Aalborg Universitet
Department of Computer Science, Frederik Bajers Vej 7E, DK 9220 Aalborg Øst

Title:

Indexing Moving Objects Using Layered Space-Filling Curves

Project period:
Dat5, Sep. 2th 2005 - Jan. 9th, 2006

Project group:
E4-110

Group members:
Jens Frøkjær
Palle B. Hansen
Ivan V. S. Larsen
Tom Oddershede

Supervisor:
Kristian Torp

Copies: 6

Article page count: 15

Abstract:

Indexing moving objects with spatial extension is very useful in many application areas. Today the dominant data structure for indexing spatial objects is the R-tree family. In this paper, we present a method, based on space-filling curves and B^+-trees that uses layers in order to assign only one space-filling index to each object. The main purpose of this approach is to speed up modifications. When dealing with moving objects the number of modifications is often orders of magnitude larger than the number of queries. We present a method for shifting layers that guarantees that objects of a certain size can fit on a layer in order to minimize the number of layers while retaining good selectivity. We also show how range and nearest neighbour queries are performed. Finally a performance study is presented and the results are compared to both the R^+-tree and the canonical approach for indexing spatial objects using space-filling curves. The study shows that the R^+-tree has 98% more overhead when modifying compared to our method. This comes at the expense of 52% slower queries for our method.

Jens Frøkjær *Palle B. Hansen*

Ivan V. S. Larsen *Tom Oddershede*

Indexing Moving Objects Using Layered Space-Filling Curves

Jens Frøkjær, Palle B. Hansen, Ivan V. S. Larsen, and Tom Oddershede
Department of Computer Science, Aalborg University
{fr0,palle,ivanvsl,tom}@cs.aau.dk

January 9, 2006

Abstract

Indexing moving objects with spatial extension is very useful in many application areas. Today the dominant data structure for indexing spatial objects is the R-tree family. In this paper, we present a method, based on space-filling curves and B^+-trees that uses layers in order to assign only one space-filling index to each object. The main purpose of this approach is to speed up modifications. When dealing with moving objects the number of modifications is often orders of magnitude larger than the number of queries. We present a method for shifting layers that guarantees that objects of a certain size can fit on a layer in order to minimize the number of layers while retaining good selectivity. We also show how range and nearest neighbour queries are performed. Finally a performance study is presented and the results are compared to both the R^+-tree and the canonical approach for indexing spatial objects using space-filling curves. The study shows that the R^+-tree has 98% more overhead when modifying compared to our method. This comes at the expense of 52% slower queries for our method.

1 Introduction

With the growth of mobile computing it has become possible to constantly track the location of moving objects, hence the need for location-based services is increasing rapidly. Such services have a wide variety of applications such as a shipping company keeping track of its ships and containers, biologists researching how different shoals of fish are moving, and telephone company knowing at all times where their customers' mobile phones are located. All of these application areas have in common that the location of the object must be transmitted to the relevant service.

One could imagine that the number of modifications are orders of magnitude larger than the number of queries, and the ability to handle large amounts of moving objects will be of even greater importance in the future as new location-aware services arise based on, e.g., RFID[Inc03] and Galileo[Tra05]. When RFID-tags become integrated in more and more products, data can be collected for later analysis. For example tracking the movement of customers in supermarkets, and using the data for optimizing store layout. The European Union is about to launch its GPS counterpart, the Galileo system which will provide more devices with location information.

The previously stated technologies have the potential to become areas of interests for spatial indexing. Due to the more moving location-aware devices and services that need to keep track of all of them, the result is more frequent updates.

The dominant indexing technique for spatial objects is the R^+-tree. However, it is rather expensive to maintain[KBPR].

This problem can be solved by using a B^+-tree, which performs better than R^+-trees, when heavy modification occurs[KBPR]. The B^+-tree only supports indexing points in a one-dimensional space. If spatial objects are to be indexed by a B^+-tree, a mapping from two dimensions to one is needed[OT02].

Space-filling curves is a method that is well suited for reducing the number of dimensions. The most popular

is the Hilbert space-filling curve that has very good clustering properties[MJFS01].

Canonical use of space-filling curves for spatial objects may cause much I/O, as the objects potentially are stored multiple times in the database[BKK99].

The method described in this article uses space-filling curves in a layered approach. This optimizes the modification speed of the moving objects in the database management system (DBMS) as we only assign one id to each object in order to store the object only once in the database. The focus of this article is on moving objects in a two-dimensional space.

This article is structured as follows. In Section 2 related work is presented. Background knowledge is presented in Section 3. Our method for indexing moving objects using layered space-filling curves is presented in Section 4. An extension to layered space-filling curves is to shift the layers, which is described in Section 5. Query types and their implementations is the topic of Section 6. In Section 7, a number of performance tests are described, and results are presented. Finally, Section 8 concludes the paper.

2 Related Work

Much work has been done within the area of indexing moving objects. This section is structured as follows: Firstly we shall look at work done within the area of R-trees. Then we will look at work regarding indexing spatial objects using space-filling curves. Finally, we shall look at different tools for data generation.

Indexing moving objects can be done in several different ways. Popular and efficient indexing method are the R-tree[Gut84] and the R^+-tree[SRF87]. Although they are applicable in many different situations, there are some problems with these data structures. For example parameters for good retrieval performance in the R-tree affect each other in ways which makes it impossible to optimize one of them without loss in the overall performance [BKSS90]. Therefore, [BKSS90] introduces the R*-tree as a method of coping with these problems.

The TPR-tree [SJLL00] is based on the R*-tree. The TPR-tree is a very efficient indexing and querying method for moving points for current and predicted future positions. The paper also provides a workload generator that simulates objects moving along routes between destinations, and generates both modification and retrieval queries. We differ from this work as we want to index moving objects with spatial extent, but only the current positions. Hence we cannot use their workload generator.

As described in [OT02], B-trees have proven to be a very efficient index for many different types of data. Although the B-trees are intended for indexing objects with only zero dimensions it can be used to index multidimensional data by utilizing other dimensionality-reducing techniques. Our work is similar to the description [OT02] give of how to index spatial extended objects. This technique partitions the space into cells of uniform size, and each cell is given a unique id. This id could for example be assigned using the Hilbert space-filling curve [FR89]. An object with spatial extension may not be able to fit into a single cell. Therefore one object may have several cell ids. This complicates modifications and queries.

To prevent an object from having several cell ids, we use a layered or hierarchical approach like[YPK05]. Like [YPK05], we use a hierarchical structure to minimize the number of objects to be searched to answer a query. However, our method is based on B-trees and space-filling curves and is not a main memory approach. In [LK01] a fast method for constructing the Hilbert space-filling curve is presented as a tree structure, which we make use of in our work for calculating the Hilbert space-filling curve.

[BKK99] proposes a layered approach similar to ours based on the Z-Ordering where objects are only assigned one id. We differ from this approach by using shifted layers, where we can guarantee that objects of a given size always can fit on a certain layer. We also differ by having user defined layers that can minimize the number of layers. Therefore, fewer cells are needed to be searched when querying.

