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

Title:

Generating and Indexing Multi-Dimensional Moving Regions in Spatio-Temporal Databases

Project period:
Dat6, Feb. 1st 2006 - Jun. 9th, 2006

Project group:
E4-110

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

Supervisor:
Kristian Torp

Copies: 8

Page count: 39

Abstract:

The number of modifications is typically orders of magnitude higher than the number of queries on a database that stores moving regions. However, the predominant index structure for moving regions is the R^+-tree and this structure handles spatial queries better than modifications. To address this problem a new indexing technique, called Layered Shifted Space-filling curves (LSS), is presented. LSS is optimized for modifications. It uses shifted layers in order to assign only a single index value to each region. Shifting layers guarantees that moving regions of a certain size always fit on a particular layer. A problem with canonical use of space-filling curves is that it requires prior knowledge of the data to be indexed. A self maintenance method for the LSS technique is proposed which makes LSS work without a prior knowledge of the data, e.g, world size. A performance study using the Oracle DBMS compares LSS to Oracle's R^+-trees and existing space-filling curve approaches. This study shows that LSS, compared to Oracle's R^+-trees, is up to 35 times faster for modifications at the expense of being up to 11 times slower for queries. Synthetic data is often preferred for performance testing as it is easy to control the conditions. This paper also proposes both a data and workload generator for generating n-dimensional moving objects. The data generator uses statistical distributions for moving and resizing the objects. When testing an index it is often desired to test a mix of modifications and queries. Therefore, this paper also provides a workload generator which produces a mix of inserts, updates, deletes, and spatial queries. A performance study shows that STDW is able to produce 27,000 object snapshots per second on a standard PC.

Generating and Indexing Multi-Dimensional Moving Regions in Spatio-Temporal Databases

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

June 9, 2006

Preface

This project is split into two different papers: Layered Space-Filling Curves for Indexing Moving Regions *and* STDW: A Multi-Dimensional Spatio-Temporal Data and Workload Generator. *The first paper is a further development our dat5 paper:* Indexing Moving Objects Using Layered Space-Filling Curves. *The following sections have been changed: In Section 2, Related Work, all work regarding data generation has been moved to the second paper. In Background, Section 3; Section 3.2, 3D Space-Filling Curves, has been added. Section 3.3, Query Types, has been modified to describe three dimensional behavior. Section 4.2, 3D Layers, has been added. Section 5.4, 3D Shifting, has been added describing problems when shifting in three dimensions. Self Maintenance, Section 6, is a new contribution in this paper. Spatial Queries, Section 7, has been modified for using the self maintenance model. Furthermore, the performance study in Section 8 has been completely rewritten using the Oracle DBMS. The second paper is new and original material.*

Jens Frøkjær | *Palle Bertram Hansen*

Ivan Vigsø Sand Larsen | *Tom Oddershede*

Layered Space-Filling Curves for Indexing Moving Regions

In the paper *Layered Space-Filling Curves for Indexing Moving Regions* a technique for indexing moving regions called LSS is presented. The LSS technique is based on B^+-trees and space-filling curves and can be built on top of an existing DBMS. LSS handles modifications better than the predominant spatial index structure, the R^+-tree, at the expense of slower spatial queries. When dealing with moving regions the number of modifications is often typical orders of magnitude larger than the number of queries, and therefore the sacrifice of slower queries can be justified.

The paper presents background knowledge for the use of space-filling curves. The canonical use of space-filling curves is one plane divided into cells, and it handles regions that extend over multiple cells by storing them multiple times. The LSS technique consists of multiple planes on top of each other with different cell sizes, which enables regions of different sizes to be stored on different layers.

The layers are shifted to guarantee that moving regions of a certain size always fit on a particular layer. Shifting minimizes the number of layers, such that spatial queries can be executed reasonably efficient. This is done by having three instances of the same layer which are displaced with respect to each other.

The technique works even without prior knowledge of the data to be indexed, when applying the self maintenance model. The model describes how to dynamically add and remove layers based on approximation of object sizes.

The performance study shows that LSS, compared to R^+-trees, is up to 35 times faster for modifications at the expense of being up to 11 times slower for queries.

STDW: A Multi-Dimensional Spatio-Temporal Data and Workload Generator

In the paper *STDW: A Multi-Dimensional Spatio-Temporal Data and Workload Generator* a data and workload generator is presented. There are already a number of data generators available. However, when testing an index structure it often involves a mix of inserts, updates and deletes. Therefore, a data generator is not adequate. Here the workload generator can be utilized.

The existing data generators all have different weak points which make them difficult to use in conjunction with a workload generator. Therefore, the paper presents both a data and workload generator, which are designed to work together. Each of them can also be used stand-alone. The workload generator can also be used in conjunction with other data generators.

The workload generator is able to add both spatial and temporal noise to data received from the data generator, in order to make the output more realistic. The workload generator is also able to generate range and k-NN queries and supply these with the results.

None of the existing data generators are able to produce data in more than two dimensions. STDW is able to produce data from one to n dimensions.

The data generator operates with two abstractions, worlds and blocked spaces, which enables the user to create an environment for the objects to reside within.

The performance shows that the generator is able to produce 27,000 object snapshots per second on a standard PC. It also shows that the time consumption is linear in the number of dimensions and objects.

Layered Space-Filling Curves for Indexing Moving Regions

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

June 9, 2006

Abstract

The number of modifications is typically orders of magnitude higher than the number of queries on a database that stores moving regions. However, the predominant index structure for moving regions is the R^+-tree and this structure handles spatial queries better than modifications. To address this problem a new indexing technique called Layered Shifted Space-filling curves (LSS) is presented. This technique is optimized for handling modifications. LSS is based on space-filling curves. It uses layers in order to assign only a single index value to each region. The layers are shifted to guarantee that moving regions of a certain size always fit on a particular layer. Shifting minimizes the number of layers, such that spatial queries can be executed reasonably efficient. A problem with canonical use of space-filling curves is that it requires prior knowledge of the data to be indexed. A self maintenance method for the LSS technique is proposed which makes LSS work without a prior knowledge of the data, e.g, world size. A performance study using the Oracle DBMS compares LSS to Oracle's R^+-trees and existing space-filling curve approaches. This study shows that LSS, compared to Oracle's R^+-trees, is up to 35 times faster for modifications at the expense of being up to 11 times slower for queries.

1 Introduction

With the growth in 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 shoals of fish are moving, and telephone companies knowing at all times where their customers' mobile phones are located. These application areas have in common that the location of the objects must be transmitted to the relevant service.

One could imagine that the number of modifications is orders of magnitude larger than the number of queries, and the ability to handle large numbers of moving objects will be of even greater importance in the future as new location-aware services arise. The European Union is about to launch its GPS counterpart, the Galileo system [21], which will provide more devices with location information.

