A New Method to Store and Retrieve Images

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

February 11, 1999

CS-TR-3991

Abstract

In this paper, we present a method to accelerate the speed of querying and retrieving images in database. First we change the storing method: pixels of an image are saved in Hilbert order instead of Row-wise order using in traditional method. Then after studying the property of Hilbert curve, we give a new algorithm which greatly reduce the data segment number on the disk. Although we have to retrieve more data than necessary, because the speed of sequential reading is much faster than random reading, we have about 10% improvement on the total query time which is showed in our simulation experiments.

1 Introduction

Handling images in a database in one of the re requirements for the current database management systems (DBMSs). Images arise in many applications, including: scientific databases, such as the satellite pictures in GLCF project [9], computer vision [2], etc.

Many works [8, 5, 4, 6] have been done about how to find the part of a image in a database that is similar to a the intended target. Once we find an interesting range, the remaining problem is: How to retrieve all the pixels inside the range efficiently? To simplify the problem, we consider the range as a rectangle in this paper.

Normally pixels of images are stored on disk row-wisely: from left to the right, for up to down, saved continuously. Several bytes are used to save the information of one pixel (color information most of the time, maybe more in some situation), which are defined as *pixel size*. Those pixels on the disk can be viewed as a string. Once we have the range, it cuts the string into many small pieces. We have to figure out the position of each piece and retrieve them one by one.

As we knew, sequential readings are much faster than random access readings. In some other experiments, we found that a sequential reading can be as fast as 15 M/sec when random access reading is less than 1 M/sec. If an algorithm can decreases the segment number for *any* query, the data retrieval speed will be improved. Our objective in this paper is to find a new way to store images and such an algorithm *without spending more disk space for each image.*

In this paper, we use Hilbert Order instead of row-wise order to save an image and design a new algorithm. As later shown in our experiment, with the same disk space, the segment number can be at least 50% less. The final performance is about 10% better than traditional method. The paper is organized as follows. Section 2 gives a brief description of the Hilbert Curve and its property. Section 3 describes our algorithm and gives some discussions. Section 4 presents our experimental results. Section 5 gives the conclusions and directions for future research.

2 Survey

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 to the line which is called the Hilbert Order of that point.

Given an image, the Hilbert Order of each pixel can be obtained by the following way: first, create a virtual grid that contains all the pixels which forms a rectangle starting at $(0,0)$ and ending at point (n,n). Next recursively generate the Hilbert Curve that covers the whole grid. Hilbert Curve is a self-similar curve which means it can be generated recursively. The basic curve is on a 2×2 grid with order 1. To derive a curve of order i, each vertex of the curve of order $i-1$ is replaced by the basic curve, which may be rotated or reflected. Figure 1 shows the curves with order 1, 2 and 3.

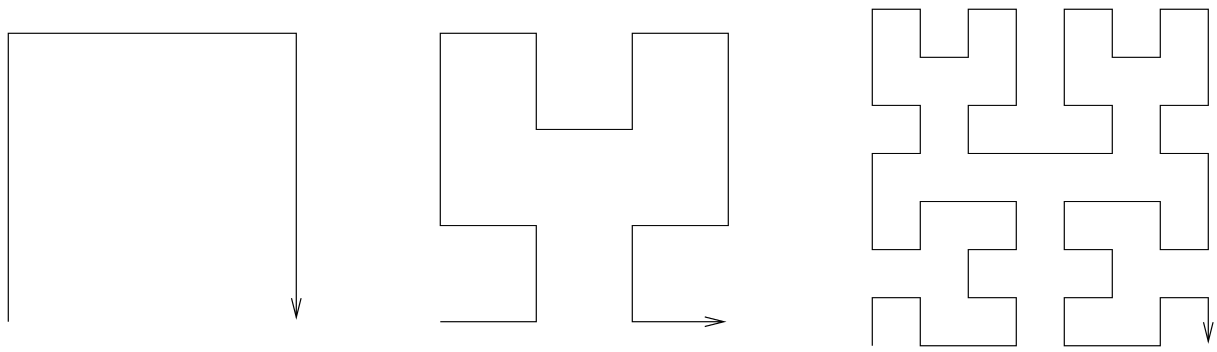

Figure 1: Hilbert Curve with order 1, 2 and 3

Now each pixel in the image stays in some place of the curve. Finally, follow along the curve, the Hilbert Order of each pixel can be uniquely decided. More detail about the Hilbert Curve can be found in [7, 1, 3].