In order to keep our performance study as realistic as possible, we will make use of data generation tools. Generation of moving object data is the focus of [NPT03], which presents both the GSTD[TSN99] and the Oporto[SM01] data generators; they both support moving objects that have spatial extension. [NPT03] also introduces some real data sets collected from different natural phenomena like animals and hurricanes[Wea05]. In spite of hurricanes actually do have spatial extension, [Wea05] does only record them as points.

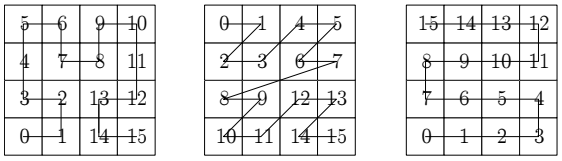
3 Background

In this section we present the background for using space-filling curves and some problems with the canonical use of space-filling curves indexing spatial objects[BKK99]. Finally, an example is introduced along with the query types which we will focus on.

3.1 Space-Filling Curves

When having a number of points in, e.g., a two-dimensional space, if these are to be indexing using a B^+-tree a reduction of the number of dimensions must be applied. This reduction can be achieved by the use of space-filling curves. When the space is two-dimensional, instead of having to save an object with a coordinate pair, the location can be approximated in a single number.

This requires the original space to be divided into a number of cells, and each of these cells is given a number according to the space-filling curve used. The world is divided using a static grid that has the following properties: It ensures all cells are of equal size. There is no overlap between cells, and the whole world is guaranteed to be covered by the grid. Examples of space-filling curves can be seen in Figure 1, inspired by [Sam04].



(a) Hilbert (b) Morton (c) Row-prime

Figure 1: Examples of space-filling curves

A space-filling curve strives to preserve the clustering properties when mapping to zero dimensions. According to [MJFS01] the Hilbert space-filling curve has good clustering properties.

When dealing with points only, the canonical approach is very useful. But when using space-filling curves for objects with spatial extension several problems arise. Canonical use of space-filling curves is a mapping of a point in a multi-dimensional space to a single number. However, with spatial objects we want to map two-dimensional objects in a two-dimensional space into a single number.

An object with spatial extension might not be able to fit into exactly one cell in the grid, e.g., in Figure 1. Therefore, a single object may be required to have several different position ids, which causes more I/O when modifying the objects[BKK99].

To avoid giving a single object several ids the cell-size can be adjusted according to the largest object in the population. Although the cell-size is larger than any object, it cannot be guaranteed that an object would not intersect cell-borders. Furthermore, when dealing with objects that change size, it may not be possible to predict their maximum size.

When an object is smaller than the cell-size, dead space occurs. The term dead space describes how much cell-space is not occupied by an object. When querying objects execution time is affected by the cell-size, because minimizing dead space reduces the number of false positives. *Dead space* is a number between 0 and 1 where a lower number is better. The smaller the number the more of the cell is occupied by the object. The dead space is given from Equation 1. It shows that the object-size is subtracted from the cell-size and then divided by the cell-size.

$$\text{deadspace} = \frac{\text{cell-size} - \text{object-size}}{\text{cell-size}} \tag{1}$$

Index selectivity is a number between 0 and 1. It shows the average number of objects that must be examined when using only the index in a range query for a single point. That is, how many objects cannot be cut of solely based on the index. Equation 2 and 3 shows how index selectivity is calculated. In Equation 2 $\text{range}_\square$ is defined as the range query that returns both true and false positives from the cells intersected by the range query. This can be visualized as a range query, where all the objects have been extended to the same size and shape of the cells in which they are contained.

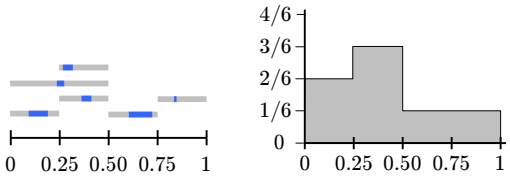
The variable c is the number of objects in the database. w_x and w_y are the lengths of the sides of the world. A simple example in one dimension can be seen in Figure 2(a) where all the objects are extended to fill all the cells they are in. The blue colour symbolizes the object and the gray colour symbolizes the dead space for the object.

The function that should be integrated is shown in Figure 2(b) which is based on the placement of the objects from Figure 2(a). Note that this function is not differentiable, hence it cannot be integrated. It

can be implemented using Riemann sums[EP02].

$$r_c = \int_0^{w_y} \int_0^{w_x} \frac{\text{count}(\text{range}_{\square}(x, y, x, y))}{c} dxdy \quad (2)$$

$$\text{index selectivity} = \frac{r_c}{w_x \cdot w_y} \quad (3)$$

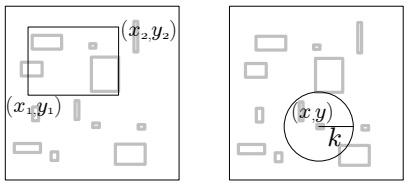


(a) Expanded objects (b) Function to be integrated

Figure 2: Example of index-selectivity

3.2 Query types

A range query[PSTW93] selects all objects within a certain rectangular area. A query of this type could be to ask "who are in a given city?". k-NN is the k nearest neighbours query[RKV95], which selects the k nearest objects to a given point. An example of a k-NN query could be when a customer requests a taxi, the service employee would inquire "who is the nearest free taxi to this customer?". This example chooses the taxi that has the shortest drive to the customer. Examples of these query types can be seen in Figure 3.



(a) Range (b) k-NN

Figure 3: Examples of Query Types

In Figure 3(b) the circle indicates the result set of the k-NN query. That is the smallest circle with centre (x, y) which encapsulates or intersects k objects.

3.3 Examples

Examples of moving objects which change size are clouds, bacteria colonies, or shoals of fish. Although it is not realistic to monitor, e.g., the position of shoals using only GPS, Galileo, or RFID, these technologies can be used with other kinds of sensor devices such as sonar. Shoals is an example of objects that move and also change their size and shape over time. When objects are not guaranteed to have a certain shape, it becomes harder to index them. The shape of a shoal is of highly dynamic nature.

To cope with the relatively inconvenient shapes the minimum bounding rectangle (MBR) is used. The MBR of an object is used in the index structure, and the actual shape of the object is saved, and is still available for computations. Using the MBR makes it possible to perform certain queries rapidly compared to querying on the actual shapes. Filtering the data for a query can be done rapidly using the MBR instead of the actual shape. This approach is used in R-trees[Gut84]. For example a range query can use the MBR to get an approximation of which objects are in a certain range. The query is then performed again on these objects, but this time the calculations are done on the actual shapes of the objects, and the answer for the query is found.

4 Layered Space-Filling Curves

In order to overcome the problems with the canonical use of space-filling curves outlined in Section 3.1, we first introduce *layered space-filling curves* that is described in this section. Adding layers reduces the amount of dead space, but the query overhead is increased.

4.1 A Layered Approach

The method described in this article assigns only one index number to each object. This approach makes modifications faster.

Instead of looking at the world as only one plane divided into cells, we look at the world as a set of planes on top of each other. Each of these planes is called a layer, which has the following properties:

- Layers are numbered bottom-up starting with 0.
- Space-filling numbers on one layer are always larger than any number on any layer below.
- A layer always has more cells than any layer above it.

