_i|, |y_i - x_{i_{high}}|\}$

We omit the proof here because it is almost the same as the previous one. Next we want to show that $Dist$ is a good hashing function.

Theorem 3.8 *Function Dist is a good hashing function. Here* $H(p') = Dist(p, p')$.

Proof: First of all, if p' is in R, we know that $D_{min} \leq Dist(p, p') \leq D_{max}$ by the definition of $Dist$. So $H(p)$ is in $H(R)$.

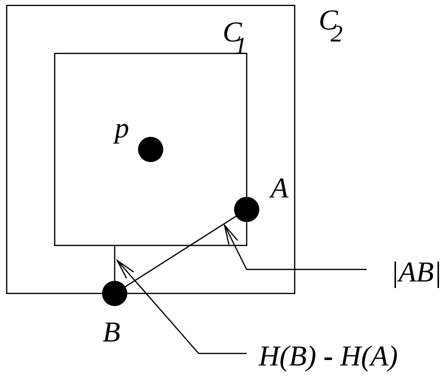

Figure 5: Max-dimension mapping is a *good* hashing function

Secondly, to any $\varepsilon > 0$, define $\delta = \varepsilon$, to any A, B, if $|AB| < \varepsilon$, then we want to show that $|H(A) - H(B)| < \delta$. Look at Figure 5, p is the selected point and A and B are two random data points. There are two hypercubes C_1 and C_2 centered with p which pass A and B. The distance between two cubes are $|H(B) - H(A)|$ which the nearest possible distance between any point on C_1 and C_2. That means $|H(B) - H(A)| \leq |AB| < \varepsilon = \delta$. $\square$

The query algorithm is the same as distance mapping. So we do not list the detail here.

Like distance mapping, we can also select a set of points instead of one. The basic idea is almost the same. And it is not hard to prove that it is a good hashing function too.

3.5 Discussion

We present four hashing functions in the previous parts. The remain two problems is about the location of the base points and the number of dimensions after transformation.

Definition 3.9 *The degree of a hashing function is the dimensionality of the transformed data.*

The degree selection is very important in the hashing method. In most cases, as the degree increases, the query becomes more efficient. That is, the data in the buckets which are in the transformed query region are more likely to be the data we need. However, at the same time, we need to use more complicated index structure which creates more overhead.

According to [17, 14], when the degree is bigger than 10, most index methods become inefficient. So to any hashing functions, the degree should be no more than 10. According to our experiment, if the selectivity of the query is very low, degree can be big, otherwise, degree should be no more than 3. The details can be found in the next section.

The other thing is about the selection of the base points. The object of base points selection is to decrease the "false alarm" to any queries. Theoretically, any positions are acceptable, even outside the data space. However as the degree is bigger than 1, things are a little different.

For example, we use multi max-dimensional function as our hashing function and choose degree to be 2. First of all, we need to select two base points. If those two points (A and B) are very close, to any data point p in the space, $Dist(A, p)$ and $Dist(B, p)$ are very close. Now we can say these two values have some relations. The object of selecting more than one base point is to include more information. However, in the above example, we gain very little benefit by introduction the second base point.

This example shows us that when we select more than one base point, those points should be far away enough to generate different information for any data points. If all the data points are in a hypercube. Corner points can be a very good selection.

4 Experimental evaluation

To access the merit of our algorithm, we implemented the algorithm and performed some experimental evaluation of Hashing Technique. The following competitive techniques are used:

- Pyramid-Technique
- Sequential Scan.

The Pyramid-Technique has been chosen for comparison, because it is the best high dimensional techniques ever designed. According to [14], Pyramid-Technique outperforms the X-tree by a factor of up to 2500 (total elapsed time) for range queries, which is an amazing result.

Sequential scan processing is inefficient in low-dimensional data space. However, according to recent research [17], it yields better performance in high-dimensional data spaces than most index-based query processing. Therefore, we included the sequential scan in our experiment.

In Pyramid-technique and our method, the index key is transformed data. We stored all the relevant information in the leaf node. Therefore, no additional object fetchings are needed. This made the comparison more fair.