There is a good property about the Hilbert Curve that can be easily seen from the recursive generation of the curve:

Property 2.1 (locality) *Suppose H_i, H_j are two Hilbert Curve with order i, j, and $i > j$. The path of H_i follows the path of H_j.*

For example, let $j = 1$, $H_i(i > 1)$ must fill the lower-left part of the space, then upper-left, upper-right, finally lower-right part which is the order of H_1.

We choose this order because in this order, pixels are grouped locally. To any rectangle range, the pixels inside the range are more likely to form some long strings instead of many short strings in traditional method, and data segment number should be small.

However, the range query on new method is not efficient as we thought. Look at Figure 2

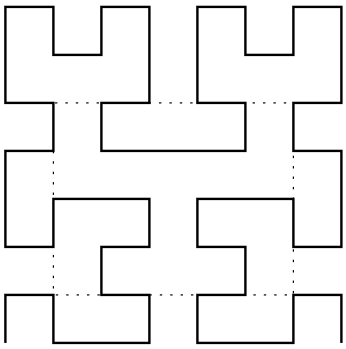

Figure 2: Range query on Hilbert Curve

Since the pixels are saved in Hilbert Order, the Hilbert curve can be viewed as the real data string on the disk. The dot rectangle is the query range. We can find that the range cuts the curve into several segments. Some segments are long but most of them along the border are quite short. The total segment number does not decrease, which can be found in our experiment later.

This can be improved by our new algorithm. The central idea of our algorithm is to increase the query range a little bit in order to decrease the segment number of Hilbert Curve inside the query range. Although we may have to retrieve more data than necessary, the query speed becomes faster.

3 Algorithm and discussion

We first do some research on Hilbert Curve. In Figure 3, suppose l is the left border of a query range in an image. For those pixels on the line, the Hilbert Curve can do the following three things:

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

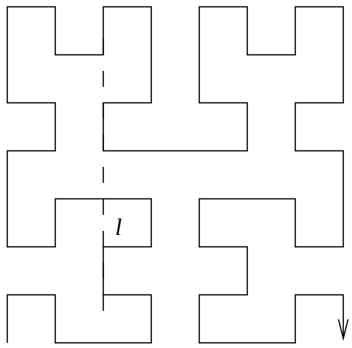

Figure 3: Hilbert Curve with a line

In the first case, the curve still stays inside the query range. (Here we assume that the border is in the range.) The rest two cases mean that the curve enter or leave the query range. We call that traverse a *cross* on the border and the pixel a *cross point*.

Theorem 3.1 *The possibility of a cross at any pixels on a border is about 50%.*

Proof: Suppose p is a pixel on the border and line l is the left border of a query range. First we symbolically move the line left a little bit. Now if p is a cross point, there must be a horizontal line segment on the curve which intersects with l, and p is its right end. Since the number of horizontal line segment on the Hilbert Curve is almost as many as the number of vertical ones, and each pixel (except two end points of Hilbert Curve) is the end point of two line segments, p has about 50% possibility to be the right end of a horizontal line segment. Since, p is randomly selected, the possibility of a cross at any pixels on a border is about 50%. $\square$

Hilbert Curve is a continuous curve. Each range query cut the Hilbert Curve into several segments. The end points of the curve segments inside the range can only appear on the query border and must be cross points. On the other side, each cross point must be either a beginning point or an end point of a segment. So the number of curve segments in the range must be the half of the number of cross points.

Corollary 3.2 *To any query range with parameter length c, the number of cross points on the border is $c/2$, and the number of Hilbert Curve segment inside the range is $c/4$.*

Our algorithm is based on the following fact: those cross points are not uniformly distributed. Suppose l is a line segment with the following function: $x = x_0, y \in [y_l, y_h]$. 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, the number of cross points can be even less. And so on.

The reason can be found from the locality attribute of Hilbert Curve.

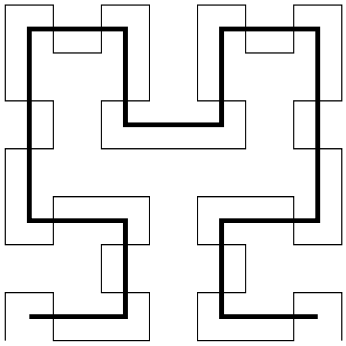

Figure 4: Hilbert Curve with order i and $i-1$

Suppose a curve with order i covers the whole grid space. As we know, the curve with order i can be generated from a curve with order $i-1$. We put the curve with order $i-1$ into the same grid space too. As shown in Figure 4. Put a line segment l ($x = x_0, y \in [y_l, y_h]$) on the image. If x_0 is an even number, any cross points of Hilbert Curve with order i must be a cross point of Curve with order $i-1$. (We omit the detail proof here.) So the average number of cross point on l if x_0 is even is about half of the average number without any limitation on x_0.