- The top layer contains only a single cell.
- The smallest space-filling number is 0 and the largest is the number of cells minus one

An example of this approach could be with the divisions 2^n so that the layers are divided into $[32, 16, 8, 4, 2, 1]$ fractions, which enables use of the Hilbert space-filling curve on each layer. When an object is inserted, it is pushed through the layers top-down, like a sieve, until it cannot fit into a cell and then it is stored on the layer above. When updating an object, the same approach is used as with inserts, where the object is pushed through the layers and then given the space-filling number of the new cell. When deleting, the object is simply removed from the database.

Every cell on a layer is given a number according to a space-filling curve, e.g., the Hilbert space-filling curve. The numbering starts at Layer 0 where it is given numbers as the space-filling curve describes. At Layer 1 the numbering follows as described in Layer 0 with the only difference that the numbering starts where it ended at the layer below. The same procedure follows with the rest of the layers.

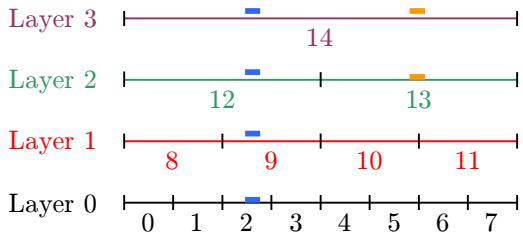


Figure 4: The layered approach in one dimension using divisions $[8, 4, 2, 1]$

Figure 4 is a conceptual view of the model in one dimension where the cells are numbered sequentially. Note that the figure is only shown in one dimension for the sake of simplicity. The blue and orange boxes symbolize objects and show on which layers they can fit into a cell. When an object touches a line, it indicates that this is where the object actually is stored. The figure shows that the blue object can be pushed the whole way down to Layer 0 and given the index 2. The orange cannot fit into the cell on Layer 1 as it will touch a cell-border. Therefore, it is stopped at Layer 2 and given the index 13. It is important that the objects are pushed to as low a layer as possible in order to increase selectivity and decrease dead space. Therefore, it would have been better if the orange object could have been pushed all the way down to Layer 0.

The method presented in this article is designed for a two-dimensional space containing objects with spatial extension. Figure 5 illustrates the method in two dimensions using the Hilbert space-filling curve. The arrow illustrates where the numbering ends at one layer and where it begins at the next layer.

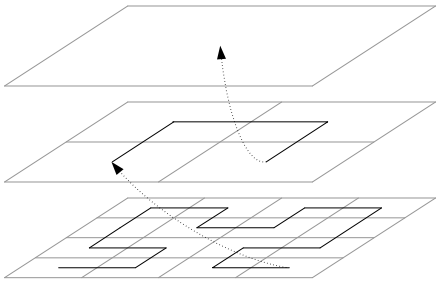


Figure 5: A layered Hilbert with divisions [4,2,1]

4.2 Layer Expansion

The Hilbert space-filling curve and other space-filling curves require that the layer must be divided into 2^{2n} cells[Sag94]. If there are not 2^n divisions the layer must be expanded in order to apply the space-filling curve. We call the number of divisions of the current layer for d, and we call the length of the sides of the world for w_x and w_y. The length of the sides of the new layer is denoted l_x and l_y

$$d' = 2^{\lceil \log_2(d) \rceil} \tag{4}$$

Now d' is defined as the smallest 2^n number larger than or equal to d as shown in Equation 4. The layer can be expanded with the missing cells by applying Equation 5 on both w_x and w_y.

$$l = w + \frac{w}{d} \cdot (d' - d) \tag{5}$$

Figure 6 outlines an example where d is 3 and shows that the number of divisions is increased so d' is 4. Note that the dotted cells will never be used for indexing, but only for applying the space-filling curve.

When introducing a layer with divisions not equal to 2^n it is not possible to use the top-down strategy described in Section 4.1, where an object is pushed down

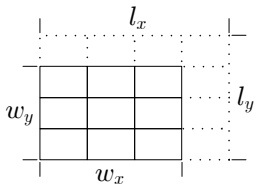


Figure 6: An example of how the layer is expanded

through the layers until it hits a cell-border. Therefore, a bottom-up strategy is introduced where an object is pushed from Layer 0 and upwards until it hits a layer where the object fits into a cell. Figure 7 shows that the object fits into Layer 0 and Layer 2. By using the bottom-up strategy it will be stored on Layer 0 and given the index 1. Note that Layer 0 originally had three divisions, but as this is not a 2^n number, the layer is expanded into four divisions.

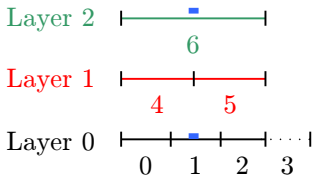


Figure 7: Indexing an object using the divisions $[3, 2, 1]$

4.3 Non-Overlapping Grids

Looking at Figure 4, e.g., if an object is positioned in the centre it would have to be pushed all the way up to Layer 3, giving poor selectivity. In order to overcome this problem we use grids that do not overlap across the different layers.

A strategy for selecting grids that do not overlap is to select a set of divisions D that are relative prime as shown in Equation 6.

$$\forall d, e \in D : d \neq e \Rightarrow \gcd(d, e) = 1 \tag{6}$$

When using non-overlapping grids it is more likely that objects will stay on the lower layers instead of being pushed all the way to the top layers. This is because that if an object by chance touches a cell-border on one layer, this line will not be in the same location on any other layer, and thereby it is more likely to find a cell on the bottom layers where the object fits.

Theorem 1 shows that if the set of divisions are relative prime then no grid on any layer will overlap. The variables x and y are positive integers, that symbolizes places where divisions are possible.

Theorem 1

$$\forall x, y, d, e \in \mathbb{N} : \gcd(d, e) = 1 \wedge x < d \wedge y < e \Rightarrow \frac{x}{d} \neq \frac{y}{e}$$

PROOF: There are three different cases, $d = e$, $d < e$, and $d > e$. If $d = e$, from $\gcd^{\text{i}}(d, e) = 1$ we know that $d = e = 1$ and there is no $x < d = 1$, hence the precondition is always false.

Now for the two latter cases. In the following it is assumed, without loss of generality, that $d < e$. Now again there are three cases, $x = y$, $x > y$, and $x < y$.

- If $x = y$ and $d < e$ meaning $d \neq e$ we have that $\frac{x}{d} \neq \frac{x}{e}$.
- If $x > y$ and $d < e$ we have that $\frac{x}{d} > \frac{y}{e}$, since something large, x, divided by something small, d, is always larger than something small, y, divided by something large, e.
- Now the last case $x < y$. $\frac{x}{d} \neq \frac{y}{e} \Leftrightarrow x \cdot e \neq y \cdot d$. If we try to prove that $x \cdot e = y \cdot d$, it must hold that $(x \cdot e) | (y \cdot d)$. We know that if $\gcd(p, q) = 1 \Rightarrow \text{lcm}^{\text{ii}}(p, q) = p \cdot q$. The smallest $y' \in \mathbb{N}$, which can be multiplied with d such that $(x \cdot e) | (y' \cdot d)$, is $y' = e$, but $y < e$ and hence it is not possible to find such a y and the inequality is fulfilled.

