To be presented at SDH'96, Delft 12-16 August 1996,

the Seventh International Symposium on Spatial Data Handling.

The Spatial Location Code

Peter van Oosterom*
Cadastre Netherlands, Company Staff
P.O. Box 9046, 7300 GH Apeldoorn, The Netherlands.
Phone: +31 55 528 5163, Fax: +31 55 355 7931.
Email: oosterom@kadaster.nl
and
Tom Vijlbrief
TNO Human Factors Research Institute,
P.O. Box 23, 3769 ZG Soesterberg, The Netherlands.
Phone: +31 3463 56309, Fax: +31 3463 53977.
Email: vijlbrief@tm.tno.nl

Keywords: spatial searching, DBMS, Field-tree, Morton code, benchmark.

In this paper we present the Spatial Location Code (SLC) which is used for indexing and clustering geographic objects in a database. It combines the strong aspects of several known spatial access methods (Quadtree, Field-tree, and Morton code) into one SLC value per object. The *unique* aspect of the SLC is that both location and extent of possibly non-zero-sized objects are approximated by this *single* value. These SLC values can then be used in combination with traditional access methods, such as the b-tree, available in every database. It is expected that the typical query response time for spatial objects is reduced by orders of magnitude for a reasonably sized data set. The examples in this paper are all given in two-dimensional space, but the SLC is quite general and can be applied in higher dimensions.

1 Introduction

The Spatial Location Code (SLC) is designed to enable efficient storage and retrieval of spatial data in a standard (relational) DBMS. The requirements for the SLC are summarized below:

- be suitable for two-dimensional data;
- solve range queries efficiently (two-dimensional search rectangles), which implies spatial clustering and indexing;
- use one code per object and not sets of codes as these might be more difficult to manage within the DBMS;
- be (as) transparent (as possible) to the data model and queries;
- do not require DBMS kernel modifications, that is, use standard data types and access methods.

Because the solution must be implementable within any DBMS environment, the SLC extension has to be based on an existing type (e.g. `integer4`) in combination with an existing access method, such as the b-tree (Bayer & McCreight, 1973). A rough outline of this solution:

*A part of this research was performed while the author was still affiliated with the TNO Physics and Electronics Laboratory, P.O. Box 96864, 2509 JG The Hague, The Netherlands.

A. Add one 'spatial location code' SLC attribute to every table in the database which has a spatial (point, line, polygon, or box) attribute. The SLC is a one-dimensional code and every object gets exactly one code. This code is an approximation of the *location* and *extend* of possibly non-zero-sized objects in the two-dimensional space. It is possible that two different objects are approximated by the same SLC, but these objects will have about the same size and location.

B. Modify the table structure to b-tree according to the SLC attribute, that is, the objects are more or less stored on disk in the order defined by the b-tree. This primary index is responsible for the spatial clustering.

C. Define two functions:

1. `Compute_SLC`: computes the SLC of a two-dimensional box. First, the bounding box (bbox) of an object is computed, then the SLC for this box is computed. So, only one `Compute_SLC` is needed;
2. `Overlap_SLC`: determines the ranges of SLCs that do overlap the given two-dimensional search rectangle (query box).

D. Finally, one could define rules/procedures within the DBMS to make the SLC transparent to application:

1. fill the SLC-attribute in case of insert or update of tuples (possible in most DBMSs);
2. 'rewrite' box-overlap queries into SLC-queries (only possible in advanced extensible DBMSs, always possible in GIS front-end).

A combination of the Field-tree and Morton code (Quadtree) is used as a basis for the SLC. These structures will be described in Section 2. Abel and Smith (1983) describe a method based on the combination of the Quadtree and Morton code. However, a small object crossing a top level boundary will get a code corresponding to a large Quadtree region. Later on, during querying, this small object will be retrieved whenever the search area overlaps this large Quadtree region. This problem can be reduced by allowing more than one code per object; e.g. 4 codes (Abel & Smith, 1984). Drawbacks of this method are the increased storage use and the query where-clause will become more complex and therefore slower. The SLC method does not have these disadvantages. More details with respect to the `Compute_SLC` and `Overlap_SLC` functions will be given in Section 3. The benchmarks are presented in the subsequent section. Finally, conclusions can be found in Section 5.

2 Basic structures

In this section brief descriptions of the Region Quadtree, the Morton code, and the Field-tree are given. More details can be found in the given references.

2.1 The Region Quadtree

The *Quadtree* is a generic name for all kinds of trees that are built by recursive division of space into four quadrants. Samet (1984, 1989) gives an excellent overview. The best known Quadtree is the *region Quadtree*, which is used to store a rasterized approximation of a polygon. First, the area of interest is enclosed by a square. A square is repeatedly split into four squares of equal size until it is completely inside (a black leaf) or outside (a white leaf) the polygon or until the maximum depth of the tree is reached (dominant color is assigned to the leaf); see Figure 1. Each leaf can be assigned an unique label, the *quadcode*. During every split, the quads are assigned an number (e.g. SW=0, NW=1, SE=2, and NE=3). From the root to the leaf these number form a unique string; short strings (only a few splits) correspond to large regions, long strings correspond to small regions (a lot of splits).