The predominant index structure for regions is the R^+-tree. However, it is expensive to maintain under heavy updating [8]. This problem can be solved by using a B^+-tree, which performs better than R^+-trees when heavy modification occurs [8]. The B^+-tree only supports indexing points in a one-dimensional space. If regions are to be indexed by a B^+-tree, a mapping from two or more dimensions to one is needed [11].

A space-filling curve is a method that is well suited for reducing the dimensionality. The most popular is the Hilbert space-filling curve that is widely believed to have the best clustering properties [10]. The clustering properties of a space-filling curve indicate how well it preserves the proximity

of objects.

Canonical use of space-filling curves handles regions that extend over multiple cells by storing it multiple times. This may cause more I/O, as the regions potentially are stored multiple times in the database [3].

This paper introduces a layered technique for using space-filling curves that can be implemented on top of an existing DBMS. This technique is called *Layered Shifted Space-filling curves* (LSS). This technique optimizes the modification speed of the moving object in the DBMS as only one index value is assigned to each object in order to store the object only once in the database. The main focus of this paper is on indexing moving regions in a two-dimensional (2D) space.

This paper is structured as follows. Sections 2 and 3 present related work and background knowledge, respectively. A technique for indexing moving regions using layered space-filling curves is presented in Section 4. Reducing the number of layers by shifting is described in Section 5. Section 6 introduces a self maintenance method for the LSS technique. Algorithms for the two spatial queries, range and k-NN, are the topics of Section 7. Section 8 presents a performance study of LSS and R^+-trees. Finally, Section 9 concludes the paper.

2 Related Work

This section is structured as follows. First, related work on the R-tree family is presented. Second, work on indexing spatial objects using space-filling curves is presented.

Indexing moving objects can be done in many different ways. Popular and efficient indexing structures are the R-tree [7] and the R^+-tree [19]. Although they are applicable in many different scenarios, there are problems with these indexing structures. One particular problem is that several directory rectangles may cover the same area, which can lead to additional path traversals [2]. The R*-tree [2] was introduced as a method for coping with these problems.

The TPR-tree [17] is based on the R*-tree. The TPR-tree is very efficient for querying the current and predicted future positions of moving points. The paper also provides a workload generator that simulates points moving along routes between destinations, and generates both modification and retrieval queries. LSS differs from the work in [17] as it proposes a technique for indexing the current positions of moving regions. This is practical because knowledge about previous positions and trajectories are not needed.

As described in [11], B-trees have proven to be a very efficient index for many different types of data. Although the B-trees are intended for indexing points in a 1D world it can be used to index multi-dimensional data by using dimensionality-reducing techniques. The work in this paper is similar to the work in [11] on how to index regions. Here space is partitioned into cells of uniform size and each cell is given a space-filling number. This number could for example be assigned using the Hilbert space-filling curve [5]. A region may not be able to fit into a single cell. Therefore, one region may be assigned multiple index values. In general, multiple index values complicate both the indexing and querying of regions [3].

To prevent a region from having multiple index values, a layered or hierarchical approach is used in this paper. This is similar to [23] that uses a hierarchical main-memory structure to answer spatial queries. However, LSS is based on B^+-trees and space-filling curves and is not a main-memory technique.

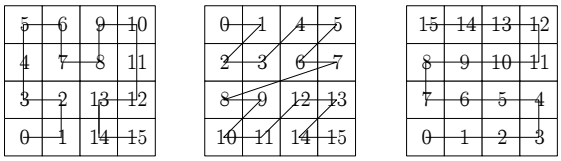
In [3] a layered technique based on the Z-ordering is presented. With this technique regions are only assigned a single index value. This is similar to LSS. However, LSS differs from the technique in [3] by using shifted layers, where it is guaranteed that regions of a given size always fit. LSS also differs by having layers with divisions that are not 2^n, which the Hilbert space-filling curve requires [16]. The use of not 2^n divisions can minimize the number of layers, and therefore fewer cells are needed to be searched when querying.

3 Background

In this section the background knowledge on space-filling curves and problems with the canonical use of space-filling curves for indexing regions [3] are presented. Finally, an example is introduced along with the query types that will be focused on in this paper.

3.1 Space-Filling Curves

To index 2D objects using a B$^+$-tree a reduction in the number of dimensions is needed. This reduction can be achieved by the use of space-filling curves. When the space is, e.g., 2D, the location is approximated by a single number, instead of having to save an object with a coordinate pair. This requires the original space to be divided into a number of cells. Each of these cells is given a *space-filling number* according to the space-filling curve. The world is divided using a static grid that has the following properties. (a) It ensures all cells are of equal size. (b) There is no overlap between cells, and (c) the whole world is guaranteed to be covered by the grid. Examples of space-filling curves can be seen in Figure 1, inspired by [18].



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

Figure 1: Examples of space-filling curves

A space-filling curve is designed to preserve the clustering properties when mapping to one dimension. According to [10] it is widely believed that the Hilbert space-filling curve has the best clustering properties.

When dealing with points only, the canonical approach in Figure 1 is very useful. But when using space-filling curves for regions 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 regions a mapping of 2D objects in a 2D space into a single number is wanted.
- A region might not be able to fit into exactly one cell in the grid, e.g., in Figure 1. Therefore, a single region may be required to have several different *index values*, which requires more I/O when modifying the regions [3].

To avoid giving a single region several index values the cell size can be adjusted according to the largest region in the population. Although the cell size is larger than any region, it still cannot be guaranteed that a region would not intersect cell borders. Furthermore, when dealing with regions that change size, it may be impossible to predict their maximum size.

3.2 3D Space-Filling Curves

In the previous section 2D usage of space-filling curves was presented. However, many space-filling curves can be used in more dimensions, e.g., the Hilbert space-filling curve can be used in n dimensions [9]. Figure 2 illustrates the Hilbert space-filling curve in three dimensions.

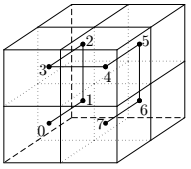
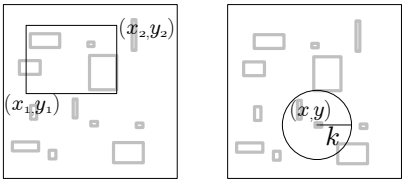


Figure 2: Hilbert space-filling curve in 3D

3.3 Query Types

Two important spatial queries are the *range* and *k nearest neighbor* (k-NN) queries. A range query [13] selects all objects within a certain rectangular area. A query of this type could be to ask "which taxis are within city limits?". A k-NN query [15] selects the k nearest objects to a given point. An example could be when a customer requests a taxi; the service employee would inquire "which are the 10 nearest free taxis to this customer?". Examples of these query types are illustrated in Figure 3.