□

5 Shifted Layers

We will now introduce the notion of *shifted layers*, which is particularly useful when a lot of objects are roughly the same size. To ensure fast queries by keeping good selectivity the number of layers is reduced. This is done to reduce the number of layers, while keeping good selectivity to ensure faster queries.

Until now it has been required to have a top layer containing only one cell containing the whole world. This was done in order to ensure that there is a layer where all objects fit regardless of size. Shifted layers are also useful when knowing the maximum size of the objects being indexed.

[i] Greastest Common Divisor
[ii] Least Common Multiple

5.1 Shifting a Layer

When knowing the maximum size of objects, the traditional one-cell top layer can be replaced by a shifted layer reducing the dead space for the objects and increasing selectivity. A shifted layer is a layer that guarantees that any object of a given maximum size can always fit on this layer. In order to find the divisions for this layer Equation 7 must be applied, where w_x and w_y are the sizes of the world and o_x is the length of the longest object on the x-axis and o_y on the y-axis. The d_x and d_y are the numbers of divisions that should be on the x- and the y-axis respectively if the object was three times bigger. The reason for choosing three is that it is the smallest number of times a layer must be shifted to be able to contain an object of a certain size. Finally, the divisions for the layer is calculated which is the minimum of the two d_x and d_y floored. Note that floor is the largest integer smaller than the input and therefore we use ceil minus one.

$$d_x = \frac{w_x}{3 \cdot o_x} \quad d_y = \frac{w_y}{3 \cdot o_y} \quad d = \lceil \min(d_x, d_y) - 1 \rceil \quad (7)$$

When shifting, the layer is copied twice so we have three identical instances of the layer, these are called sub-layers. Now the second and the third sub-layers are moved respectively $\frac{w_x}{3d}$ and $\frac{2w_x}{3d}$ to left, they are also moved respectively $\frac{w_y}{3d}$ and $\frac{2w_y}{3d}$ down in the two-dimensional case. Figure 8 shows an example in one dimension of a shifted layer with three divisions.

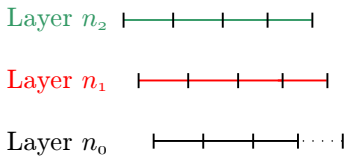


Figure 8: Shifted layers in one dimension where $d = 3$

Any layer in the method described in Section 4 can be replaced by a shifted layer. Often it is the top layer that is replaced by a shifted layer, as the selectivity is very poor in this layer. However, if the majority of the objects are of similar size, a shifted layer fitting this size could be beneficial.

5.2 Applying Space-Filling Curves

Now we need to apply the space-filling curve to the layer. Any non-shifted layer below the shifted layer is numbered as described in Section 4.1. The numbering at the shifted layer starts, as with any other layer, with the largest number below plus one. The next layer above, regardless whether it is shifted or not, will also start with the largest number at this layer plus one.

A shifted layer consists of three identical copies of the layer and therefore the three sub-layers have exactly the same number of cells. The numbering of the original layer is the space-filling numbering with the only difference that it is multiplied by three. The second sub-layer is given the space-filling numbering multiplied by three and then one is added. The last sub-layer uses the same approach only two is added instead of one. This is illustrated in the one-dimensional example in Figure 9, where the subscript 3 symbolizes a shifted layer.

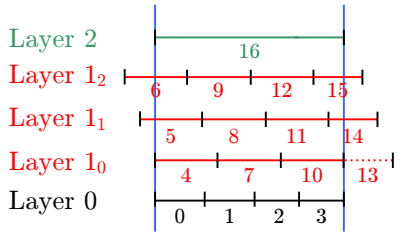


Figure 9: One-dimensional example with divisions $[4, 3_3, 1]$

Figure 10 shows an example of a shifted layer with three divisions, numbered according to the Hilbert space-filling curve. The black sub-layer is the original layer, where the Hilbert index is multiplied by three. The red grid is the second sub-layer with the Hilbert curve multiplied by three and one is added. Last is the green grid where the Hilbert curve is multiplied by three and two is added. The bottom layer illustrates an expansion of the world as shown in Figure 6.

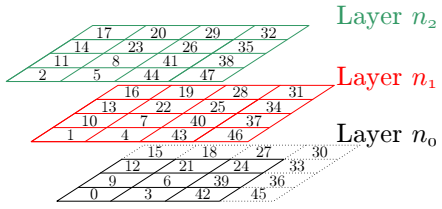


Figure 10: Two-dimensional shifted layer with Hilbert space-filling curves where $d = 3$

5.3 Insert Object

When an object is indexed using a shifted layer a slightly different approach, as the one described in Section 4.1, is used for placing the object. Firstly the object is put in the first sub-layer that is the black sub-layer on Figure 10. If it fits within one cell it is given the space-filling number of this cell. If the object does not fit into a cell in the first sub-layer, the object is "moved" $\frac{w_x}{3d}$ to the right and $\frac{w_y}{3d}$ up. The object is not actually moved, it is just in order to visualize the concept. Now if the object fits into a cell in the first sub-layer it is assigned this space-filling number plus one. This is equivalent to an object fitting into the red sub-layer on Figure 10. If the object does not fit into the second sub-layer either, then it is "moved" again $\frac{w_x}{3d}$ to the right and $\frac{w_y}{3d}$ up, and if it fits into a cell on the first sub-layer it is assigned this space-filling number plus two. This is equivalent to an object fitting into the green sub-layer on Figure 10.

When the divisions, for the shifted layer, are chosen from the maximum size of the objects, it is unavoidable that the dead space for a cell is at least $\frac{8}{9}$ as the object only occupies $\frac{1}{3}$ of the cell on each axis in two-dimensional case.

```
1  function getIndex(o_x1,o_y1,o_x2,o_y2,D)
2    foreach(d ∈ D)
3      if(isShifted(d)) S←[0, 1/3, 2/3]
4      else S←[0]
5      foreach(s ∈ S)
6        o'_x1←⌊(d·o_x1)/w_x + s⌋, o'_y1←⌊(d·o_y1)/w_y + s⌋
7        o'_x2←⌊(d·o_x2)/w_x + s⌋, o'_y2←⌊(d·o_y2)/w_y + s⌋
8        if(o'_x1 = o'_x2 ∧ o'_y1 = o'_y2)
9          return hilbert(o'_x1, o'_y1, d, D, s)
10   return −1 //error
```

Listing 1: Calculating index number for an object

Listing 1 shows the algorithm for calculating the index number for an object. The function takes in (Line 1) the coordinates for the lower left corner of the MBR (o_{x_1}, o_{y_1}), and the upper right corner of the MBR (o_{x_2}, o_{y_2}) and D which is the set of divisions. Line 2–9 iterates through all the layers. Lines 3–4 examines whether the layer is a shifted or not. If the layer is shifted Line 5–9 iterates through the three sub-layers. Lines 6–7 creates an integer coordinate-set that represents which cell the corners are in. The variable s is added in order to move the object, in order to fit into the shifted layers. Line 8 examines whether the whole MBR is inside a cell and if it is then the id of the cell is returned on Line 9.

6 Spatial Queries

In this section we will present algorithms for the range and k-NN queries. The algorithms builds up the `WHERE` clause that can be executed on an existing DBMS.