2.2 The Morton code

The ordering of point locations with integer coordinates (or the raster cells of a two-dimensional grid or two-dimensional nodes of a Quadtree) can be used for *tile indexing*. It transforms a two-dimensional problem into a one-dimensional one. Several orderings have been described in the literature (Abel & Mark, 1990;

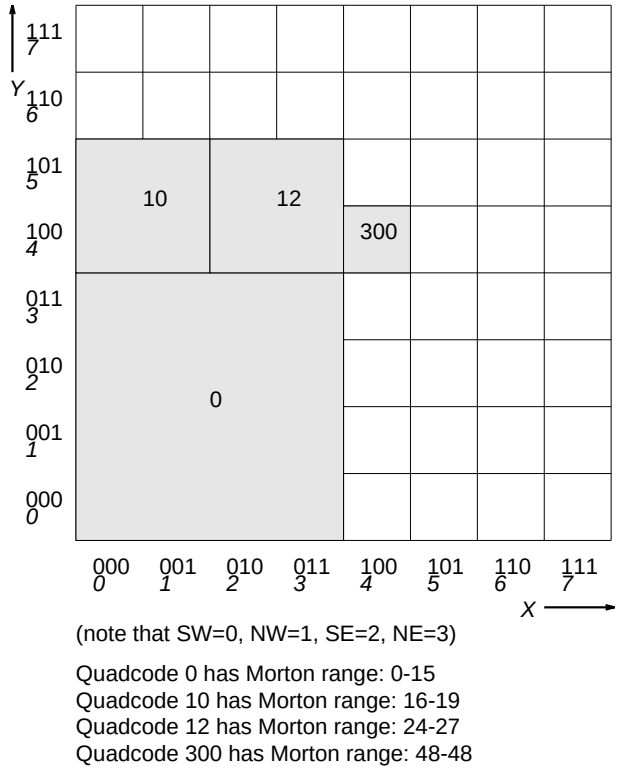

Fig. 1: The Quadtree and quadcodes

Goodchild & Grandfield, 1983; Jagadish, 1990; Nulty & Barholdi, III, 1994): row, row prime, Morton, Hilbert, Sierpinski, Gray code, Cantor-diagonal, and spiral.

During the storage of spatial data, the transformation to one dimension has to be made as the two-dimensional data has to be stored in the one-dimensional computer memory structure. The purpose of these orderings is clustering the spatial data: objects that are close together in the two-dimensional space should also be stored close together in memory (on the disk), as it is likely that they will be retrieved by the same query.

The bitwise interleaving of the two coordinates results in a one-dimensional key, called the *Morton* key (Orenstein & Manola, 1988). The Morton key is also known as Peano key, or N-order, or Z-order. For example, row $2 = 010_{bin}$ column $1 = 001_{bin}$ has Morton key $6 = 000110_{bin}$ (see Fig. 2).

The Morton key has several relationships with the Quadtree. As can be seen in Fig. 1, each quadcode can directly be translated into a range of Morton codes. Short quadcode strings relate to large Morton code ranges and long quadcode strings relate to small Morton code ranges. (Abel & Mark, 1990) have identified three desirable properties of spatial orderings:

1. An ordering is *continuous* if the cells in every pair with consecutive keys are 4-neighbors.
2. An ordering is *quadrant-recursive* if the cells in any valid Quadtree subquadrant of the matrix are assigned a set of consecutive integers as keys.
3. An ordering is *monotonic* if for every fixed x, the keys vary monotonically with y in some particular way, and vice versa.

Abel and Mark (Abel & Mark, 1990) conclude from their practical comparative analysis of five orderings (they do not consider the Cantor-diagonal, the Spiral and, the Sierpinski orderings) that the Morton ordering and the Hilbert ordering are, in general, the best alternatives. Goodchild (Goodchild, 1989) proved that the expected difference of 4-neighbor keys of an n by n matrix is $(n+1)/2$, which is exactly the same as in the case of row and row-prime ordering. However, the average farthest distance of the neighbors in the Morton ordering is higher than in the Hilbert ordering (Faloutsos & Roseman, 1989).

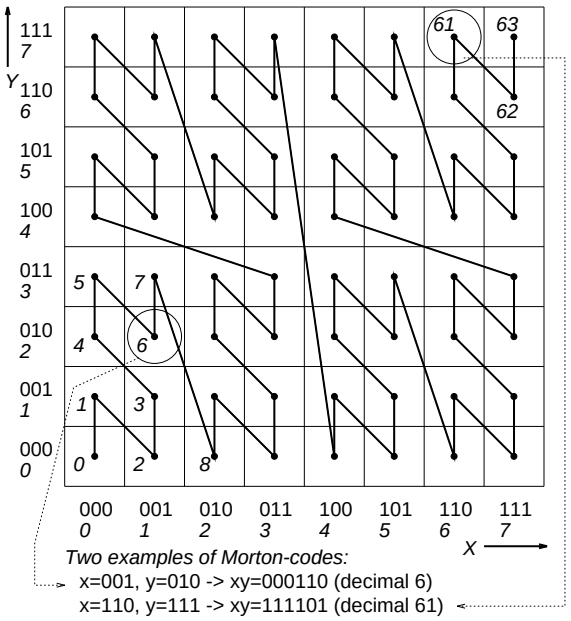

Fig. 2: The Morton-codes (shown on a 8∗8 grid)

2.3 The Field-tree

The *Field-tree* can store polylines and polygons in a non-fragmented manner. The Field-tree is one of the few structures which takes account of the fact that there may exist geometric objects of varying importance. During the last decade several variants of the Field-tree have been published by Frank *et al.* (1983, 1986, 1989). In this subsection attention will be focused on the *Partition Field-tree*. Conceptually, the Field-tree consists of several levels of grids, each with a different resolution and a different displacement; see Fig. 3. A grid cell is called a *field*. Actually, the Field-tree is not a hierarchical tree, but a directed acyclic graph, as each field can have one, two, or four ancestors. At one level the fields form a partition and therefore never overlap.