(a) Range (b) k-NN

Figure 3: Examples of query types

In Figure 3(a) the black rectangle illustrates the range query. In Figure 3(b) the circle illustrates the

result set of the k-NN query. That is the smallest circle with center (x, y) which intersects k objects.

For the 3D case the range query is extended by a z-dimension, so the query area is a *cuboid*, that is a 3D rectangle. With this range query the approach is the same as above. All objects that intersect with the cuboid are contained in the result set.

For k-NN in 3D, a query point (x, y, z) is selected. Then the k nearest neighbors are returned, with respect to the distance to the nearest neighbors in any dimension. The result set can be visualized as a sphere.

3.4 Index Measurement

In order to measure the quality of an index technique, two metrics termed *dead space* and *index selectivity* are used in this paper. In this section only the 2D case is examined.

When a region is smaller than the cell size, dead space occurs. The area not occupied by an object in the cell, in which it resides, is called dead space. Dead space may lead to *false positives*. A false positive is an object that is retrieved by the index to answer a query but not part of the result set. 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 in Equations 1 and 2. It shows that for each object in the database the object size is subtracted from the cell size and then divided by the cell size. The variable db represents the set of all objects in the database and $|db|$ is the number of objects.

$$ds = \sum_{obj \in db} \frac{obj_{cellSize} - obj_{objectSize}}{obj_{cellSize}} \quad (1)$$

$$\text{dead space} = \frac{ds}{|db|} \quad (2)$$

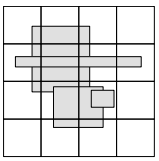
Index selectivity is also a number between 0 and 1. It is the average number of result set candidates that must be examined, when using only the index, in a range query for a single point. That is, how many objects cannot be filtered out solely based on the index. Equations 3 and 4 show how index selectivity is calculated. Note that x and y are repeated in Equation 3 in order to make a point appear as a region.

$$is = \int_0^{w_y} \int_0^{w_x} \frac{|\text{range}_\square(x, y, x, y)|}{|db|} dxdy \quad (3)$$

$$\text{index selectivity} = \frac{is}{w_x \cdot w_y} \quad (4)$$

The variables w_x and w_y are the lengths of the sides of the world. In Equation 3 $\text{range}_\square$ is defined as the range query that returns both true and false positives from cells intersected by the rectangle used in the query. This can be visualized as a range query, where all the regions have been extended to the same size and shape of the cells in which they are contained.

A simple example is shown in Figure 4 with four regions. The numerator of Equation 3 is shown in Figure 4(b) which is based on the placement of the objects from Figure 4(a). Note that this function is not differentiable. However, it can be computed using Riemann sums [4]. Finally, the result is normalized to the size of the world in Equation 4. The index selectivity of Figure 4 is 28.125% as $is = \frac{18}{4} = 4.5$ and thereby index selectivity is $\frac{4.5}{4 \cdot 4} = 28.125\%$.



(a) Example data set

1	1	1	0
2	2	2	1
1	2	3	0
0	1	1	0

(b) $\text{Range}_\square$

Figure 4: Example of index selectivity

Lower index selectivity is better, as fewer false positives are returned from the index.

4 Layered Space-Filling Curves

As a first step in trying to overcome the problems with the canonical use of space-filling curves outlined in Section 3.1, *layered space-filling curves* are introduced in this section.

4.1 A Layered Approach

The LSS indexing technique described in this paper assigns only one index value to each region. LSS

makes modifications faster compared to the canonical Hilbert as the region only needs to be stored once in the database. An example of a table, **tab**, is shown in Figure 5. The **id** column is the primary key, the **geometry** column is the actual spatial region, and the column **iv** (index value), which is the space-filling number of the cell in which the object resides. The last column, **data**, symbolizes other columns.

id	geometry	iv	data
1	$((0.2, 0.3), (0.4, 0.4))$	0	...
2	$((0.7, 0.3), (0.9, 0.4))$	3	...
3	$((0.6, 0.7), (0.6, 0.9))$	2	...

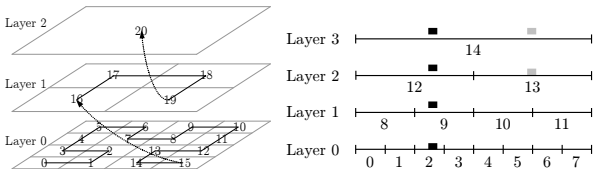
Figure 5: Example of table **tab**

Instead of looking at the world as only one plane divided into cells, this paper looks at the world as a set of planes on top of each other. Each of these planes is called a *layer*. They have the following properties.

- Layers are numbered bottom-up starting with 0
- Cells are numbered using a space-filling curve
- The *top layer* contains only a single cell
- A layer always has more cells than any layer above it
- The smallest space-filling number is 0 and the largest is the number of cells minus one
- Space-filling numbers on one layer are always larger than any number on any layer below

Note that the last two bullets ensure that cell numbers are unique across layers. An example with three layers is shown in Figure 6(a). The layers use the 2^n *divisions* $[4, 2, 1]$. This means that the index is divided into 4×4 cells at Layer 0, 2×2 at Layer 1, and 1×1 at Layer 2. The 2^n division enables use of the Hilbert space-filling curve. In the figure the arrows illustrate where the numbering ends at one layer and where it begins at the next layer.

When an object is inserted, it is pushed through the layers top-down, until it cannot fit into a cell, i.e., touches a cell border, and it is then stored on the layer above. When updating an object, the same approach is used, i.e., the object is pushed through the layers and the index value of the object is set to the space-filling number of the new cell. When deleting, the object is simply removed from the database.



(a) A 2D LSS with divisions $[4, 2, 1]$

(b) A 1D LSS with divisions $[8, 4, 2, 1]$

Figure 6: Layered Hilbert space-filling curves

Figure 6(b) is an example of the model in 1D where the cells are numbered sequentially. Note the example is only in one dimension for the sake of simplicity. The black and gray boxes symbolize two regions (intervals) and show on which layers they can fit into a cell. When a region touches a line, it indicates that this is where the region is actually indexed. The figure shows that the black region is pushed down to Layer 0 and given the index value 2. The gray region cannot fit into the cell on Layer 1 as it will touch the cell border between 10 and 11. Therefore, it is stopped at Layer 2 and given the index value 13. It is important that the objects are pushed to the lowest possible layer in order to achieve index selectivity and decrease dead space. Therefore, it would have been better if the gray region could have been pushed all the way down to Layer 0.

