

# Using Hilbert Curve in Image Storing and Retrieving\*

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

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

## ABSTRACT

In this paper, we present a method to accelerate the speed of retrieving subset of uncompressed images in a database without using extra disk space. First we change the storing method: pixels of an image are saved in Hilbert order instead of Row-wise order in traditional method. After studying the property of Hilbert curve, we give a new algorithm which greatly reduces the data segment numbers on the disk. Although we have to retrieve more data than necessary, because the speed of sequential readings is much faster than the speed of random readings, our method spends about 10% less elapsed time which is showed in our simulation experiments. In some systems, the saving can be as high as 90%.

## 1. INTRODUCTION

Handling images in a database is one of the requirements for the current database management systems (DBMSs). Images arise in many applications, including: scientific databases, such as the satellite pictures in ESIP project [12], computer vision [1], etc.

Many works [2, 4, 5, 7, 9] have been done to find a part of an image that is similar to the intended target. After we find the range, the remaining problem is: how to retrieve all the pixels inside the range efficiently? The problem is sometimes called subset query problem: given an image  $I$  and a range  $R$ , retrieve all the pixels of  $I$  which are in  $R$ . In this paper, to simplify the problem, we only consider the range as a rectangle.

Basically, there are two methods to store images: in compressed format or in original format. Compressed format is widely used when the image size is not very big. In a subset query, the whole compressed data is retrieved from the disk and decompressed in main memory before the program extracts the subset. However as image size goes big, (for example, in ESIP project, each image has 7 bands and

each band has size up to 60 M bytes), it is impossible to decompress the whole image in main memory. Thus, in this situation, storing images in original format seems to be the only choice.

Traditionally, in original format, pixels of images are stored on disk row-wisely: start from the top-left pixel, from left to right, from up to down, saved continuously. Several bytes are used to save the information of each pixel (Most of the time, those bytes are color information), which we called *pixel size*. We can imagine that the pixels stored on the disk form a pixel string. Once we have a subset range, the range cuts the pixel string into small pieces. In order to get the subset query result, We have to figure out the start position of those pieces and retrieve them one by one.

The idea of our algorithm is based on the following fact: normally sequential readings are much faster than random access readings. In some other experiments, we found that sequential readings can be as fast as 15 M/sec when random access reading is less than 1 M/sec. So, if an algorithm can decrease the piece number of pixel strings for *any* query ranges, the retrieval efficiency will be improved.

In this paper, we use Hilbert order instead of row-wise order to save an image and design a new subset query algorithm. As later showed in our experiments, *with the same disk space*, the number of pixel string pieces can be at least 50% less. The total subset query time is about 10% less than traditional method. In some systems, the saving can be as high as 90%.

## 2. HILBERT CURVE

Hilbert curve is a continuous curve which passes through each point in the space exactly once. So it enables one to continuously map an image onto a line and is an excellent 2-d-image-to-line mapping. Each point in the image has a position on the line which is called the Hilbert order of that point. More detail about the Hilbert curve can be found in [8, 3, 6].

We choose Hilbert order because in this way, pixels are grouped locally. Using the string model, to any rectangle range, the pixels inside the range are more likely to form some long string segments instead of short segments in traditional method, and total segment number may be small.

However, it is not so efficient as we thought. Look at Figure 1.

Since the pixels are saved in Hilbert order, the Hilbert curve can be viewed as the pixel string. The dot rectangle is the query range. We find that the range cuts the curve into several segments. Some segments are quite long but

\*Paper presented at the MIR 2000 in Los Angeles

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM Multimedia Workshop Marina Del Rey CA USA  
Copyright ACM 2000 1-58113-311-1/00/11...\$5.00