A newly inserted object is stored in the smallest field in which it completely fits. As a result of the different displacements and grid resolutions, an object never has to be stored more than three levels above the field size that corresponds to the object size. Note that this is not the case in a Quadtree-like structure, because the edges at different levels are collinear. A drawback of the Field-tree is that an overflow page is sometimes required, as it is not possible to move relatively large or important objects of an overfull field to a lower level field.

3 Spatial location code

Why should the SLC values not directly use a quadcode (or range of Morton codes)? There are two possible options, but each one has a drawback:

1. Assign one quadcode to every object. However, a small object may cross one of the major split lines and therefore get a short string quadcode. It is obvious that this large region does not approximate the object very well.
2. Alternatively, several quadcodes could be assigned to a single object. The corresponding regions now better approximate the object, but they are too complex to be managed efficiently in the DBMS (a set of values is associated with every tuple).

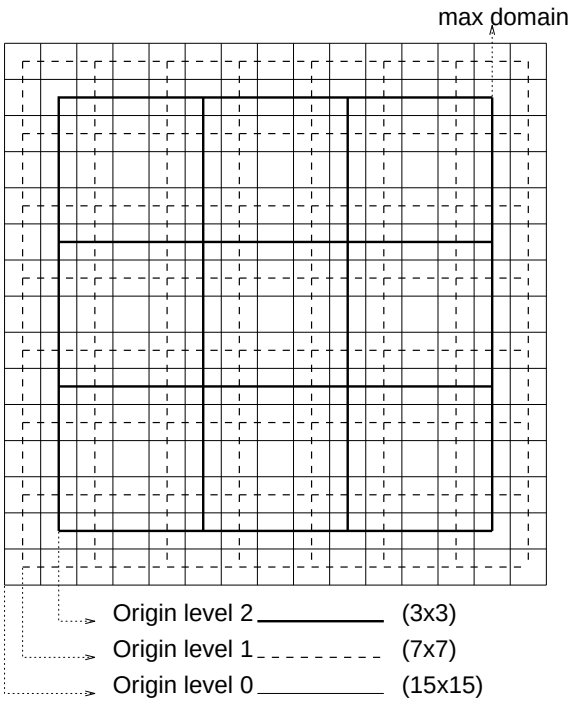

Fig. 3: The Field-tree (shown with 3 levels)

The Field-tree was used to get a better approximation of an object by a single value (the actual field in which the object is stored). Due to the 'shifting grids' of the Field-tree, the small objects will never be assigned to very large regions. The SLC encodes both the grid level and the grid cell at this level (e.g. by a Morton code) in one value. The SLC, as it has roughly been described in the introduction, could be used as shown in the SQL example of Fig. 4. Query 1 defines a table with a SLC-attribute. Then two records are inserted in this table in queries 2 and 3. Somehow, the SLC-attribute is also filled; e.g. by using a database rule which derives this value from the place, a box type, attribute. In query 4 a spatial selection is defined by using the `overlap` function with a given box. Again, somehow this query has to be translated into terms of SLC-values.

In subsection 3.1, the details of the SLC bit encoding are described. The pseudo code for the `Compute_SLC` and `Overlap_SLC` functions is given in subsection 3.2.

3.1 SLC bit encoding

For encoding objects with an SLC only their bounding box (bbox) is needed, because if an

Fig. 4: Example use of the SLC within the database

```
1. create table spatial (name char(100), place box, SLC integer4)
2. insert values into table spatial (name="jan", place="(0,0,1,2)")
    /* SLC filled with rule; e.g. SLC=115679 */
3. insert values into table spatial (name="klaas", place="(1,1,2,2)")
    /* SLC filled with rule; e.g. SLC=113456 */
4. select * from spatial where place overlaps "(0,2,2,3)"
    /* where-clause translated into where clause that uses SLC:
       where (SLC>100000 and SLC<150000) or (SLC>200000 and SLC<225000) or ... */
```

object has to fit in a cell of the Field-tree, just its bounding box has to fit. The following SLC encoding schema is based on `integer4`:

- Use variable length bit patterns for addressing grid-cells (finer grids require more bits for addressing);
- Start with a coarsest level of '255*255 grid'. The next finer level contains $2 * 255 + 1 = 511$ cells in one direction. The number of cells for the next finer levels are computed in the same way (1023, 2047, etc.); see columns 'level' and 'actual #cell 1 dim' in Fig. 5 (the other columns are explained later on).
- Allow flexible encoding of the number of levels (e.g. 5, 8, 11) and domain, so the SLC can be fine-tuned for a given application;

The basis is `integer4` values because they are more efficient in combination with the b-tree in many databases, compared to using a simple string encoding for the SLC (char).

The second column in Fig. 5 'SLC-code' describes the bit-pattern on a given level. The higher/coarser levels have more bits for the level indication and less 'xy'-bits, but this is no problem as there are also less grid cells to address. Note that the first bit is not used (to avoid possible problems with negative values of integer4). The column '#bit grid code' gives the number of bits available for addressing grid cells; 'x' and 'y' bits are added together. The column 'max #cell 1 dim' gives the maximum number of cells in one dimension with the available number of bits. The next column ('actual #cell 1 dim') gives the actual number of grid cells in one dimension for this level in the Field-tree. It can be observed that these are always less or equal than the available number. The total number of grid cell per level in the Field-tree is given in column '#cells per level'.