4.2 3D Layers

In the previous section, 1D and 2D cases were presented. Going from 2D to 3D is straight-forward. At each layer the third dimension (the z-axis) is added with the same division as on the other axes. Then the world is a cuboid, divided into smaller cuboidical cells. Figure 7 shows a 3D LSS with the divisions $[4, 2, 1]$.

When indexing objects, the same approach is used as in Section 4.1. If an object touches a cell border (in the 3D case a cuboid side) the object must be saved on the layer above.

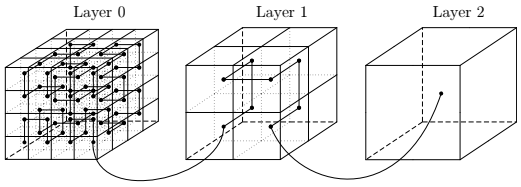


Figure 7: A 3D LSS with divisions $[4, 2, 1]$

4.3 Layer Expansion

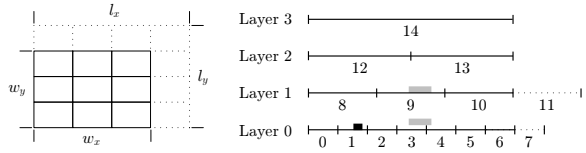
The Hilbert space-filling curve and other space-filling curves require that the layer must be divided into 2^n cells in each dimension where $n \in \mathbb{Z}^+$. If there are not 2^n divisions the layer must be expanded in order to apply the space-filling curve. The number of divisions of the current layer is called d, and the lengths of the sides of the world are called w_x and w_y. The lengths of the sides of the new layer are denoted l_x and l_y

$$d' = 2^{\lceil \log_2(d) \rceil} \tag{5}$$

Now d' is defined as the smallest 2^n number larger than or equal to d as shown in Equation 5. The layer can be expanded with the missing cells by applying Equation 6. Note that Equation 6 only calculates w_x. The same must be done for w_y.

$$l_x = w_x + \frac{w_x}{d} \cdot (d' - d) \tag{6}$$

Figure 8(a) 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 are only used for applying the space-filling curve.



(a) An example of layer expansion

(b) Indexing a region using the divisions $[7, 3, 2, 1]$

Figure 8: Expanding the world

When introducing a layer with divisions not equal to 2^n it is not suitable to use the top-down insert strategy described in Section 4.1, where an object is pushed down through the layers until it hits a cell border. Instead, 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 a cell. Figure 8(b) shows that the black is stored on Layer 0 and given the index value 2 as this cell is the first from the bottom-up it can fit. The gray region fits on Layer 1 and Layer 3. Using the bottom-up strategy it will be stored on Layer 1 and given the index value 9. Note that Layer 0 originally had seven divisions, but as this is not a 2^n number, the layer is expanded into eight divisions. The same procedure follows with Layer 1.

4.4 Non-Overlapping Grids

Looking at Figure 6(b), if a region is positioned in the middle of the world it would have to be pushed all the way up to Layer 3, resulting in poor selectivity and increased dead space. To overcome this problem, cell borders are not shared across the layers.

A strategy for finding non-sharing cell borders is to select a set of divisions, D, that are relative prime as shown in Equation 7 where $\gcd(a, b)$ is the greatest common divisor of a and b.

$$\forall d, e \in D : d \neq e \Rightarrow \gcd(d, e) = 1 \tag{7}$$

Relative primes are used because if a region touches a cell border at one layer, this border will not be in the same location at any other layer. This makes it more likely to find a cell at a low layer where the indexed regions do not touch any cell border.

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 symbolize places where divisions are possible.

In the following theorem and proof, lcm(a,b) is the least common multiple of a and b. $a|b$ means that a is a divisor in b, i.e., $\frac{b}{a} \in \mathbb{N}$.

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 $\gcd(d, e) = 1$ it is known 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 $d \neq e$ then $\frac{x}{d} \neq \frac{x}{e}$.
- If $x > y$ and $d < e$ then $\frac{x}{d} > \frac{y}{e}$, 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 $x \cdot e = y \cdot d$, it must hold that $(x \cdot e)|(y \cdot d)$. It is known that if $\gcd(p, q) = 1 \Rightarrow \text{lcm}(p, q) = p \cdot q$. 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.

□

When looking at Figure 6(b) it can be seen that two regions of the same size are placed on two different layers. Therefore, the focus of the next section is to ensure the regions are stored on the lowest possible layer.

5 Shifted Layers

A problem with the indexing technique described in the previous section is that a single-celled top layer is needed to ensure that all regions can be indexed even if the largest region is much smaller than the world. This cell, covering the whole world, is needed because no other layer gives any guarantee about the size of the regions that can be indexed. The top layer has a poor selectivity and dead space is increased dramatically. One way of ensuring that only few regions are indexed at the top layer is to have many layers. This makes it possible to reduce dead space when placing moving regions. However, having many layers results in poor query performance, because a query has to look at all layers. These problems are addressed in this section. Sections 5.1, 5.2, and 5.3 focuses on 2D. Section 5.4 generalizes some of the results from Sections 5.2 and 5.3 to three or more dimensions.

5.1 Shifting a Layer

A *shifted layer* is a layer that guarantees that regions of a given maximum size always fit on this layer. This is beneficial, e.g., when knowing the typical size of regions. A 2D shifted layer has the following properties.

- It consists of three sub-layers that are shifted with respect to each other
- The numbering at each sub-layer is unique

Figure 9 shows a shifted layer with three divisions. The bottom layer illustrates an expansion of the world as shown in Figure 8(a). The subscript 3 symbolizes a shifted layer.

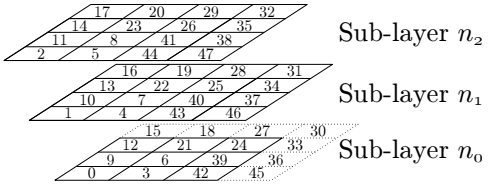


Figure 9: 2D shifted layer ($d = 3_3$)

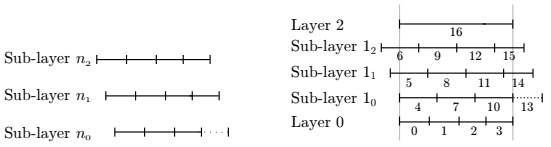
Any layer in the technique described in Section 4 can be replaced by a shifted layer. If a maximum size of regions is known then the top layer can be replaced by a shifted layer. Otherwise, the single-celled top layer is still needed.