Based on this fact, we present algorithm 1 in Figure 5.

In this algorithm, we can find that the query range increases a little bit by changing the coordinate of the query range. That means in our algorithm, the system has to read some useless pixels which is out of the query range. But the algorithm is still efficient because the fact we mentioned at the first section: sequential readings are much faster than random access readings.

We also showed that the number of curve segments could still be less if the coordinate of the query range can be divisible by 4, 8 or 2^n. However the more we augment the query range, the more useless pixels we have to retrieve. A very extreme example is that we retrieve the whole image. At

```
point-set rangeQuery (lowx, lowy, highx, highy) {
    if (lowx is an odd number)
        lowx = lowx - 1; // increase the query range a little bit
    if (highx is an odd number)
        highx = highx + 1;
    /* same as lowy, highy */
    normalRangeQuery (lowx, highx, lowy, highy);
    filter the useless pixels;
}
```

Figure 5: Algorithm 1

that time, only *one* curve segment — the whole curve — will be retrieved.

Now We change algorithm 1 a little bit by introducing a new parameter n in Figure 6.

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

Figure 6: Algorithm 2

Algorithm 1 is a special case of algorithm 2 when we set $n = 1$. And if $n = 0$, we have the normal range query algorithm.

Suppose t_1 is the average time of a search on a disk in a system, t_2 is the average time of a 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 $c/8$ segment, which saves us $(c/8)t_1$ time. But at the same time, we have $c/2$ useless pixels in our augmented range. This costs us $(c/2)st_2$, where s is the *pixel size*. So our benefit 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 three times more. As $n = k$, our 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.

From the above formula, we can find that the selection of n^* in different system is different. It depends on how fast a search on a disk 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.

The other thing that affects the selection of n^* is the *pixel size*. If the *pixel size* is big, when we include one more pixel in the query range, we have to retrieve more bytes and spend more time. So we can not afford to include many useless pixels. That makes n^* to be a small number.

One thing we want to mention is that 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.

4 Experimental results

To access the merit of our algorithm, we implement it in C++. All the experiments are run on a Sun Untra 1 machine with 128 M memory. We compared the performance of our algorithm in different parameters along with traditional method. The CPU time is negligible, we base our comparison on (a) the segment number and (b) total data retrieving time.

The data space is an image with 1024×1024 pixels. In different application, different number of bytes are used to store the information of one pixel. We call it *pixel size*. So the total size of the 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 disk page size is 4 K.

We select row-wise method, $n = 0$ (normal query algorithm), $n = 1, 2, 3, 4$ and 5. We compare the segment numbers inside the query range. More than 10,000 possible positions are selected and the average results are listed in Table 1.

Query size	Row	$n = 0$	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$
750×750	750	743.91	356.45	182.73	84.40	48.81	23.85
600×600	600	619.00	273.82	148.80	69.96	39.78	19.78
450×450	450	475.04	212.12	116.75	56.30	29.31	14.57
300×300	300	306.67	142.06	73.92	35.79	16.88	8.17
150×150	150	155.04	71.78	37.70	17.54	9.04	4.01
50×50	50	51.27	24.71	13.36	7.23	4.28	1.41

Table 1: Number of data segment

In the above table, we can find that comparing to traditional method, normal query algorithm has almost the same segment numbers. In our algorithm, as n increases by 1, the segment numbers almost decrease by half. In large queries, it can be even more than half. Also, when $n = 1$, segment number in the range is almost a quartor of the total papameter length of the query range. Both results fit our theoritical calculation well.

Next, we want to compare the running time of different methods. Some issues are very important to the accuracy of the result:

- No buffer techniques (from C++ functions or Operating System) should be used. Otherwise we can not find the exact time for disk reading.
- The position of the query is very crucial to the final result, many queries should be selected randomly and tested. Only the average number can be used.
- No "special" readings. We found from our experiment that in a query, some readings spend much more time than average (about 1000 times more). The reason is that when the test

program ran those readings, the operating system switched the control to other programs. When our test program regained the control, much time has elapsed. We call those readings "special".

We do the following things to make the final result more accurate:

- We use low level function calls (such as *read, write* etc.) instead of high level function calls.
- We create a very large file on the disk (about 1 G bytes) and a pointer. Before we start a new query, we move the pointer randomly to a new position. That position is viewed as the start place of the image. Since the main memory is not very large, the old things we read have very little possibility to be stored in the memory buffer and reused later.
- More than 10,000 test cases are generated and the average time is counted for each data in our final result.
- We monitor each single reading. If it costs more time than our threshold (normally 1,000 times more than average), we do not count the time of that reading. The threshold is so high that in each query, number of "special" readings is less than 3 (comparing to about 1000 readings in one query).