In Fig. 5 an example is given in which the finest level is 'level 0' with 25m*25m cells (this can be adapted if one needs larger domain; see '*25' in the column 'len cell in m'), the coarsest level is 'level 7' (with 3.2km*3.2km cells). Everything too large to fit into the coarsest cells (3.2km), gets a special SLC code (e.g. -1): the *overflow bucket*, which should not become 'too full'[1]. If one assumes the origin or lower-left point of the domain at (0,0), then the upper-right point of the domain is at (816,816) in km. In the column 'origin of domain (x) in m' it can be seen that the finer levels get a little offset in the Field-tree; see also Fig. 3.

The x- and y-coordinates of each grid cell are bitwise interleaved, which has the effect of the Morton encoding per level (which is very much related to the Quadtree; see Fig. 1). The level indication (bit-pattern; see Fig. 5) together with this Morton code, forms the complete SLC of each possible object in the given domain.

Note that this schema is general and can be extended to less or more levels (e.g. 5, 8, 11 or 15) or even different dimensions (e.g. three-dimensional). It might turn out that 8 levels are not required, given a certain data set, for stating the queries efficiently (ranges at 8 levels). When only 5 levels are used, the size of the smallest cell should probably be larger than 25m in order to avoid too many objects in the overflow bucket for a given application, e.g. 100m (and 1.6km on level 4).

[1] If the overflow bucket becomes too full, a more coarse level should be added to the Field-tree (SLC value).

Fig. 5: The bits in the SLC coding; `ORIGIN=(0,0), FINEGRID=25m`

```
   SLC-code:                          #bit max   actual len    #cells  origin of
        - 01's are for level          grid #cell #cell  cell   per     domain (x)
level   - xy's are for Morton code    code 1 dim 1 dim  in m   level   in m
-------------------------------------------------------------------------------
-  11xyxyxy xyxyxyxy xyxyxyxy xyxyxyxy              NOT USED!
-  10xyxyxy xyxyxyxy xyxyxyxy xyxyxyxy              NOT USED!

0  01xyxyxy xyxyxyxy xyxyxyxy xyxyxyxy 30 32768 32767  *25 1073676289 -1587.5
1  0011xyxy xyxyxyxy xyxyxyxy xyxyxyxy 28 16384 16383   50  268402689 -1575.0
2  0010xyxy xyxyxyxy xyxyxyxy xyxyxyxy 28 16384  8191  100   67092481 -1550.0
3  0001xyxy xyxyxyxy xyxyxyxy xyxyxyxy 28 16384  4095  200   16769025 -1500.0

4  000011xy xyxyxyxy xyxyxyxy xyxyxyxy 26  8192  2047  400    4190209 -1400.0
5  000010xy xyxyxyxy xyxyxyxy xyxyxyxy 26  8192  1023  800    1046529 -1200.0
6  000001xy xyxyxyxy xyxyxyxy xyxyxyxy 26  8192   511 1600     261121  -800.0
7  00000011 xyxyxyxy xyxyxyxy xyxyxyxy 24  4096   255 3200      65025     0.0
```

Fig. 6: Some constants (generic solution):

```
NRLEVELS=8   // 5 levels might be better
FINEGRID=25  // 100m might be better
THEORIGX=0
THEORIGY=0

for (i=0; i<NRLEVELS; i++) do
   NR[i]   = 2^(15-i);        // - 1;
for (i=0; i<NRLEVELS; i++)
   do SIZE[i] = FINEGRID * 2^i;

ORIGX[NRLEVELS-1] = THEORIGX;
ORIGY[NRLEVELS-1] = THEORIGY;
for (i=NRLEVELS-2; i>=0; i--) do
   ORIGX[i]= ORIGX[i+1] - SIZE[i+1]/4;
   ORIGY[i]= ORIGY[i+1] - SIZE[i+1]/4;

// bitpatterns for encoding level
BITS[MAXLEVEL]={
01000000 00000000 00000000 00000000,
00110000 00000000 00000000 00000000,
00100000 00000000 00000000 00000000,
00010000 00000000 00000000 00000000,
00001100 00000000 00000000 00000000,
00001000 00000000 00000000 00000000,
00000100 00000000 00000000 00000000,
...}
```

3.2 SLC pseudo code

The algorithms in this section are given in C-style pseudo code. First, remember the definition of the different Field-grids; see Fig. 5. In the pseudo code the levels are numbered 0 (finest grid) to 7 (coarse). The coarsest Field-tree grid has no (additional) displacement, the finer levels get more and more displacement; see Fig. 5, column 'origin of domain (x) in m'. In Fig. 6 some constants used in the pseudo code are initialized. Note that `ORIGX[NRLEVELS-1]` is initialized without additional displacement, and that the other levels get their additional displacement starting with `ORIGX[NRLEVELS-2]` down to finest level `ORIGX[0]`.

The function `Compute_SLC (bbox)` computes the SLC for a given object (bbox); see Fig. 7. First, it determines the finest grid in which the object might fit (lines 1 and 2). Then the exact grid level and SLC value is computed in line 3, by testing if the minimum and maximum point of the `bbox` have the same SLC value. The SLC value of these points is computed with the function `Compute_SLC_Point (level,x,y)`; see Fig. 8.

The `Overlap_SLC (bbox)` computes the set of SLC ranges that belong to the specified search box `bbox`; see Fig. 9. For each level, the quadcodes (or Morton ranges) corresponding to the `bbox` are computed (step 1). This is done with the function `Compute_quadcodes (level, bbox)` (Fig. 10), which in turn uses the recursive function `Add_quad_level (quaddomain, minSLC, maxSLC, bbox)`. This recursive function adds the new ranges for the specified domain to the set of ranges; see Fig. 11.