A shifted layer that guarantees regions of up to o_x in the length on the x-dimensions and up to o_y in the length on the y-dimension can be created. A division for such a layer is found using Equation 8 where w_x and w_y are the lengths of the sides of the world. The d_x and d_y are the numbers of divisions on the x- and the y-dimensions respectively if the region 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 a region of a certain size. This will be elaborated in Section 5.4. Finally, the divisions for the layer are 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 ceil minus one is used.

$$d_x = \frac{w_x}{3 \cdot o_x} \quad d_y = \frac{w_y}{3 \cdot o_y} \quad d = \lceil \min(d_x, d_y) - 1 \rceil \tag{8}$$

When shifting, the layer is copied twice so there are three identical instances of the layer. This is called shifting three times. The second and the

third sub-layers are shifted $\frac{w_x}{3d}$ and $\frac{2w_x}{3d}$ to the left, respectively. They are also shifted $\frac{w_y}{3d}$ and $\frac{2w_y}{3d}$ down in the 2D case, respectively. Figure 10(a) shows an example in 1D of a shifted layer with three divisions (note that in 1D only two sub-layers are needed, but in 2D three sub-layers are needed). Figure 10(b) solves the problem from Figure 6(b) where two regions of equal size were placed on layers with significantly different dead space.



(a) Shifted sub-layers in 1D where $d = 3$

(b) 1D example with divisions $[4, 3_3, 1]$

Figure 10: Shifted layers

5.2 Applying Space-Filling Curves

Now space-filling curves must be applied 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 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 the shifted layer plus one.

A shifted layer consists of three identical copies of the layer and therefore the three sub-layers have 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 added one. On the last sub-layer the space-filling number is multiplied by three and two is added. This is illustrated in the 1D example in Figure 10(b) and in 2D in Figure 9. Note that the three sub-layers are considered to be one layer because the space-filling curve on the three sub-layers interleaves. This is done to take advantage of the proximity property of the space-filling curve.

5.3 Modifying Objects

When an object is indexed using a shifted layer a slightly different approach, compared to the approach in Section 4.1, is used for assigning index value to the object. It is tested if the object fits the first sub-layer, that is Sub-layer n_0 in Figure 9. If the object fits, it is assigned the space-filling number of this cell. If the object does not fit at the first sub-layer, it is shifted $\frac{w_x}{3d}$ to the right and $\frac{w_y}{3d}$ up. Recall d is the number of divisions, and w is the size of the world. If the object now fits at the first sub-layer it is assigned the original space-filling number plus one. This is equivalent to a region fitting into Sub-layer n_1 in Figure 9. If the object does not fit at the second sub-layer either, it is again shifted $\frac{w_x}{3d}$ to the right and $\frac{w_y}{3d}$ up. If the object now fits at the first sub-layer it is assigned the original space-filling number plus two. This is equivalent to a region fitting into Sub-layer n_2 in Figure 9. If the object does not fit at any of the sub-layers it is tested at the next layer. Note that by always trying to place objects at the lowest possible sub-layer, the number of different indexed cells containing objects is reduced.

When the divisions for the shifted layer are chosen from the maximum size of the regions, and all regions are of this size, the dead space for a cell is at least $\frac{8}{9}$ as the region only occupies $\frac{1}{3}$ of the cell on each dimension in the 2D case.

```
1  function getIndex(o_x1,o_y1,o_x2,o_y2,D)
2    foreach(d ∈ D)
3      if(d ∈ {n_3|n ∈ ℕ}) 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)
```

Listing 1: Calculating index value for an object

Listing 1 shows the algorithm for calculating the index value for a region. 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. Lines 2–9 iterate through the layers. Lines 3–4 examine whether the layer is shifted or not. Lines 5–9 iterate through the sub-layers. Lines 6–7 create an integer coordinate-set that represents which cell the corners are in. The variable s is added in order to move the region, in order to fit into the shifted layers. Line 8 examines whether the whole MBR is inside a cell. The Hilbert number is calculated for the object on the given sub-layer on the

given layer in Line 9.

```
1  function insert(o_id,o_x1,o_y1,o_x2,o_y2,D,data)
2    i=getIndex(o_x1,o_y1,o_x2,o_y2,D)
3    INSERT INTO tab (id, geometry, iv,data)
         VALUES (o_id, i, region(o_x1,o_y1,o_x2,o_y2),data
         )
4
5  function update(o_id,o_x1,o_y1,o_x2,o_y2,D,data)
6    i=getIndex(o_x1,o_y1,o_x2,o_y2,D)
7    UPDATE tab SET iv=i, geometry=region(o_x1,
         o_y1,o_x2,o_y2), data=data WHERE id=o_id
8
9  function delete(o_id)
10   DELETE FROM tab WHERE id=o_id
```

Listing 2: Insert, update, and delete algorithms

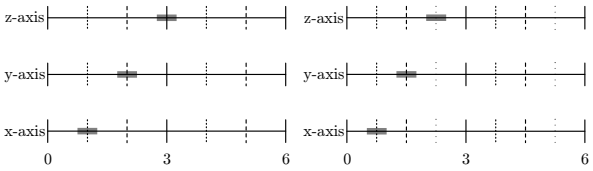
Listing 2 shows the algorithms for insert, update, and delete. It shows that with insert and update the index is calculated with the `getIndex` function in Lines 2 and 6. When index value is calculated it can be used for the modification. In Line 3 the insert is actually performed where the object is inserted into the table **tab** with the index value i. The update is performed in Line 7 where the objects in table **tab** are updated with the new position and new index value. Delete is done in Line 10 and is straight forward.

5.4 nD Shifting

When working in, e.g., 3D it is not enough to shift the layers three times. Consider a shifted layer with the division 2 and the world size 6. At Sub-layer 0 there is a cell borders at positions 0, 3, and 6. At Sub-layer 1 there is a cell border at position 2 and 5. Finally, at Sub-layer 2 there is a cell border at position 1 and 4.

Furthermore, consider a cube, $0.2 \times 0.2 \times 0.2$ in size, where the bottom left corner is located at $(0.9, 1.9, 2.9)$. This object is not able to fit on any of the sub-layers as it will overlap a cell border from each sub-layer. Figure 11(a) shows each dimension separately in a layer shifted three times. All cell borders are illustrated on each dimension. Cell borders from different sub-layer are illustrated with different line types on each dimension. The figure shows the cube touches a cell border from different sub-layers at each dimension. Therefore, no sub-layer can be found where the cube does not touch a cell border.

For shifting to work in three dimensions, four shiftings are needed, as there would be a sub-layer where the cuboid does not touch a cell border. This



(a) Three sub-layers (b) Four sub-layers

Figure 11: Each dimension from a 3D shifted layer

is shown in Figure 11(b) where the object is able to fit on the sub-layer with the cell border illustrated with the solid lines.