The experiments are run on a Sun Ultra 1 machine with 128 M memory.

In the first part, we compared the performance of four different hashing functions. Then in the second part, we used the best hashing function to perform the comparison with the other two techniques. In both part, range queries are used since range query is a basic operation for other queries. The queries are hypercubes selected randomly from the data space. But each hypercube has same volume. This volume can be viewed as the selectivity of a query if the data points are uniformly distributed.

4.1 Evaluation of different hashing functions

In the last section, we introduced four hashing functions, in this part, we compared the performance of those functions. The data set we used contains 1,000,000 uniformly distributed points in a 40-dimensional data space.

We chose distance mapping, degree 2 multi-distance mapping, max-dimension mapping, degree 2 and 3 multi max-dimension mapping functions. Since the data points are uniformly distributed, the selectivity is the volume of the query hypercube to the total volume of the data space. The result is showed in Figure 6.

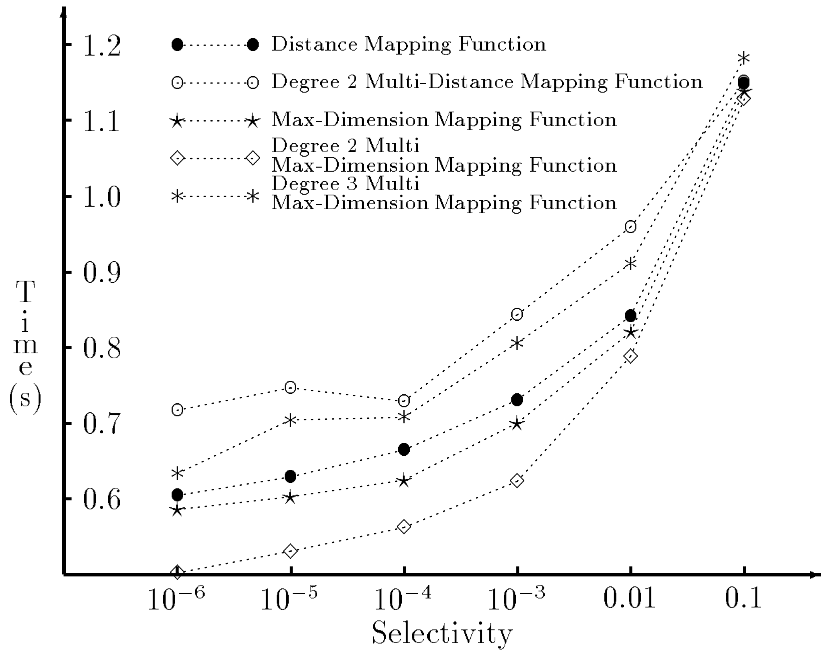

Figure 6: Performance Behavior over Different Hashing Functions

In our first experiment, we measured the performance of five functions on a given set of data points. The selectivity varied from 10^{-5} to 0.1. The page size is 4096 bytes, which leads to an effective page capacity of 25 objects per page. Three methods use R*-tree whose minimum number of entries in a page is 10 objects per page. All multi degree functions use corner points as base points.

We observed that in these five results, Degree 2 multi max-dimension mapping is the most efficient. Especially when the selectivity is low. And degree 2 multi-distance mapping is the worst. The max-dimension mapping is always better than distance mapping. As the selectivity becomes close to 1, all five functions seems to have no big difference.

Now look at three max-dimension mapping functions. When the degree increases from one to two. The decrease of the "false alarms" makes the method more efficient. However at the same time, we must spend more overhead on the two-dimensional index structure. Here the gain is more than loss, so the total elapsed time becomes low. As the degree increases again from two to three, the gain is no longer more than loss. Then the total time increases. We can claim that the multi max-dimension method will become even more inefficient as the degree is greater than 3.

4.2 Evaluation of different techniques

In this part, we want to compare our method with sequential scan and Pyramid-Technique. In the last part, we found that degree 2 max-dimension mapping is the best hashing function. So we use that function in this part along with Pyramid Technique and Sequential Scan. We used two data sets. Both contains 500,000 50-dimensional data points. In the first set, all the data are uniformly distributed and in the second set, they are very skewed.