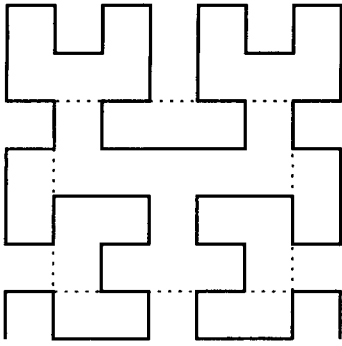


Figure 1: Range query on Hilbert curve

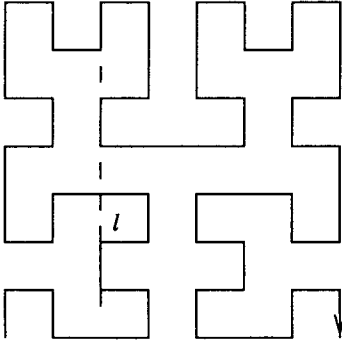


Figure 2: Hilbert curve with a line

many of them along the border are very short. The total segment number does not decrease, which is showed later in our experiment. However, this can be solved by our new algorithm.

### 3. ALGORITHMS AND DISCUSSION

We first do some research on Hilbert curve. In Figure 2, suppose  $l$  is the left border of a query range. Hilbert curve passes every pixel of the image. For those pixels on the line, the Hilbert curve can do the following three things:

1. From the nearest pixel on the right of the line or on the line (upper or down), traverses the pixel and goes up (or down) or right.
2. From the nearest pixel on the left of the line, traverses the pixel and goes up (or down) or right.
3. From the nearest pixel on the line or on the right, traverses the pixel and goes left.

In case 1, the curve still stays inside the query range. (Here we assume that the border is part of the range.) In the rest two cases, the curve enters or leaves the range. We call that traverse a *cross* on the border and the pixel a *cross point*.

**THEOREM 1.** *Each pixel on a border has 50% possibility to become a cross point.*

Due to the space limit, we do not give the proof here, the detail proof can be found in [10].

Hilbert curve is a continuous curve. The end points of those curve segments inside the range can only appear on the range border and must be cross points. So the segments inside the range must be the half of the number of cross points.

**COROLLARY 2.** *To any rectangle ranges with parameter length  $c$ , the average number of cross points on the border is  $c/2$ , and the average number of Hilbert Curve segment inside the range is  $c/4$ .*

Our algorithm is based on the following observation: those cross points are not uniformly distributed. Suppose the left border of a query range  $l$  can be defined as:  $x = x_0, y \in [y_l, y_h]$ . Cross points are more likely to appear if  $x_0$  is an odd number. If  $x_0$  is an even number, the number of cross points on that line segment sharply decreases. Furthermore, if  $x_0$  is exactly divisible by 4, 8, etc., the number of cross points can be even less. The algorithm is listed in Figure 3.

```

point-set rangeQuery (lowx, lowy, highx, highy, n) {
  if (lowx is not divisible by 2^n);
    lowx = max (x | x <= lowx and x mod 2^n = 0);
  if (highx is not divisible by 2^n)
    highx = min (x | x >= highx and x mod 2^n = 0);
  /* same for lowy, highy */
  normalRangeQuery (lowx, highx, lowy, highy);
  filter the useless pixels;
}

```

Figure 3: Fast subset query algorithm

Now the problem left is how big the  $n$  should be in order to get the best performance. Suppose  $t_1$  is the average time of one search on disks in a system,  $t_2$  is the average time of one reading. There is a range query with parameter length  $c$ . When  $n = 0$ , there is no useless pixels but  $c/4$  curve segments inside the query range. When  $n = 1$ , we have only  $c/8$  segments, which saves  $(c/8)t_1$ . However at the same time, we have about  $c/2$  useless pixels in our augmented range. This costs us  $(c/2)st_2$ , where  $s$  is the *pixel size*. Now the benefit we gained by increasing query range is:  $B_1 = (c/8)t_1 - (c/2)st_2$ . When  $n$  increases by 1, the number of the curve segments in the range decreases by half and the number of useless pixels is about three times more. As  $n = k$ , the benefit is:  $B_i = (c/2^{n+2})t_1 - 2c3^{n-2}st_2$ . Define  $n^*$  to be the biggest integer which makes  $B_{n^*} > 0$ . At that point, the system has the best performance.