6.1 Range Queries

The range query returns objects that are within a specified range. Listing 2 shows the algorithm for the range

```
1  function range(q_x1, q_y1, q_x2, q_y2, D)
2    foreach(d ∈ D)
3      if(isShifted(d)) S←[0, 1/3, 2/3]
4      else S←[0]
5      foreach(s ∈ S)
6        q'_x1←⌊(d·q_x1)/w_x + s⌋, q'_y1←⌊(d·q_y1)/w_y + s⌋
7        q'_x2←⌊(d·q_x2)/w_x + s⌋, q'_y2←⌊(d·q_y2)/w_y + s⌋
8        for(q_x←q'_x1; q_x ≤ q'_x2; q_x←q_x + 1)
9          for(q_y←q'_y1; q_y ≤ q'_y2; q_y←q_y + 1)
10           h←hilbert(q_x, q_y, d, D, s)
11           if(q_x = q'_x1 ∨ q_x = q'_x2 ∨ q_y = q'_y1 ∨ q_y = q'_y2)
12             r←r ∪ {h} //partially included
13           else
14             R←R ∪ {h} //fully included
15   foreach(o ∈ R)
16     w_R←w_R ∪ {"index=o"}
17   w'_R←implode(w_R," OR ")
18   foreach(o ∈ r)
19     w_r←w_r ∪ {"index=o"}
20   w'_r←implode(w_r," OR ")
21   t←"(w'_R) OR ((w'_r) AND intersects(box(q_x1,q_y1,q_x2,q_y2
       )))"
22   return t
```

Listing 2: The algorithm for range queries

query. The function takes in (Line 1) the coordinates for the lower left corner of the range (q_{x_1}, q_{y_1}) and the upper right corner of the range (q_{x_2}, q_{y_2}) and D which is the set of divisions. Line 2–14 iterates through all the layers. Lines 3–4 examines whether the layer is shifted or not. If the layer is shifted Line 5–14 iterates through the three sub-layers. Lines 6–7 creates an integer coordinate-set that represents which cells the corners are in. The variable s is added in order to move the object according to the shifted layer as described in Section 5.3.

Lines 8–14 iterates through all the cells that the range covers. For each cell that is iterated by Lines 8–14, Line 10 finds the Hilbert number. Lines 11–14 examines the cells that are covered by the range, that is, whether a cell is completely inside the specified range. If it is, the space-filling number is added to the sets R otherwise it is added to the set r.

Lines 15–21 creates the actual SQL statement that specifies the range query. Lines 15–20 includes the ids that are covered by the range and adds them to w_r and w_R respectively and `OR` them together. This is done by the `implode` function on Line 17 and Line 20 which takes in a set and returns a string with all of the elements separated by the second argument. Line 21 creates the `WHERE` clause that examines whether or not the objects in the cells from r are actually within the range. Finally, the SQL statement is returned on Line 22. In a practical implementation, this can be an extremely long SQL statement. Therefore we use the `BETWEEN` operator when possible. This approach requires that the sets r and R are sorted.

6.2 k-NN Queries

The k-NN query returns the k nearest objects to a given point.

```
1  function k−NN(k, q_x, q_y, D)
2    l←sqrt((w_x·w_y)·k/n) / 2
3    a←k
4    do
5      l←l · c · sqrt(k/a)
6      a←count(range(q_x − l, q_y − l, q_x + l, q_y + l, D))
7      if (a = 0) a←0.5
8    while(a < k)
9    t←range(q_x − l, q_y − l, q_x + l, q_y + l, D)
10   t'←sort(t,distance(q_x, q_y))
11   o←number(t', k)
12   l'←distance(o, (q_x, q_y))
13   if (l' ≤ l) r←t
14   else r←range(q_x − l', q_x − l', q_x + l', q_y + l')
15   return r + " ORDER BY DISTANCE TO (q_x,q_y) LIMIT k"
```

Listing 3: The algorithm for k-NN queries

Listing 3 shows the algorithm for the k-NN query. The function takes in (Line 1) the number of desired objects, the coordinate-set of the point (q_x, q_y) and D which is the set of divisions. Line 2 creates a variable l which is an indication of statistically how large a portion of the world is needed to get k objects when a uniform distribution of objects in the world is assumed. n is the total number of objects in the database. The variable a from Line 3 indicates how many objects that are currently found, initially it is set to k. Lines 4–8 is a loop that makes a range query and keeps expanding it until it has found k objects. In Line 5 the variable l is increased slightly by using c which is a constant used for increasing the probability for getting a correct result in each iteration. We use a c value of 1.05, which is a expansion of 5%. Furthermore, l is increased with the square root of $\frac{k}{a}$, that is a calculation of how much the range needs to be increased statistically to find k objects. In a non-uniformly distributed world, the algorithm will adjust itself to how much it should expand in the next iteration, according to the number of objects currently found. Line 6 counts how many objects there are in the specified range. Line 7 examines whether a is zero and in that case it is set to 0.5 in order to prevent division by zero on line 5.

Line 9 executes the range query with the coordinates calculated in Lines 4–8. On Line 10 the result is sorted according to the distance to the query-point. Then the k'th element is selected on Line 11, and the distance to this object is calculated on Line 12. At Line 13 it is examined whether the whole result-set is in the already executed range, and if it is then the old result-set is used, else a new range query is done on Line 14. The result-set may be too large, but it is guaranteed to include at least the closest k objects. On Line 15 the result is sorted by distance and only the k closest objects are selected. The `DISTANCE TO` function can be implemented by the user as they see fit.

7 Performance Study

This section will examine how the methods perform with regard to inserts, updates, and deletes. Furthermore, tests are carried out using the spatial queries from Section 6.

7.1 Test Setup

The tests were performed on a 1.8 GHz Intel Pentium 4 processor with 1 GB RAM. The operating system was Microsoft Windows Server 2003 Enterprise Edition with Servicepack 1 running the MySQL 5.0.15 DBMS. MySQL implements spatial extensions according to the specifications of the Open GIS Consortium[Con05]. In MySQL spatial indexing is implemented using R^+-trees[MyS06]. MySQL formed the basis for comparing the indexing approaches designed in this article. In this section we will use different data sets, for carrying out the different tests. They are generated using the Oporto data generator[SM01].

We also compare our method to the canonical approach from Section 3.1, where the world is a single plane divided into cells, and if objects overlap cell-borders, the object is stored more times with the re-

spective space-filling numbers. Finally, tests are carried out where no index on the objects is maintained. The last test is useful for calculating the time used for I/O, in order to find the actual overhead of using a given index structure which is given by Equation 8. This equation shows the overhead of using an index for modifications.

$$\text{cost} = \frac{\text{with index} - \text{no index}}{\text{no index}} \tag{8}$$

In a relational database there are two obvious ways of implementing the canonical Hilbert. The first solution is to have a single table containing all data, and if an object must be partitioned it is saved more times in the table[ARR+97]. The other solution is to have two tables where data is put in one and the Hilbert index is put in the other. Our tests have shown that the first implementation performs the best and therefore will form the basis for the comparison.

The table schema consists for all the tables of an attribute containing the id, the polygon, and a tuple containing 200 bytes of data in order to make the tests more realistic. Furthermore, an attribute for the space-filling index is used when testing the Hilbert space-filling curve.