Theorem 2 *In n dimensions, $n+1$ shiftings are needed to guarantee that objects with a spatial extend no larger than $\frac{c}{n+1}$ on each dimension can fit a cell with the size c in this dimension.*

Proof: Assume $c = 1$ for each dimension. Each cell on each dimension is split into $n+1$ equally sized *sub-parts*, that are exactly $\frac{1}{n+1}$. As the objects are at most $\frac{1}{n+1}$ they can only fit *over* two of these sub-parts in each dimension, that is one border between sub-parts. At most n sub-part borders are overlapped, but there are $n+1$ borders; hence one is free. This final sub-part border represents the sub-layer on which the object fits. □

When shifting four times in 3D the guaranteed object size is $\frac{1}{4}$ of the cell size. This has the consequence that the divisions must be larger in order to still guarantee the same maximum object size. This leads to worse index selectivity and dead space. In practice only $\frac{3}{3} \cdot \frac{2}{3} \cdot \frac{1}{3} = \frac{6}{27}$ of the previously guaranteed objects touches cell borders from each sub-layer. Furthermore, if the objects are only half of the guaranteed size $\frac{6}{27 \cdot 8} = \frac{1}{36}$ has this problem. The hypothesis is that, index selectivity wise, it is often more feasible to let objects not fitting this layer be placed on other layers. This should be compared to doing four shifts, which increases the index selectivity to $\left(\frac{4}{3}\right)^3 \approx 2.37$ times its original value.

6 Self Maintenance

The LSS technique described in the preceding sections depends on knowledge of the data to be indexed, that is size of the objects and size of the

world. If the user does not have this knowledge the index performance may degrade as all objects potentially will end up on the top-layer or an object resides outside the world. Therefore, in order to have the same capabilities as the R^+-tree the LSS technique must also be able to perform without knowledge of the data. This section describes such a self maintenance-model.

6.1 Applying Self Maintenance

In order to apply self maintenance to the LSS technique the following aspects must be considered.

- What should the world size be?
- When should a layer be added?
- When should a layer be deleted?
- Which divisions should be used for the layers?

All these questions must be answered before the technique is self maintaining. To decide whether changes must be made to the layers, they must be examined periodically. This could for example be for every $5,000$ modifications, or at 15 minute intervals.

6.2 World Size

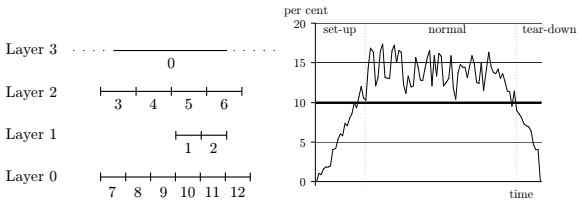
As described in Section 4 the LSS technique can index a predefined area, called the world size. However, if this area is not known in beforehand, the technique cannot be used. In order to make the LSS technique able to work, even when not knowing the world size, some changes to the layers must be made.

Instead of having a fixed world size for all layers, a fluid world size is introduced, where each layer has its own size. Initially, only a single-cell top layer is added with the size $[-\infty;\infty] \times [-\infty;\infty]$ for the 2D case. This layer is able to index all regions no matter where they are located. Other layers are added later and will have their own size. Therefore, regions are always able to be indexed, in all cases at the top layer. The layers have the following properties in the 2D case:

- The top layer has the size $[-\infty;\infty] \times [-\infty;\infty]$
- New layers are larger or equal size of all other layer, excluding the top layer
- The cell size on a layer is always larger than on the layers below

A 1D example is shown in Figure 12(a) with the divisions [6,2,4,1]. Here Layer 1 was added first (the smallest). Afterwards Layer 2 was added, and as it has larger cells than Layer 1 it is placed above Layer 1. Finally, Layer 0 was added, and as it has the smallest cell size it was placed in the bottom.

In a self maintaining LSS the *world size* is defined as the MBR of all regions ever indexed. When adding a new layer, this should have at least the size of the world.



(a) Self maintaining LSS with divisions $[6, 2, 4, 1]$ (b) Phases for a self maintaining layer

Figure 12: Examples of self maintenance

Self maintenance works by dynamically adding and deleting layers. A layer is added when it contributes significantly to the index selectivity. Layers are deleted when the number of objects on the layer is too low.

6.3 Adding a Layer

As a rule, a new layer must be added when the gain of adding this layer exceeds a given threshold. This gain is approximated using the index selectivity. The default threshold value is 50% better index selectivity. However, it is not desirable to look at all the data (sequential scan) in order to calculate the index selectivity as this potentially is very time consuming. Therefore, a new column is added to the schema from Figure 5 called **diagonal**. This column holds a scalar value representing the length of the diagonal of the region indexed. On this column a B^+-tree index is maintained so the sizes of the regions can be retrieved based solely on the index.

A new layer is marked with a time stamp in order to give it time for indexing regions. This phase is called the set-up phase as shown in Figure 12(b).

If the layer was not marked with a time stamp, it would immediately be marked for deletion in the next examination as the layer did not have time to index enough regions. After a given time interval or a number of modifications, the new layer will be considered on an equal footing as all other layers. This phase is called the normal phase in Figure 12(b). The default value is the total number of objects in the database.

```
1  function getNewLayer(D, diagonals)
2    best_s←1, best_d←1
3    foreach(d←2 to maxdiv)
4      d_pos←min_wpos, d_size←max_wsize
5      D'←D ∪ {d_3}
6      s←approximateSelectivity(D', diagonals)
7      if (s < best_s)
8        best_s←s, best_d←d
9    s←approximateSelectivity(D, diagonals)
10   if (best_s ≤ sel_threshold · s)
11     return best_d
```

Listing 3: Algorithm for adding a new layer

Listing 3 shows the algorithm for adding a new layer. In Line 1 the function takes in two parameters, D which is the divisions and *diagonals* which is a histogram [20] of the column **diagonal**. In Line 2 $best_s$, holding the best selectivity found, is initialized to 1 and $best_d$, holding the layer that must be added to achieve the selectivity $best_s$, is also initialized to 1. Lines 3–8 iterate through each candidate layer, d, from 2 to a predefined maximum number of divisions (default 100). In Line 4 d is given the size of the world and added to D in Line 5. The selectivity for the new D' is approximated in Line 6. How this approximation is done is the subject of Section 6.5. In Lines 7–8 $best_s$ and $best_d$ are updated if the current candidate layer has better selectivity than any previous candidate layer. Then in Lines 9–11 the selectivity without any candidate layer is approximated and compared to the best candidate layer. The candidate layer is returned if it gives better selectivity. The variable $sel_{threshold}$ is as default 0.5, which is 50% better selectivity.