Observe the above formula, we find that the selections of  $n^*$  differ in various systems. It depends on how fast a search on disks can be comparing to a reading. It is obvious that if  $t_1$  is big and/or  $t_2$  is small, i.e. a sequential reading is very fast comparing to a random access reading,  $n^*$  will be big.

Another thing that affects the selection of  $n^*$  is the *pixel size*. If *pixel size* is big, every useless pixel we included in the query range costs more time to retrieve, which makes us unable to afford to increase the range too much. This makes  $n^*$  to be a small number.

However, the selection of  $n^*$  does not depend on the size of the query range. This is a little different from our original guess, but the experiments prove it. It means in any system, once  $n^*$  is obtained, it will be the same for all the queries.

## 4. EXPERIMENTAL RESULTS

To access the merit of our algorithm, we implement it in C++. First, we apply our algorithm on a set of synthetic data. The queries on the experiments are randomly selected. Then we use the algorithm in our real-life project.

The first part is to evaluate our algorithm by using synthetic data set. The experiments are run on a Sun Ultra 1 machine with 128 M memory. We compared the performance of our algorithm in different parameters along with the traditional row-wise method. The CPU time is negligible, we base our comparison on (a) the segment number and (b) total data retrieving time.

The data set consists of images with  $1024 \times 1024$  pixels. The total size of each image is *pixel size*  $\times$  1 M bytes on the disk. Those pixels are saved continuously on the disk according to their Hilbert order. The page size is 4 K.

We select row-wise (traditional) method,  $n = 0$  (normal query algorithm),  $n = 1, 2, 3$  and 4. We compare the segment numbers inside the range. More than 10,000 possible positions are selected and the average results are showed in Figure 4.

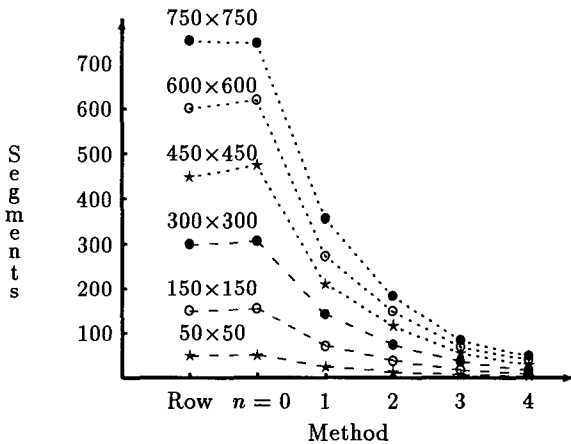


Figure 4: Number of data segment

In Figure 4, the number on the line (e.g.  $750 \times 750$ ) is the size of query range. We find that comparing to traditional method, normal query algorithm has almost the same segment numbers. The reason is discussed in section 2. In our new algorithm, as  $n$  increases by 1, the segment numbers almost decrease by half. When the query sizes are big, the number can be even less than half. The second observation is that when  $n = 0$ , segment number in the range is almost a quarter of the total parameter length of the query range. Both observations fit our theoretical calculation well.

Next, we want to compare the running time of different methods. We checked different range size along with different *pixel size*. Results are in Figure 5, 6 and 7.

Look at those figures, we found that to any *pixel size*, as  $n$  increases, the total elapsed time decreases at first which is due to the decrease of the segment number, then the time increases again. The optimized point is at  $n^*$ . We compare the time in  $n^*$  with that in row-wise method, the saving is about 5% - 20%. The average is about 10%.