We check different query size along with different *pixel size*. Results are in Table 2, 3 and 4

Look at each row of above three tables, we can find that to any query, as n increases, the total time decreases at first due to the decrease of the segment number, then it increases again. The optimized point is at n^*. We compare the value in n^* with traditional method, and put the result in the save column.

Another observation is about the value of n^*. When *pixel size* $= 12$, n^* is 2, sometimes 3. As *pixel size* $= 8$, n^* is 3. And n^* is 3 or 4 when *pixel size* $= 4$. If the *pixel size* is big enough, for example as big as one disk page, n^* will be 0, i.e. the normal data retireval algorithm will have the best performance.

Query size	Row	$n = 0$	$n = 1$	$n = 2$	$n = 3$	$n = 4$	Save (%)
800×800	1.28	1.26	1.09	1.15	1.22	1.27	14.44
700×700	1.00	1.02	0.85	0.90	0.96	1.00	15.25
600×600	0.73	0.71	0.63	0.67	0.73	0.77	13.79
500×500	0.58	0.63	0.48	0.46	0.53	0.58	20.44
400×400	0.33	0.33	0.31	0.34	0.37	0.39	6.81
300×300	0.24	0.24	0.20	0.20	0.23	0.25	15.77
200×200	0.13	0.14	0.11	0.12	0.12	0.13	16.39
100×100	0.060	0.062	0.052	0.052	0.062	0.069	13.26

Table 2: Average reading time (sec) when *pixel size* = 12

Query size	Row	$n = 0$	$n = 1$	$n = 2$	$n = 3$	$n = 4$	Save (%)
800×800	0.73	0.78	0.70	0.65	0.72	0.81	9.68
700×700	0.57	0.56	0.54	0.51	0.61	0.65	10.33
600×600	0.48	0.48	0.44	0.42	0.48	0.50	12.70
500×500	0.38	0.38	0.31	0.33	0.34	0.37	16.43
400×400	0.27	0.28	0.24	0.24	0.25	0.26	11.93
300×300	0.17	0.19	0.15	0.15	0.16	0.17	11.90
200×200	0.070	0.065	0.062	0.060	0.073	0.080	11.13
100×100	0.035	0.032	0.030	0.030	0.034	0.037	15.36

Table 3: Average reading time (sec) when *pixel size* = 8

Query size	Row	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$	Save (%)
800×800	0.37	0.37	0.36	0.34	0.35	0.39	6.66
700×700	0.33	0.34	0.30	0.30	0.29	0.31	13.16
600×600	0.26	0.27	0.25	0.22	0.24	0.27	13.83
500×500	0.18	0.20	0.17	0.16	0.15	0.18	16.90
400×400	0.12	0.13	0.12	0.11	0.10	0.13	12.96
300×300	0.079	0.078	0.075	0.076	0.075	0.081	5.69
200×200	0.043	0.043	0.040	0.038	0.039	0.042	10.63
100×100	0.030	0.031	0.028	0.026	0.026	0.028	11.88

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

5 Conclusion

The goal of our algorithm is to generate more sequential readings in a query. 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 to get best performance. The major conclusion is that the optimal value decreases when the *pixel size* increases.

Future research could check the performance in a paralle environment, and use the same technique on Hilbert R-tree.

References

[1] T. Bially. *Space-filling curves: Their generation and their application to bandwidth redunction*, IEEE Trans. on Information Theory, IT-15(6):658-664, November 1969.

[2] D. Ballard and C. Brown. *Computer Vision.* Prentice Hall, 1982.

[3] J. Griffiths. *An algorithm for displaying a class of space-filling curves*, Software-Practice and Experience, 16(5):403-411, 1986.

[4] K. Hirata and T. Kato. *Query by visual example — content based image retrieval.* In Advances in Database Technology, Vienna, Austria, 1992.

[5] T. Gevers and A. W. M. Smuelders. *An approach to image retrieval for image databases.* Database and Expert Systems Application, Prague, Czechoslovakia, 1993.

[6] J. Liang and C. C. Chang. *Similarly retrieval on pictorial databases based upon module operation.* Database Systems for Advanced Application, Taejon, South Korea, 1993.

[7] I. Kamel and C. Faloutsos. *Hilbert R-tree: An improved R-tree using fractals* Proc. of VLDB Conference, Santiago, Chile, Sept. 12-15, 1994, pp. 500-509.

[8] 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 Systems: Integrating Artificial Intelligence and Database Technologies, 3(3-4):231-262, 1994.

[9] http://glcf.umiacs.umd.edu/, 1999.