Note that when the number of ranges becomes large (e.g. too large for the database where-clause), then a few 'tricks' may be applied to reduce the number of ranges (these will include all required SLC-codes, but also include some unneeded SLC-codes, outside the search box):

- do not descend to the lowest level of the

Fig. 7: `Compute_SLC(bbox)`:

```
0. SLC = -1 // does not fit value!
1. get bbox max extend (in x or y dimension) -> max_ext
2. go to finest level with size>max_ext -> level[i]
3. for (j=i; j<i+3 && j<NRLEVELS && SLC== -1; j++) // worst case on level i+3
     if (Compute_SLC_Point(j,x_min(bbox),y_min(bbox)) ==
         Compute_SLC_Point(j,x_max(bbox),y_max(bbox)))
       SLC=Compute_SLC_Point(j,x_min(bbox),y_min(bbox))
```

Fig. 8: `Compute_SLC_Point(level,x,y)`:

```
1. x = (short)trunc((x-ORIGX[level])/SIZE[level]);
   y = (short)trunc((y-ORIGY[level])/SIZE[level]);
2. bitwise interleave xy -> SLC;
3. add level bitpattern (bitwise OR) -> SLC = SLC | BITS[level];
```

Quadtree, or

- join ranges even if there is a small gap between them, or
- reduce the number of levels to begin with (e.g. only 5).

Especially at the fine-level grids (e.g. 0, 1, and 2), the problem of the large number of ranges may occur. So, the 'tricks' mentioned above should first be applied at these levels. Experimental results and data distributions should be used to fine-tune the SLC encoding and functions.

4 Benchmarks

Two types of benchmarks are presented. In the first subsection, measurements with respect to the number of ranges (and the reduction thereof) are presented. In the second subsection, some results with actual (cadastral) data are given.

4.1 The number of ranges

As mentioned in the previous section, the number of ranges produced by the `Overlap_SLC` may become critical. Therefore, a function was created which closes the smaller gaps between two successive ranges. This function continues until the required number of ranges is reached. Every time two successive ranges are joined, other (unwanted) cells are also included; e.g. assume range 1 is (10,17) and range 2 is (20,23), then the joined range (10,23) contains unwanted cells 18 and 19.

Table 1: Ratio retrieved/required cells for the different query sizes (with a 5 level grid SLC)

max # ranges	.25*.25 km^2	.5*.5 km^2	1*1 km^2	2*2 km^2	4*4 km^2	8*8 km^2
6	36.6	36.5	18.1	23.8	10.8	19.1
10	1.5	1.8	2.3	4.2	3.0	2.8
15	1.0	1.2	1.4	1.7	1.8	1.8
20	1.0	1.1	1.3	1.4	1.5	1.5
30	1.0	1.0	1.1	1.2	1.3	1.3

For each test, 100 random queries of a certain size are generated in the following way: the location of the query is random over the whole domain (uniformly distributed), the size of the edge of a square query region is random over the range (0.75*X, 1.25*X), where X is the specified average size. The SLC values are based on a 5 level Field-tree.

Table 1 shows the measurements for six different query sizes (X is 0.25, 0.5, 1, 2, 4, and 8 km) and for five different upper bounds of the allowed number of ranges (at most 6, 10, 15, 20, and 30 ranges). Over the 100 random queries, the average initial number of corresponding SLC ranges per query (without closing range gaps > 0) are 13.1 (X=0.25), 22.5 (X=0.5), 40.3 (X=1), 77.9 (X=2), 155.6 (X=4), and 308.6 (X=8) respectively. These large number of ranges in the where-clause may be of problem for some RDBMSs. The effect of reducing the number of ranges is captured by the ratio between the number of cells addressed in this way (including the unnecessary cells) and the number of required cells. Except for the upper bound of at most 6 ranges, the same measurements are visualized in Fig. 12: on the x-axis, the average

Fig. 9: `Overlap_SLC(bbox):`

```
1. for each level do
      Compute_quadcodes(level, bbox)              // see function below
2. ranges of SLC codes can be joined to larger ranges
```

Fig. 10: `Compute_quadcodes(level, bbox):`

```
Initialize quaddomain              // depends on level
set_of_ranges={}                   // start with empty set
minSLC=0; maxSLC=(nr[level])^2 - 1 // the SLC ranges

Add_quad_level(quaddomain, minSLC, maxSLC, bbox)
```

Fig. 11: `Add_quad_level(quaddomain, minSLC, maxSLC, bbox):`

```
if (minSLC==maxSLC) do
  1. add (minSLC, maxSLC) to set_of_ranges                    // finished
else
  2. split quaddomain in quad[0], quad[1], quad[2], quad[3]
  3. for i=0 to 3 do
     3a. new_minSLC = minSLC + i*((maxSLC+1-minSLC)/4))
         new_maxSLC = minSLC + (i+1)*((maxSLC+1-minSLC)/4)) -1
     3b. if bbox FULL_COVERS quad[i]
            add (new_minSLC,new_maxSLC) to set_of_ranges    // finished
     3c. else if bbox PARTIAL_COVERS quad[i]                // recursion
            add_quad_level(quad[i], new_minSLC, new_maxSLC, bbox)
```

size of the query region; on the y-axis, the ratio between the number of retrieved and required cells. The different lines in the graph represent the different upper bounds of the allowed number of ranges. For interactive applications a ratio of at most 2 is acceptable. From Table 1 and Fig. 12, it can be concluded that about 15 ranges are required (for SLC values based on 5 levels).