Another observation is about  $n^*$ . When *pixel size* = 12,  $n^*$  appears in 2, sometimes 3. As *pixel size* = 8,  $n^*$  is 3, and  $n^*$  becomes 3 or 4 when *pixel size* = 4. If the *pixel size* is big enough, for example as big as one disk page,  $n^*$  will

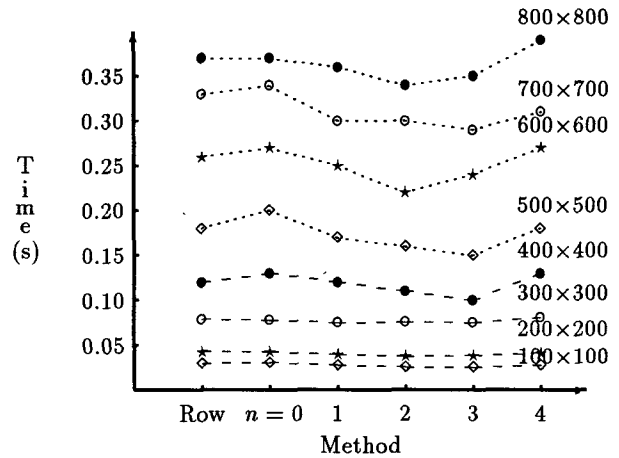


Figure 5: Average reading time (sec) when *pixel size* = 4

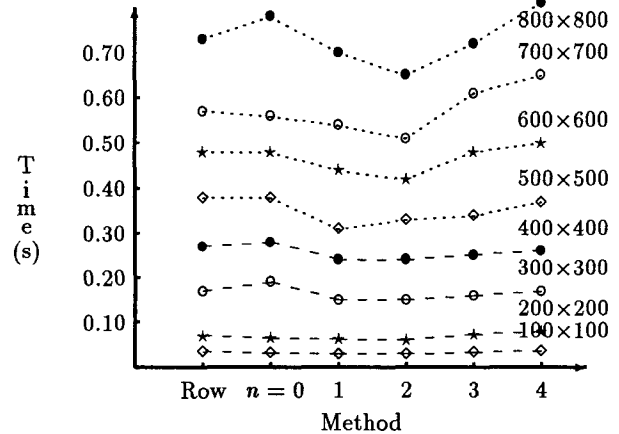


Figure 6: Average reading time (sec) when *pixel size* = 8

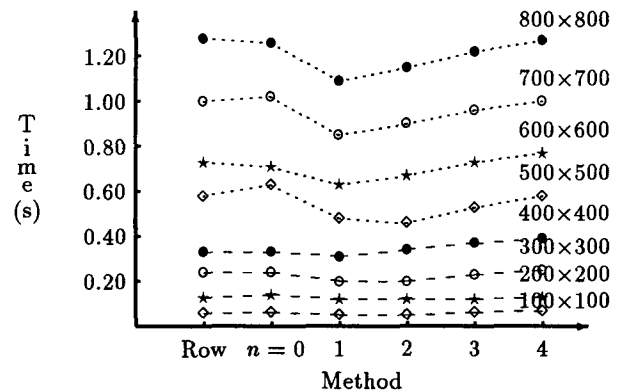


Figure 7: Average reading time (sec) when *pixel size* = 12

be 0, *i.e.* the normal data retrieval algorithm will have the best performance.

Then, we use our method in ESIP project. The project is supported by NASA. It provides online services for users to check and download satellite images for earth science. The size of the total data reaches as high as several Tera bytes. One important requirement is subset query. Users can check the previews of each image and mark a subarea on any previews, and the system will find the image, retrieve all the pixels in that area, create a file and give that file back to users.

This application uses IBM HPSS system to store images. HPSS is very efficient for large file retrieving. The 8-way parallel ftp can be as fast as 45 M/sec. But the page size is very big. (In our system, the page size is 1 M.) Meanwhile, the time used for one disk search is very long (comparing with the data retrieving time). That means, reading 1 M bytes data at once may be faster than ten 1 byte readings. As mentioned in section 3, in this case, our method is very attractive.

The program is run on one node of an IBM SP2 machine. The queries are running on real satellite pictures in the database which is stored in an IBM HPSS system. When users select a rectangle range from a preview picture, the system will find the image and run the subset query. Each image used in the experiment has 7 bands, and the size of each band is 58.8 M bytes. The *pixel size* is 1 byte per pixel. The memory buffer for each query is 2 M. The result is showed in Figure 8.