We only show results for single-threaded tests as MySQL only supports table-level locking for tables containing spatial objects[MyS06] hence the results should be equal. This is also supported by tests.

In the tests in this section we use a world area of $50,000 \times 50,000$ and the size of the objects has a normal distribution with a mean $\mu = 125,000$ and a standard derivation $\sigma = 20,000$. This means that 1,000 objects together on average occupy 5% of the world. All objects are squares and their positions are uniformly distributed.

All tests are performed five times where the best and the worst are discarded and the average of the three remaining is calculated.

7.2 Finding Divisions

In order to use the method described in this article a set of divisions must be selected. First of all the 2^n divisions from Section 4.1 are tested using the divisions $[32, 16, 8, 4, 2, 1]$; this is *data set 1*. These numbers are chosen as the divisions are doubled at each layer. We use a data set with the divisions $[31, 17, 8, 5, 3, 1]$, which are all relative primes. These numbers are chosen as the divisions are close to *data set 1*; this we will call *data set 2*.

Another data set has the divisions $[50_3, 41_3, 1]$; this we will call *data set 3*. These numbers are chosen based on Equation 7, which is calculated on the smallest 50% of the objects and the smallest 96%. The numbers are slightly adjusted to ensure they are relatively primes. This is illustrated in Figure 11.

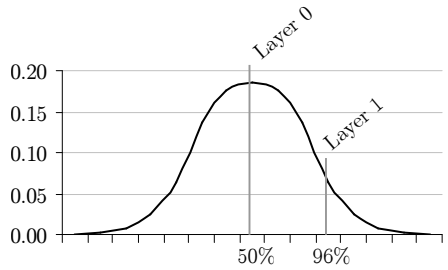


Figure 11: Choosing divisions for *data set 3*

When comparing to the canonical Hilbert space-filling curve we use 8 divisions; this is *data set 4*.

The previous data sets have the following data distribution over the layers. The tests were conducted with 50,000 objects in the database. *Data set 1* has the distribution $[61\%, 19\%, 10\%, 6\%, 3\%, 1\%]$ where the first number is the bottom layer. *Data set 2* has the distribution $[62\%, 30\%, 7\%, 0\%, 0\%, 0\%]$ and *data set 3* has the distribution $[96\%, 4\%, 0\%]$. Even though there is no objects on the top layer, it is not guaranteed that there never will be any.

	Dead space	Selectivity
Data set 1	96.6%	2.780%
Data set 2	96.5%	0.292%
Data set 3	87.6%	0.038%
Data set 4	99.7%	1.722%

Figure 12: Dead space and index selectivity

Figure 12 shows the dead space and index selectivity as described in Section 3.1. As *data set 3* has the best distribution with most objects on the bottom layer and furthermore it has the best dead space and index selectivity. This will form the basis for the comparison throughout the rest of this section. As it appears from Figure 12, *data set 3* has 87.6% dead space which is less than the theoretical upper bound described in Section 5.3. This is because the objects can fill the whole cell

as the divisions are not calculated based on maximum size.

7.3 Modification

In the following we will test how inserts, updates, deletes, and a mix of the three performs.

7.3.1 Insert

The insert test is conducted on an empty database. The data set consists of 1,000,000 objects, which are all inserted individually. At every 50,000 inserts 2,000 inserts are timed and the average time is plotted on the chart.

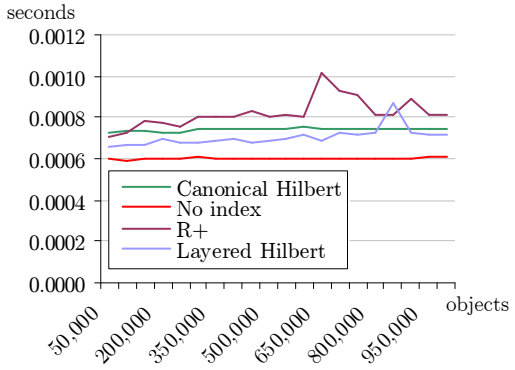


Figure 13: 1,000,000 inserts

In Figure 13, looking at the average times of the layered Hilbert, the R$^+$-tree, and without index, and applying Equation 8 we see that the overhead with the R$^+$-tree is 116% larger than the layered Hilbert.

7.3.2 Update

The update test is conducted on an empty database. 50,000 inserts are conducted and the 2,000 updates are performed and this is repeated. The testing is done by timing the 2,000 updates and calculating the average. The update test is done up till 1,000,000 objects.

In Figure 14, looking at the average times of the layered Hilbert, the R$^+$-tree, and without index, and applying Equation 8 we see that the overhead with the R$^+$-tree is 94% larger than the layered Hilbert.

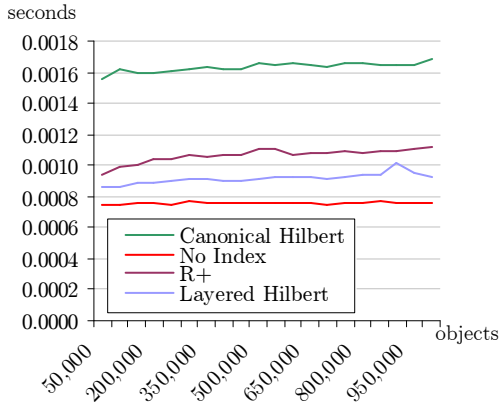


Figure 14: 1,000,000 updates

7.3.3 Delete

The delete test is conducted on a database with 1,000,000 preloaded objects. Every object is deleted individually and the same timing mechanism as with the update test is used. The x-axis shows how many objects there are in the database when the delete is conducted.

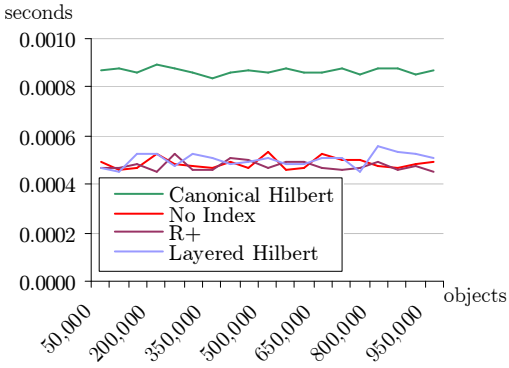


Figure 15: 1,000,000 deletes

As is appears from Figure 15, there is no significant difference between the tested methods except for the canonical Hilbert, as it has to delete more rows. Against intuition, the R$^+$-tree is very fast at deleting and rebalancing the index.

7.3.4 Mixed Modification

In order to test how the different indexes perform when altering the trees heavily, a workload is generated doing a mix of inserts, updates, and deletes. The work-

load consist of a set of preloaded object, and then modifications are done at the ratio 10-80-10 and 30-40-30 of inserts, updates, and deletes. 500,000 objects are preloaded. On the preloaded data we will perform 1,000,000 operations, which are timed. There is performed one transaction per operation.