4.2 The 'Flevoland' test

The test data set was provided by the Dutch Cadastral and Public Registers Agency. The data set Flevoland `grens` contains all (land owner) parcels in the 'new' province Flevoland created by man on the floor of the former sea; see Fig. 13, which shows the SLC implementation in our GIS GEO++ (Vijlbrief & van Oosterom, 1992; van Oosterom & Vijlbrief, 1994). For spatial indexing, the SLC method has been applied with the following specifications: 5 levels, finest grid 100m, maximum number of ranges in a query where clause is 20. The total area of the bounding box of this province is about 3000 km^2 of which about 2000 km^2 are actually filled with data.

The data density varies a lot: very high density in the center of towns, medium density in the other urban areas, low density in the natural/agricultural areas, and very low density in the large lakes 'Markerwaard' and 'IJsselmeer'. Note that in Fig. 13 all objects with an SLC value in Field-tree 'level 0' are gray, and all others are black. The overview window (upper-

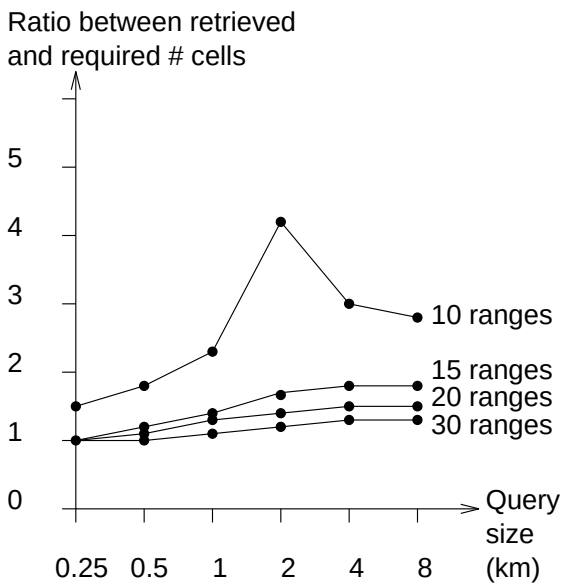

Fig. 12: Ratio retrieved/required cells (with a 5 level grid SLC)

Table 2: The 10 different queries

query nr.	bounding box of query region (coordinates in RD km)	area (km^2)
1	(150.0,490.0,160.0,500.0)	100
2	(155.0,495.0,160.0,500.0)	25
3	(170.0,525.0,175.0,530.0)	25
4	(172.0,526.0,173.0,527.0)	1
5	(172.0,526.0,172.5,526.5)	0.25
6	(180.0,520.0,190.0,525.0)	50
7	(180.0,520.0,181.0,521.0)	1
8	(180.5,520.5,181.0,521.0)	0.25
9	(175.0,500.0,180.0,505.0)	25
10	(179.0,500.0,180.0,501.0)	1

right) is a selection of all features with SLC values above Field-tree 'level 3'. The representation is quite good and it can be retrieved and displayed very efficiently. This indicates that the SLC values can also be used for *map generalization*.

4.3 The timings

The hardware configuration consists of a Sun SparcStation 10 (2 processors, 64Mb Main memory). The most important software components are Solaris 2.4 and CA OpenIngres 1.1/03 (beta) with OME/SOL (Object Management Extension/Spatial Object Library). In the benchmarks 10 representative queries are used (see Table 2) and every bbox range query is translated into a similar SLC query. Besides the `grens` table with 156.998 records, also the `lynstring` table with 633.397 records has been used in the benchmarks. The `lynstring` table contains the topographic features. Both tables use the OME/SOL spatial data types line(33) and box (and compression). The table `grens` takes 66.8Mb when using a btree storage structure. For the table `lynstring` this figure is 264.8Mb. The benchmarks are based on counting objects and not on actually retrieving them. This gives the best impression of the effect of (spatial) indexing.

When using no SLC value the DBMS has to do a sequential scan, which takes always about the same time. For the SLC index scan benchmarks, the response times (column 4 in Table 3) are

Table 3: Querying the table `grens`

query	no slc		slc		#slc_obj/
	sec	#obj	sec	#slc_obj	#obj
1	691	652	15	3650	5.6
2	718	454	7	1212	2.7
3	724	2609	14	3267	1.3
4	922	1000	5	1182	1.2
5	734	682	5	952	1.4
6	712	10434	82	23458	2.2
7	655	91	4	253	2.8
8	647	3	2	72	24.0
9	647	16045	60	17924	1.1
10	660	31	3	76	2.5

Table 4: Querying the table `lynstring`

query	no slc		slc		#slc_obj/
	sec	#obj	sec	#slc_obj	#obj
1	2861	32623	193	51243	1.6
2	3023	12528	73	21371	1.7
3	2871	5992	39	7403	1.2
4	2973	1818	11	2247	1.2
5	3222	1132	9	1610	1.4
6	2843	21477	262	53938	2.5
7	3457	169	14	441	2.6
8	2862	11	3	80	7.3
9	2707	51497	229	62397	1.2
10	2773	304	15	575	1.9

proportional to the number of counted objects (column 5 in Table 3): about 200 counted objects per second, unless the number of counted objects is very low. This indicates that the SLC method with btree is functioning well and reduces the query times from several hundreds of seconds to several seconds for restrictive queries. When comparing the number of objects with overlap (column 3) with the number of objects reported by the SLC method (column 5), it can be seen their ratio (column 6) is quite close the expected ratio; see previous subsection.