In the first experiment (Figure 7) we find that Pyramid-Technique is the best method in uniformly distributed data set. Our method is better than Sequential Scan (use about 1/2 total elapsed time) but worse than Pyramid-Technique, especially when the selectivity

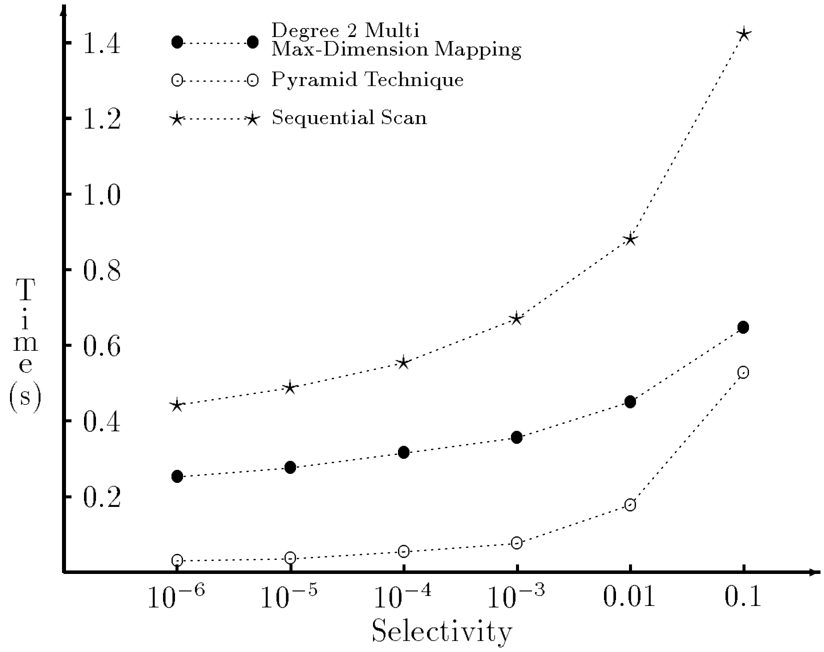

Figure 7: Performace of three techniques on uniformly distributed data

is low.

In the second experiment, we use a skewed data set. It is a d-dimensional data but only k dimensions are independent. The rest $d - k$ dimensions have some relation with those k dimensions. This time, we varied k and set $d = 50$. The number of data points is 500,000 and the volume of query range is 0.01% of the total volume. The result is displayed in Figure 8.

As Figure 8 shows, our method outperforms Pyramid-Technique when k is low. However, as k increases, i.e. the data become less skewed, Pyramid-Technique shows its power. Sequential scan couldn't compete with our method for any of these queries.

Summarize the results of our experiments, we make the following observations:

1. For all our hashing functions, degree 2 multi max-dimension mapping function is the best one.

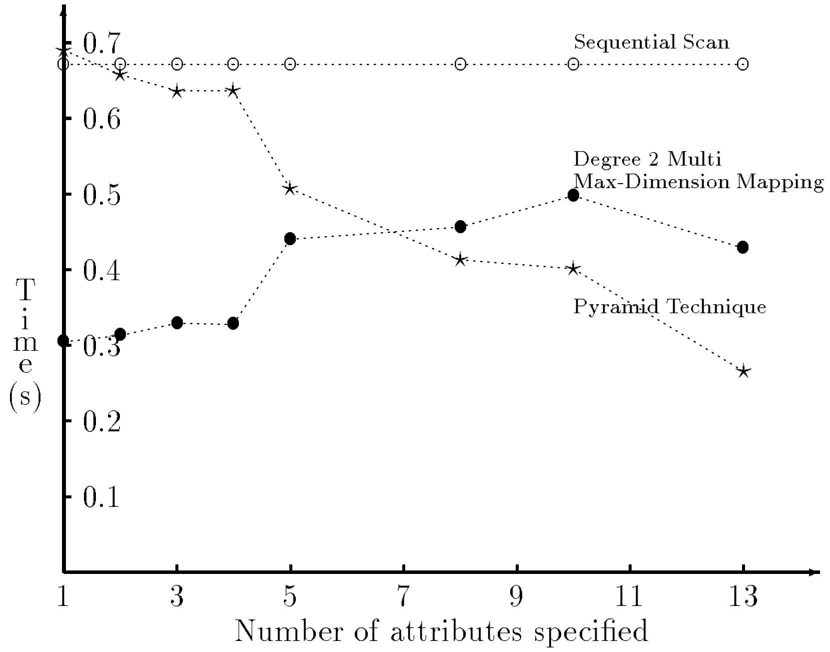

Figure 8: Performace of three techniques on uniformly distributed data

2. For uniformly distributed data set, Pyramid-Technique is the best method. However if the data set is very skewed, both sequential scan and our method is faster. Our method can beat sequential scan in all cases.

5 Conclusions

In this paper, we proposed a new indexing method, the Hashing-based index method. The basic idea of this method is to select a hashing function. Then put the high dimensional data into low dimensional buckets, and use the existing method to store and retrieve those buckets. The most important thing in this method is the selection of the hashing function. We gave some properties of a good hashing function and presented 4 different functions.

The concepts of Hashing Technique are designed for hypercube range query in a high-dimensional space. For skewed data set, our method performs better than other index structures. However, if the data has uniform distribution, Pyramid-Technique is still the best choice.

We plan to find more hashing functions in our future work.

References

[1] Comer D. *The Ubiquitous B-tree* ACM Computing Surveys, 1979.

[2] Robinson J. T. *The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes* Proc. ACM SIGMOD, 1981.

[3] H. Edelsbrunner. *Algorithms in Combinatorial Geometry* Springer-Verlag, 1987.

[4] Bechmann N., Kriegel H.P., Schneider R., Seeger B.. *The R*-tree: An Efficient and Robust Access Method for Points and Rectangles* Proc. ACM SIGMOD, 1990.

[5] Tzi_cker Chiueh. *Content-Based Image Indexing* Proc. of VLDB , 1994.

[6] Hull, D. *Improving text retrieval for the routing problem using latent semantic indexing* Proc. ACM SIGMOD, 1994.

[7] R. Ng, A. Sedighian. *Evaluating multi-dimensional indexing structures for images transformed by principle component analysis* Proc. ACM SPIE, 1994.

[8] Lin K, Jagadish H. V., Faloutsos C. *The TV-Tree: An Index Structure for High-Dimensional Data* VLDB, 1995

[9] Nick Roussopoulos, Yannis Kotidis, Mema Roussopoulos. *Cubetree: Organization of and Bulk Incremental Updates on the Data Cube*

[10] Philip Korn, Nikolaos Sidiropoulos, Christos Faloutsos, Eliot Siegel, Zenon Protopapas *Fast Nearest-Neighbor Search in Medical image Databases* Conf. on VLDB, 1996

[11] Stefan Berchtold, Daniel A. Keim, Hans-peter Kriegel. *The X-tree: An Index Structure for High-Dimensional Data* Proc. of VLDB, 1996.

[12] David White, Ramesh Jain. *Similarity Indexing with the SS-tree* Proc. of ICDE, 1996.

[13] Sergey Brin. *Nearest Neighbor Search in Large Metric Spaces* Proc. of VLDB, 1997.

[14] Stefan Berchtold, Christian Bhon, Hans-Peter Kriegel. *The Pyramid-Technique: Toward Breaking the Curse of Dimensionality* Proc. of the ACM SIGMOD, 1998.

[15] K. V. Rave Kanth, Divyakant Agrawal, Ambuj Singh. *Dimensionality Reduction for Similarity Search in Dynamic Database* Proc. of ACM SIGMOD, 1998.

[16] Stefan Berchold, Bernhard Ertl, Kaniel Keim, Hans-Peter Kriegel, Thomas Seidl. *Fast Nearest Neighbor Search in High Dimensional Space* Proc. of ACM SIGMOD, 1998.

[17] Beyer K., Goldstein J., Ramakrishnan R., Shaft U. *When is "Nearest neighbor" Meaningful?* submitted for publication, 1998.