6.4 Deleting a Layer

A layer is deleted when the number of regions on the layer is under a given percentage of the total number of regions indexed. This threshold could for example be 10% of the total number of regions indexed, otherwise selectivity is too poor. This phase is called the tear-down phase as shown in Figure 12(b).

For this approach to work the layers must be examined periodically in order to find out how many regions there are on each layer. As a B^+-tree index is maintained on the **iv** column, these queries can be answered using only the index. Furthermore, as the result does not need to be 100% correct, the query can also be executed as a non-blocking call.

When a layer contains less regions than the threshold it is marked for deletion. This is done, because if a layer is deleted immediately it would be expensive, as there are still regions on it, and moving the remaining regions may be time consuming as this requires many updates. Therefore, the layer is only *marked* for deletion which means that in future inserts and updates the layer is not considered as a potential layer for the region to be modified.

When a layer is marked for deletion, gradually there will become fewer regions on it, as they are updated. As the focus of this paper is scenarios with many modifications it is assumed that most of the regions on the layer are updated within reasonable time. When the layer is marked for deletion it is given a time stamp for when it should be empty. As default this value could be the total number of objects indexed, as then statistically most objects should have been updated.

Listing 4 shows the algorithm for marking a layer for deletion. Lines 2–6 iterate through all the layers. Line 3 examines if the layer has entered the normal phase. If this is the case then in Line 4 it examines if the number of objects is under a given threshold. If it is then the time stamp for deletion is added in Line 5 and the layer is marked for deletion in Line 6.

```
1  function getLayersForDeletion(D)
2    foreach(d ∈ D)
3      if (hasFinishedSetUp(d))
4        if ( d_count / |db| < obj_teardownThreshold)
5          d_teardown←now + timestamp_offset
6          Del←Del ∪ d
7    return Del
```

Listing 4: Algorithm for layer tear-down

When the time has passed where the layer should be empty, or the number of regions that are left on the layer is under a given threshold, the remaining regions will be removed from this layer. Either the regions are forced to a new layer where they fit the best or they are all placed on the top layer. Due to simpler calculations the second choice is used here. When the layer is completely empty, it can

be deleted, and thereby no longer considered when querying.

```
1  procedure deletingLayers(D)
2    foreach(d ∈ D)
3      if (d_teardown > now ∨ d_count/|db| < obj_deleteThreshold
           )
4        UPDATE tab SET iv=0 WHERE iv
             BETWEEN d_minIv AND d_maxIv
5        D←D\{d}
```

Listing 5: Algorithm for deleting layers

Listing 5 shows the algorithm for deleting a layer. Lines 2–5 iterate through all the layers. Line 3 examines if the time has passed for deletion or the number of objects is under a given threshold. The variable $d_{teardown}$ is the time stamp for when the layer should be deleted. $obj_{deleteThreshold}$ is the threshold for how few objects there are on the layer before it is feasible to delete the layer. In Line 4 the remaining objects on the layer are placed on the top layer (space-filling number 0). In Line 5 the layer is finally deleted.

6.5 Approximating Index Selectivity

The strategy for approximating the index selectivity is to approximate the number of objects on each layer.

The best approximation of the object is of course the object itself but as this requires a full table scan at each examination, this would be expensive. Therefore, the position of the object is disregarded as, with enough objects, according to the law of averages [22], this should even out. When approximating the shape and size of the object, without actually knowing the length of the sides, this can be done in at least three different ways; the area, the circumference, or the diagonal length. In Figure 13 a region of size 1×1 is approximated using these three methods.

Figure 13 shows which shapes a region can take when using the different methods. The lower left corner is placed in $(0, 0)$ and the upper right corer is placed on one of the three lines. The x- and y-axis in the figure represents the x- and y-sizes of the region. It can be seen from the figure that the diagonal length is the best approximation of the length of the sides because all possible regions has shorter, meaning closer to 1, sides than the other approximation methods.

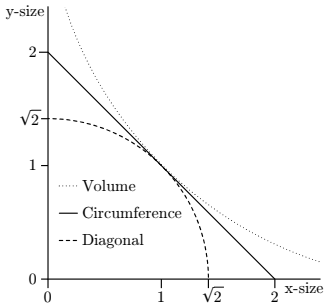


Figure 13: Approximating the region 1×1

Only a histogram of the diagonal length is needed. This speeds up calculations and with many regions it should not matter to know the exact size of each diagonal as, according to the law of averages, the roundings should even out.

In order to calculate the probability of an object fitting on a layer the probability of the object not overlapping at least one cell border from each sublayer is calculated. This is done using the diagonal length.

In the unshifted case it is much easier to calculate the probability of an object fitting on a layer compared to the shifted case. The probability for the unshifted case is given by Equation 9 where $objsize_{dim}$ is the size of the object in the given dimension, and $cellsize_{dim}$ is the size of the cells at the given layer. p is the probability of an object not overlapping a cell border on a layer.

As only the diagonal length is stored in the database, and not the actual lengths of the sides, Equation 9 can be generalized to Equation 10 where $objsize_{diagonal}$ and $cellsize_{diagonal}$ are the length of the diagonal of the object and cell on the layer respectively. It is assumed that the object has the same shape of the cell it is in as only the diagonal is stored. Therefore, it is only necessary to apply Equation 10.

$$p = \prod_{dim=1}^{dimensionality} 1 - \frac{objsize_{dim}}{cellsize_{dim}} \tag{9}$$

$$p = \left(1 - \frac{objsize_{diagonal}}{cellsize_{diagonal}}\right)^{dimensionality} \tag{10}$$

When a layer is shifted, the above approach cannot be used. In the rest of this section a layer

shifted three times is assumed. This approach works for n dimensions.

The first values to be calculated are the probabilities for not touching one, two or three cell borders from different sub-layers (p'_1, p'_2, and p'_3) in one dimension. This means that, e.g., p'_2 is the probability for an object not touching two cell borders from two different sub-layers. This is shown in Equation 12; note that $cellsize'_{diagonal}$ is $\frac{cellsize_{diagonal}}{3}$. The intuition for p'_1 in Equation 12 is the same as Equation 10 for one dimension. The intuition for p'_2 and p'_3 is that the object is shortened $\frac{1}{3}$ and $\frac{2}{3}$ of the cell diagonal and then the same procedure as from Equation 10 is used.

To simplify notation the function φ is introduced in Equation 11, it returns 0 if the input is less than 0, 1 if it is larger than 1 and the input otherwise.