The effect of using the SLC method becomes even more important when the table becomes larger as can be seen in Table 4, which present the measurements for the large `lynstring` table.

Using 5 levels seems a very practical solution. In the current Cadastral software (based on a network DBMS and an own implementation of

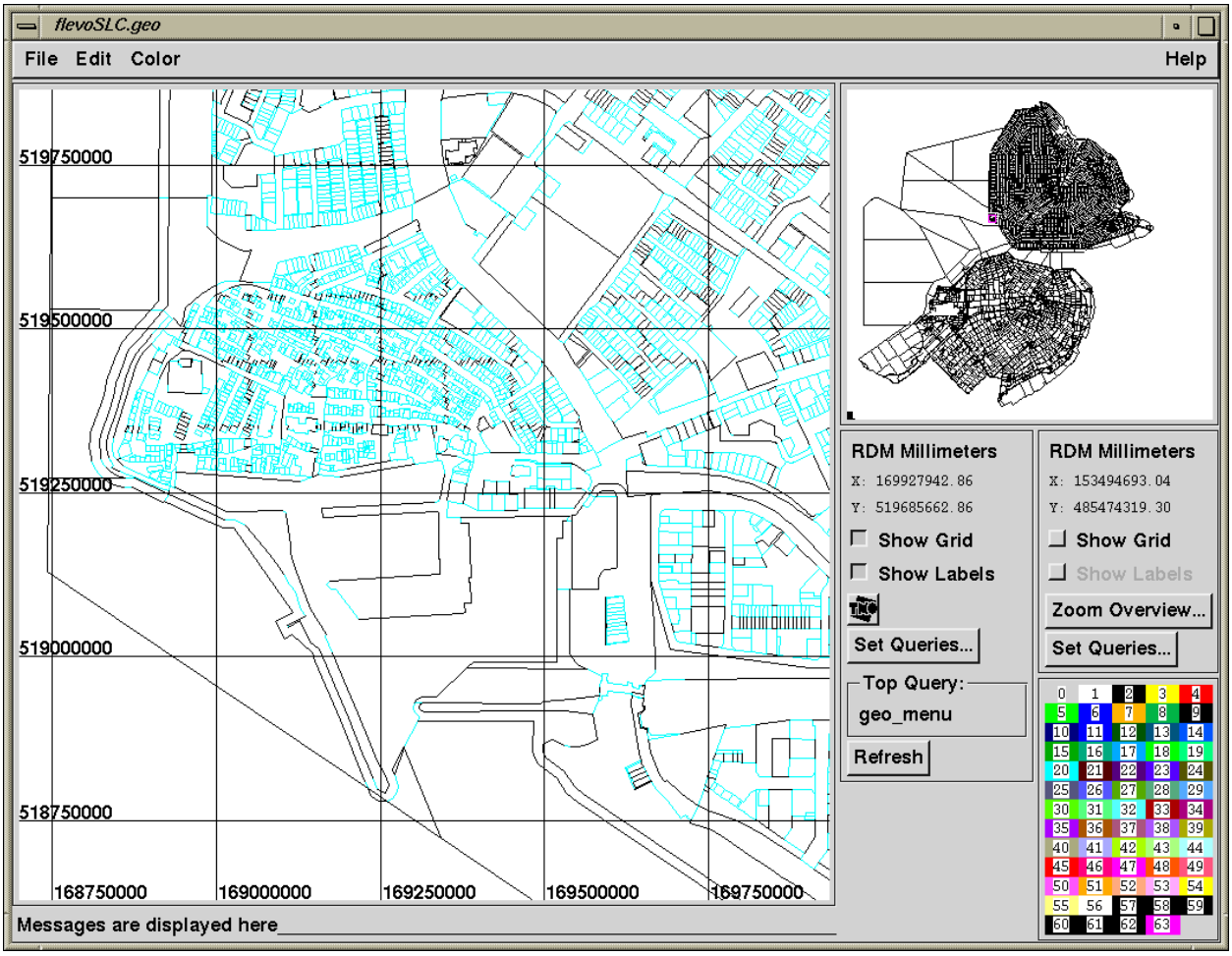

Fig. 13: The Flevoland `grens` benchmark in GEO++ (an area of about 1.25km $*$ 1.25km near the harbor of Urk): 'level 0' SLC objects displayed in gray, others in black

a Field-tree for geometric data), the number of levels is 5 and the finest grid-size is 100m. This Field-tree has been satisfactory for the large amounts of data stored in the database (called LKI). Now the LKI database is moving from a network DBMS to a relational environment, and the Field-tree is maintained by means of the SLC values. In July 1994, the LKI database contained about 20.000.000 `lynstring` features and 10.000.000 `grens` features. So in the heavily used part of the domain, the number of SLC values is about equal to the number of `lynstring` features. Not all SLC values will be used, and some will be used more than once.

5 Conclusion

The pros and cons of the SLC values are summarized in the next subsection, followed by a description of possible future work.

5.1 Advantages and disadvantages

The disadvantages of the current implementation of the SLC values:

1. The SLC approximation of an object can be quite coarse, especially in case of a 'long' (non-square) object. This will result in unnecessary selection of some objects based on their SLC.
2. The structure has to be tuned for a given application: the domain and the number of levels have to be chosen, but this can be done by means of automatic data analysis.
3. The number of SLC ranges in a query (as result of the `Overlap_SLC (bbox)` function) can be large. Several solutions for this prob-

lem have been described in Sections 3 and 4. One other alternative is to have multiple queries; e.g. one per level.