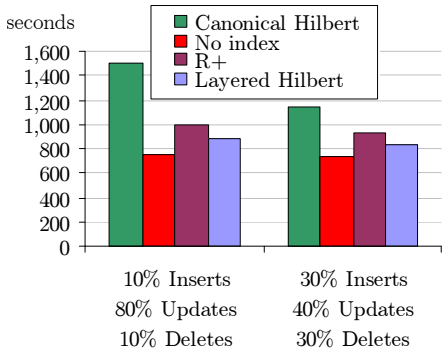


Figure 16: Mixed modifications

Figure 16 shows the mixed test performed with 500,000 preloaded objects. As it appears from the figure, the best performance is achieved with no index on the objects, because it does not need to maintain an additional index. With the ratio 10-80-10 we see that the overhead with the R^+-tree is 98% larger than the layered Hilbert. With the ratio 30-40-30 the overhead with the R^+-tree is 106% larger than the layered Hilbert. This is due to the higher number of inserts where the B^+tree has a larger advantage than on updates. Tests have shown that the ratio between the different methods are the same with 150,000 preloaded objects.

7.4 Querying

The method described in this article is designed for fast modification speeds. We will now test how much performance loss at query time the method has, compared to the R^+-Tree. The database is preloaded with 500,000 objects.

7.4.1 Range Query

Testing the range query, 100 range queries are performed and the average is calculated and plotted on the chart. We will change the percentage of the world, which is selected in the range.

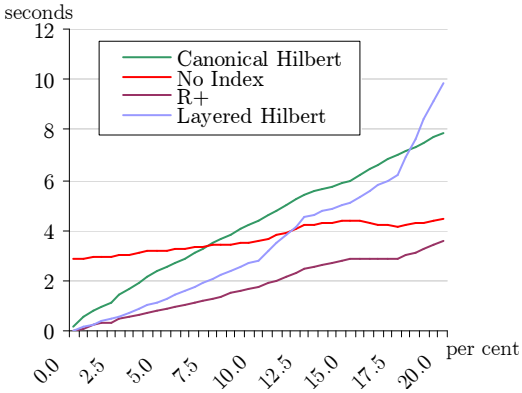


Figure 17: Range query

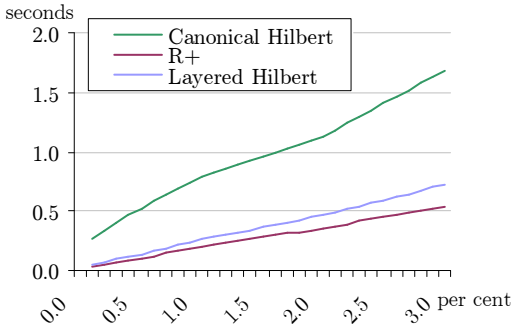


Figure 18: Magnification of Figure 17 from 0–3%

As it appears from Figure 17, the range query in the layered Hilbert is slower than the R^+-tree. It shows that when a range is larger than 10% of the world, sequential scan is faster than the layered Hilbert. Therefore, sequential scan should be used when using layered Hilbert when querying more then 10% of the world. The reason why the canonical Hilbert performs worse than our method could be that the `DISTINCT` keyword must be applied as the table contains duplicates which must be filtered out.

The tests show that layered Hilbert is 52% slower than the R^+-Tree when degenerated to sequential scan after 10%.

7.4.2 k-NN Query

Testing the k-NN query, 100 random query points are selected. These will be used for all the tests. We will change the number of objects we want to be returned, to see how it scales. However, MySQL does not implement the distance function and therefore it is not

possible to do the k-NN query directly. Therefore, we have implemented a distance function in MySQL that we will use on all the tests. The implemented distance function that is listed in Appendix A.

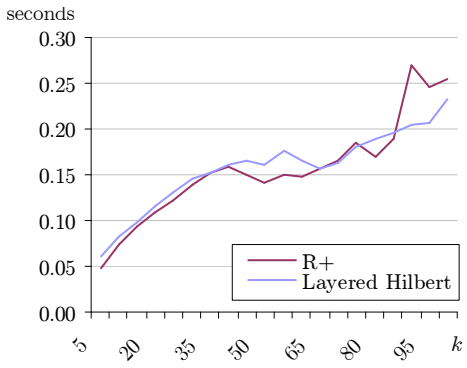


Figure 19: k-NN queries

As it appears from Figure 19 the performance of the layered Hilbert and the R$^+$-tree are about the same, which is a result of equal implementations. The reason why, against intuition, the two perform equally could be that the overhead of by using the distance-function is too large.

7.5 Summary

In a scenario with 2 % range queries and modifications in the ratio 10-80-10 inserts, updates, and deletes the worsened query speed is more than justified when having a modification/query ratio of 500 to 1.

8 Conclusion

In this article we have proposed a layered approach for indexing moving objects with spatial extension. We introduced the notion of shifted layers which proved to be very efficient for achieving better selectivity and reducing dead space.

We showed that the overhead for maintaining the index is significantly smaller than on the R$^+$-tree and the canonical way of using the Hilbert space-filling curve.

This modification speedup comes at the expense of a slightly lowered query speed. The number of modifications are often several orders of magnitude larger than the number of queries. Therefore, this sacrifice may be justified.

In this article, MySQL was chosen as DBMS as it is easy to install and use. However, the support for spatial objects is still very new. Some features are still missing, such as the k-NN query. Therefore, an interesting topic for further testing is to evaluate the implementation on a DBMS with full spatial support.

It would be interesting to look into how the method performs on skewed data. We have not been able to find a data generation tool, which fulfils our needs with respect to, e.g., skewedness, amounts and spatial extension.

Throughout this article user specified constants, world size and divisions, have been used. It would be interesting to make the method more dynamic in order to ensure the method itself can maintain and modify these constants.

Finally, future work includes generalizing the method to work in more dimensions.

Acknowledgements

The authors would like to thank José Moreira et al for providing us with the Oporto[SM01] source code. We would also like to thank Mario Nascimento for his help during data generation. Finally, we thank Kristian Torp for guidance and help throughout the project period. Without his feedback this article could not have been realized.

References

[ARR$^+$97] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer. Space-filling curves and their use in the design of geometric data structures. *Theoretical Computer Science*, 181(1):3–15, 1997.

[BKK99] C. Böhm, G. Klump, and H. Kriegel. Xz-ordering: A space-filling curve for objects with spatial extension. *Symposium on Large Spatial Databases*, (6), 1999.

[BKSS90] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *ACM SIGMOD International Conference on Management of Data*, 1990.

[Con05] Open GIS Consortium. Open gis consortium. http://www.opengis.org, November 2005.

[EP02] C. H. Edwards and D. E. Penny. *Calculus*, volume 6. Pintice Hall, 2002. ISBN 0-13-095006-8.

[FR89] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Symposium on Principles of Database Systems*, number 8, 1989.

[Gut84] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *ACM SIGMOD International Conference on Management of Data*, 1984.

[Inc03] Allied Business Intelligence Inc. Rfid white paper. http://www.dri.co.jp/free/abi_rfid02wp.pdf, 2003.

[KBPR] K. Kalpakis, J. Behnke, M. Pasad, and M. Riggs. Performance of spatial queries in object-relational database systems.

[LK01] J. K. Lawder and P. J. H. King. Querying multi-dimensional data indexed using the hilbert space-filling curve. *ACM SIGMOD Record*, 30(1), March 2001.

[MJFS01] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions On Knowledge And Data Engineering*, 13(1), 2001.

[MyS06] MySQL. Mysql 5.0 reference manual. http://dev.mysql.com/doc/refman/5.0/en/, January 2006.