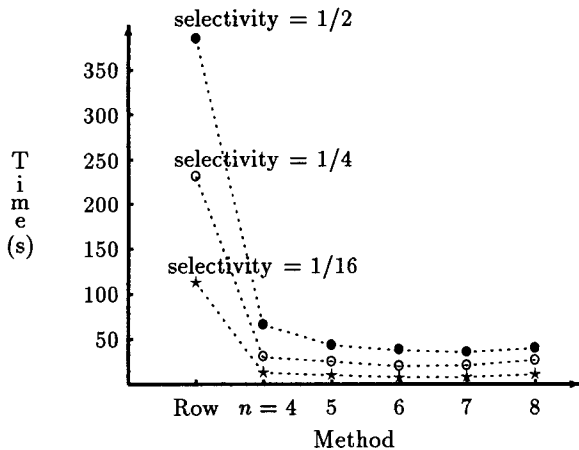


Figure 8: Average transaction time in different methods

The time in the graph is the total transaction time which includes hpss login/logout time, the query time and the time of writing the result into a destination file. The selectivity is the ratio of the query area to the total image size.

We find from the experiment result that in this system configuration,  $n^*$  is between 6 and 7. So we do not display the value when  $n = 1, 2$ , and 3. The result shows that using either  $n = 6$  or  $n = 7$ , the total saving is about 90%.

## 5. CONCLUSION

The goal of our algorithm is to generate more sequential readings in a retrieval process. We first save the image pixels in Hilbert order then exploit the clustering properties of the Hilbert curve and propose to increase the query range a

little bit before retrieving data from the disk. We performed experiments to test how big the range should be in order to get the best performance. The major conclusion is that the optimal value decreases when the *pixel size* increases.

Future research could check the performance in a parallel environment, and use the same technique on high-dimensional data.

## 6. ACKNOWLEDGEMENTS

We would like to thank Doug Moore, for his fast Hilbert curve function lib online [11].

## 7. REFERENCES

- [1] D. Ballard and C. Brown. *Computer Vision*. Prentice Hall, 1982.
- [2] S. Berchtold, C. Bohm, B. Braunmuller, D. Kein, and H. Kriegal. Fast parallel similarity search in multimedia database. In *Proc. of SIGMOD*. Tucson, Arizona USA, 1997.
- [3] T. Bially. Space-filling curves: Their generation and their application to bandwidth reduction. *IEEE Trans. on Information Theory*, IT-15(6):658–664, November 1969.
- [4] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Perkovic, and W. Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information System: Integrating Artificial Intelligence and Database Technologies*, (3-4):231–262, 1994.
- [5] T. Gevers and A. Smuelders. An approach to image retrieval for image databases. In *Proc. of Database and Expert Systems Application*. Prague, Czechoslovakia, 1993.
- [6] J. Griffiths. An algorithm for displaying a class of space-filling curves. *Software-Practice and Experience*, (5):403–411, 1986.
- [7] K. Hirata and T. Kato. Query by visual example — content based image retrieval. In *Proc. of Advances in Database Thchnology*. Vienna, Austria, 1992.
- [8] I. Kamel and C. Faloutsos. Hilbert r-tree fractals. In *Proc. of VLDB Conference*. Santiago, Chile, September 1994.
- [9] J. Liang and C. Chang. Similarly retrieval on pictorial databases based upon module operation. In *Proc. of Database Systems for Advanced Application*. Taejon, South Korea, 1993.
- [10] Z. Song and N. Roussopoulos. A New Method to Store and Retrieve Images. Technical Report CS-TR-3991. University of Maryland, 1999.
- [11] <http://www.caam.rice.edu/dougm/twiddle/Hilbert/>, 1999.
- [12] <http://www.umiacs.umd.edu/research/esip/>, 1999.