The most important advantages of the SLC value can be summarized as:

1. The SLC is very compact, only 4 bytes per object when `integer4` is used. Compared to a `bbox` based on 8 byte floating point coordinates as used in the R-tree (and many other structures), this is only 1/8.
2. The SLC can be used in many DBMS's as long as `integer4` and an index structure is supported (b-tree).
3. The SLC can be transparent to the user if the DBMS supports triggers and procedures.
4. The SLC will be even more efficient if it is implemented inside a DBMS.
5. The SLC provides good clustering and indexing, and will therefore enable efficient spatial DBMS queries.
6. The SLC can also be used to support 'map generalization' and multi-scale queries by retrieving only the coarse levels (which contain only large or important objects).
7. Empty SLC grid cells do cost nothing. Further, overfull cells are not possible, because many objects can have the same SLC value.

5.2 Future work

Future work will include performing more benchmarks with other real data sets and evaluating other promising orderings instead of the Morton code; e.g. Hilbert or Sierpinski orderings.

One important use of the SLC-values, other than spatial searching, is selection for interactive generalization: for an overview map (small scale) only the coarse grids have to be used; see Fig.13. However, small but important objects may be missed in this way, because the are stored in one of the finer grids. Therefore, the `Compute_SLC` function could be modified in a way that a user may specify a lower bound for the finest grid. In this way, important objects may be stored at coarser ('more important') grid levels even if they would fit at finer levels

Another improvement is having at each level two grids of the same resolution: the *shadow grid technique*. The second grid is just translated in x and y-direction by halve the size of a grid cell. In the way an object is never stored more than 1 level higher (coarser) than its own size (instead of 3 levels in the original method). This will reduce the number of overflow objects and is also better for generalization. The drawback is that twice as much ranges of SLC-values are required in the where-clause of a spatial overlap query.

Acknowledgments

Many valuable comments and suggestions were made in preliminary discussions with Chrit Lemmen and Tapio Keisteri. Paul Strooper made important additional comments with respect to both the contents and the presentation of the paper. Without their contributions we could not have achieved the current results. Finally, three anonymous reviewers of SSD'95 inspired us to rewrite the paper and to add the timings of queries to the benchmark section. The Dutch Cadastral and Public Registers Agency sponsored the described research and also provided the cadastral maps, which were used in our benchmarks.

References

Abel, D. J., & Mark, D. M. (1990). A Comparative Analysis of Some Two-Dimensional Orderings. *International Journal of Geographical Information Systems*, *4(1)*, 21–31.

Abel, D. J., & Smith, J. L. (1983). A Data Structure and Algorithm Based on a Linear Key for a Rectangle Retrieval Problem. *Computer Vision, Graphics and Image Processing*, *24*, 1–13.

Abel, D. J., & Smith, J. L. (1984). A Data Structure and Query Algorithm for a Database of Areal Entities. *The Australian Computer Journal*, *16(4)*, 147–154.

Bayer, R., & McCreight, E. (1973). Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, *1*, 173–189.

Faloutsos, C., & Roseman, S. (1989). Fractals for secondary key retrieval. in *Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pp. 247–252.

Frank, A. U., & Barrera, R. (1989). The Fieldtree: A Data Structure for Geographic Information System. in *Symposium on the Design and Implementation of Large Spatial Databases, Santa Barbara, California*, pp. 29–44 Berlin. Springer-Verlag.

Frank, A. (1983). Storage Methods for Space Related Data: The Field-tree. Tech. rep. Bericht no. 71, Eidgenössische Technische Hochschule Zürich.

Goodchild, M. F., & Grandfield, A. W. (1983). Optimizing Raster Storage: An Examination of Four Alternatives. in *Auto-Carto 6*, pp. 400–407.

Goodchild, M. F. (1989). Tiling Large Geographical Databases. in *Symposium on the Design and Implementation of Large Spatial Databases, Santa Barbara, California*, pp. 137–146 Berlin. Springer-Verlag.

Jagadish, H. V. (1990). Linear Clustering of Objects with Multiple Attributes. in *ACM/SIGMOD, Atlantic City*, pp. 332–342 New York. ACM.

Kleiner, A., & Brassel, K. E. (1986). Hierarchical Grid Structures for Static Geographic Data Bases. in *Auto-Carto London*, pp. 485–496 London. Auto Carto.

Nulty, W. G., & Barholdi, III, J. J. (1994). Robust Multidimensional searching with spacefilling curves. in *Proceedings of the 6th International Symposium on Spatial Data Handling, Edinburgh, Scotland*, pp. 805–818.

Orenstein, J. A., & Manola, F. A. (1988). PROBE Spatial Data Modeling and Query Processing in an Image Database Application. *IEEE Transactions on Software Engineering, 14 (5)*, 611–629.

Samet, H. (1984). The Quadtree and Related Hierarchical Data Structures. *Computing Surveys, 16 (2)*, 187–260.

Samet, H. (1989). *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, Mass.

van Oosterom, P., & Vijlbrief, T. (1994). Integrating Complex Spatial Analysis Functions in an Extensible GIS. in *Proceedings of the 6th International Symposium on Spatial Data Handling, Edinburgh, Scotland*, pp. 277–296.

Vijlbrief, T., & van Oosterom, P. (1992). The GEO++ System: An Extensible GIS. in *Proceedings of the 5th International Symposium on Spatial Data Handling, Charleston, South Carolina*, pp. 40–50 Columbus, OH. International Geographical Union IGU.