[NPT03] M. A. Nascimento, D. Pfoser, and Y. Theodoridis. Synthetic and real spatiotemporal datasets. *IEEE Data Engineering Bulletin*, 26(2):26–32, 2003.

[OT02] B. C. Ooi and K. Tan. B-trees: bearing fruits of all kinds. In *Australasian Conference on Database Technologies*, number 13, 2002.

[PSTW93] B. Pagel, H. Six, H. Toben, and P. Widmayer. Towards an analysis of range query performance in spatial data structures. In *Symposium on Principles of database systems*, number 12, 1993.

[RKV95] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *International conference on Management of data*, 1995.

[Sag94] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994. ISBN 0-387-94265-3.

[Sam04] H. Samet. Object-based and image-based object representations. *ACM Computing Surveys*, 36(2), 2004.

[SJLL00] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *ACM SIGMOD international conference on Management of data*, pages 331–342, 2000.

[SM01] J. Saglio and J. Moreira. Oporto: A realistic scenario for moving objects. *GeoInformatica*, 5(1), March 2001.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The r^+-tree: A dynamic index for multi-dimensional objects. *Very Large Data Bases*, (13), 1987.

[Tra05] European Commission's Transport. Galileo european satellite navigation system. http://europa.eu.int/comm/dgs/energy_transport/galileo/index_en.htm, 2005.

[TSN99] Y. Theodoridis, J. R. O. Silva., and M. A. Nascimento. On the generation of spatiotemporal datasets. *Intational Symposium on Spatial Databases*, 6, July 1999.

[Wea05] Unisys Weather. Atlantic hurricane data. http://weather.unisys.com/hurricane/index.html, 2005.

[YPK05] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *International Conference on Data Engineering*, number 21, 2005.

A Distance in MySQL

In Listing 4 the algorithm for calculating the distance from a point to a polygon is outlined.

```
1  function OurDistance(p, geo)
2    if (IsWithin(p, geo) return 0
3    d←∞
4    foreach(l ∈ geo)
5      d′←DistanceToLineSegment(p, l)
6      if (d′ ≤ d) d←d′
7    return d
```

Listing 4: The distance function in pseudo-code

On Line 1 in Listing 4 the input is defined such that it takes in a point, p, and a polygon, *geo*, which is an ordered set of lines. On Line 2 it is tested whether p is within *geo* and if so, 0 is returned. d is the distance from p to *geo*. d is initialized to ∞ on Line 3. Lines 4–6 loops over every lines, l in *geo*. The distance from p to l is calculated on Line 5 and if this distance is shorter than the current partial result d. The distance is saved in d on Line 6. The implementation for the OurDistance and DistanceToLineSegment can be seen in Listing 5.

```
1  DELIMITER '//';
2  -- inspired by http://www.vb-helper.com/
       howto_distance_point_to_line.html
3  DROP FUNCTION IF EXISTS DistanceToLineSegment//
4  CREATE FUNCTION DistanceToLineSegment(px DOUBLE, py
       DOUBLE, x1 DOUBLE, y1 DOUBLE, x2 DOUBLE, y2
       DOUBLE) RETURNS DOUBLE
5  BEGIN
6        DECLARE dx DOUBLE; DECLARE dy DOUBLE;
             DECLARE t DOUBLE;
7        DECLARE nearx DOUBLE; DECLARE neary DOUBLE
             ;
8
9        SET dx = x2 - x1;
10       SET dy = y2 - y1;
11       IF( dx=0 AND dy=0) THEN
12             -- It's a point not a line segment
13             SET dx = px - x1;
14             SET dy = py - y1;
15             SET nearx = x1;
16             SET neary = y1;
17             RETURN SQRT(dx * dx + dy * dy);
18       END IF;
19
20       SET t = ((px - x1) * dx + (py - y1) * dy) / (dx * dx +
             dy * dy);
21
22       IF (t < 0) THEN
23             SET dx = px - x1;
24             SET dy = py - y1;
25             SET nearx = x1;
26             SET neary = y1;
27       ELSEIF (t > 1) THEN
28             SET dx = px - x2;
29             SET dy = py - y2;
30             SET nearx = x2;
31             SET neary = y2;
32       ELSE
33             SET nearx = x1 + t * dx;
34             SET neary = y1 + t * dy;
35             SET dx = px - nearx;
36             SET dy = py - neary;
37       END IF;
38
39       RETURN SQRT(dx * dx + dy * dy);
40 END//
41
42
43 DROP FUNCTION IF EXISTS OurDistance//
44 CREATE FUNCTION OurDistance (px DOUBLE, py DOUBLE,
       geo POLYGON) RETURNS TEXT
45 BEGIN
46       DECLARE geostring TEXT;
47       DECLARE pairstring TEXT;
48       DECLARE distance DOUBLE;
49       DECLARE testdistance DOUBLE;
50       DECLARE x1 DOUBLE; DECLARE y1 DOUBLE;
             DECLARE x2 DOUBLE; DECLARE y2 DOUBLE
             ;
51       DECLARE commapos INT;
52       DECLARE spacepos INT;
53
54       IF(WITHIN(GEOMFROMTEXT(CONCAT('POLYGON((',
             px, ' ', py, '))')),geo)) THEN
55             RETURN 0;
56       END IF;
57
58       SET geostring = ASTEXT(geo);
59       SET geostring = SUBSTRING(geostring, LENGTH('
             POLYGON((')+1, LENGTH(geostring) - ( LENGTH('
             POLYGON((') + LENGTH('(('')));
60
61       SET commapos = LOCATE(',', geostring);
62       SET pairstring = LEFT(geostring, commapos-1);
63       SET geostring = SUBSTRING(geostring, commapos+1,
             LENGTH(geostring)-commapos);
64       SET spacepos = LOCATE(' ', pairstring);
65       SET x1 = LEFT(pairstring, spacepos - 1);
66       SET y1 = SUBSTRING(pairstring, spacepos, LENGTH(
             pairstring)-(spacepos-1));
67
68       SET distance = DistanceToLineSegment(px, py, x1, y1,
             x1, y1); -- default to something that is true
69
70       WHILE (INSTR(geostring, " ")) DO
71             SET commapos = LOCATE(',', geostring);
72             SET pairstring = LEFT(geostring, commapos-1);
73             IF(commapos<1) THEN
74                   SET pairstring=geostring;
75                   SET geostring = "";
76             ELSE
77                   SET geostring = SUBSTRING(geostring,
                         commapos+1, LENGTH(geostring)
                         -commapos);
78             END IF;
79             SET spacepos = LOCATE(' ', pairstring);
80             SET x2 = LEFT(pairstring, spacepos - 1);
81             SET y2 = SUBSTRING(pairstring, spacepos,
                    LENGTH(pairstring)-(spacepos-1));
82             SET testdistance = DistanceToLineSegment(px,
                    py, x1, y1, x2, y2);
83
84             IF(testdistance < distance) THEN
85                   SET distance = testdistance;
86             END IF;
87             SET x1 = x2;
88             SET y1 = y2;
89       END WHILE;
90
91       RETURN distance;
92 END//
93 DELIMITER ';'//
```

Listing 5: The OurDistance and DistanceToLineSegment functions in SQL