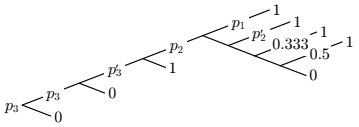
$$\varphi(x) = \begin{cases} 0 & x < 0 \\ x & 0 \leq x \leq 1 \\ 1 & 1 < x \end{cases} \tag{11}$$

$$\begin{aligned} p'_1 &= 1 - \varphi\left(\frac{objsize_{diagonal} - cellsize'_{diagonal} \cdot 0}{cellsize'_{diagonal}}\right) \\ p'_2 &= 1 - \varphi\left(\frac{objsize_{diagonal} - cellsize'_{diagonal} \cdot 1}{cellsize'_{diagonal}}\right) \\ p'_3 &= 1 - \varphi\left(\frac{objsize_{diagonal} - cellsize'_{diagonal} \cdot 2}{cellsize'_{diagonal}}\right) \end{aligned} \tag{12}$$

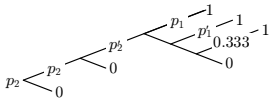
Figure 14 contains three binary probability trees [14], which is an assignment to the leftmost variable (p_i) of the probability of reaching any state labeled 1 (not overlapping one, two or three cell borders). In each binary node only one outgoing edge has its probability, q, specified. The probability for the other edge is $1 - q$.

Equation 12 only gives the result in one dimension, so in order to get the probability for n dimensions, the binary probability trees in Figure 14 must be used once for each dimension, that is a loop over dimensionality. p_1, p_2, and p_3 should be initialized to 1. In the binary probability trees, p_1, p_2, and p_3 refer to the probability from *after* the previous loop. This can easily be implemented by calculating p_3 first, p_2 second, and p_1 last, as p_2 only refers to p_2 itself and p_1. Furthermore, p_1 only refers to itself.

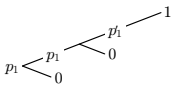
After applying, e.g., Figure 14(b) once for each dimension p_2 is the probability of an object not touching cell borders from two different sub-layers. The intuition for Figure 14(b) is (from left to right); first of all, if the object has already touched two different cell borders from different sub-layers in a previous run ($1 - p_2$), then the result is 0. Otherwise, if the object touches two cell borders in the current dimension ($1 - p'_2$), then the result must also be 0. At this point the fact is that at most one cell border is touched in the current dimension. So if no cell borders were touched in a previous run (p_1), then at most one cell border is touched. On the other hand if one cell border has been touched in a previous run ($1 - p_1$), but no more are touched in this dimension (p'_1), then only one cell border is touched. The final case is that one cell border was touched in the previous run ($1 - p_1$) and one cell border also is touched in this dimension ($1 - p'_1$). Now there is $\frac{1}{3}$ probability that it is the same cell border that has been touched and thereby still only one cell border has been touched. On the other hand, there is $\frac{2}{3}$ probability that it is not the same cell border, and thereby two different cell borders have been touched and the result should be 0.



(a) Calculating p_3



(b) Calculating p_2



(c) Calculating p_1

Figure 14: Calculating p_3, p_2 and p_1

This sums down to Listing 6. Line 1 takes in the layers, D, and the histogram of diagonals, *diagonals*. Lines 2–18 iterate over each entry, *dia*, in the histogram. *obj*, which represents the percentage of times the object does not fit on a lower layer, is initialized to 1 in Line 3. Lines 4–18 iterate through each layer where Lines 6–13 are only executed if the layer in question is shifted else Line 15 is executed. Lines 6–8 is in direct reference to Equation 12. Line 9 initializes p_1, p_2, and p_3 to 1 as described earlier. Lines 11–13 are iterated once for each dimension, this is a deduction from Figure 14. Line 15 is from Equation 10. Line 16 decreases the probability p_3 as only objects that are positioned

```
1  function approximateSelectivity(D, diagonals)
2    foreach(dia ∈ diagonals)
3      obj←1
4      foreach(d ∈ D)
5        if (d ∈ {n_3 | n ∈ ℕ}) //d is shifted
6          p'_1←1 − φ( (dia_length − d_length·0) / d_length )
7          p'_2←1 − φ( (dia_length − d_length·1) / d_length )
8          p'_3←1 − φ( (dia_length − d_length·2) / d_length )
9          p_1←1, p_2←1, p_3←1
10         foreach(dimension)
11           p_3←p_3 · p'_3 · p_2 · p_1 + p_3 · p'_3 · p_2 · (1 − p_1) · p'_2
                 + p_3 · p'_3 · p_2 · (1 − p_1) · (1 − p'_2) · (2/3)
                 + p_3 · s'_3 · (1 − p_2) · p'_1
                 + p_3 · p'_3 · (1 − p_2) · (1 − p_1) · (2/3)
12           p_2←p_2 · p'_2 · p_1 + p_2 · p'_2 · (1 − p_1) · p'_1
                 + p_2 · p'_2 · (1 − p_1) · (1 − p'_1) · (1/3)
13           p_1←p_1 · p'_1
14       else //d is not shifted
15         p_3←(1 − dia_length / d_length)^dimensionality
16       p_3←p_3 · d_volume / w_volume
17       d_count←d_count + p_3 · obj · dia_count
18       obj←obj − obj · p_3
19   foreach(d ∈ D)
20     is←is + d_count / d^dimensionality · d_volume / w_volume
21   return selectivity
```

Listing 6: Approximating index selectivity

over the layer can fit this layer. This assumes uniformly distributed objects. Another approach is to make an estimate from the number of objects estimated to fit on this layer, when not adding additional layers, and the actual number of objects on this layer. This can easily be done without any overhead as the number of objects per layer is already known from Section 6.4. Line 17 increases the number of objects that is estimated to fit on this layer by the probability of the objects fitting here (p_3) multiplied by the number of objects that has this diagonal and statistically did not fit a lower layer ($obj \cdot dia_{count}$). obj is decreased in Line 18 to reflect the percentage of the objects that fits this particular layer. In Lines 19–20 the selectivity is summed for each layer, and is returned in Line 21.

7 Spatial Queries

This section presents algorithms for the widely used spatial range and k-NN queries. The algorithms use the LSS technique presented in the previous sections. LSS can be implemented on top of an existing DBMS and therefore the algorithms presented here builds SQL queries that also can be executed on existing DBMSs. This makes it possible to directly compare LSS to existing spatial indexing techniques. Note that this section focuses solely on the 2D case. Three or more dimensions are straightforward.

7.1 Range Queries

Listing 7 shows the algorithm for creating the `WHERE` clause for a range query in 2D. The remaining part of the query is straightforward to build. The function takes as input (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. Lines 2–14 iterate through all the layers. Lines 3–4 examine whether the layer is shifted or not. If the layer is shifted Lines 5–14 iterate through the three sublayers. Lines 6–7 create an integer coordinate-set that represents which cells the corners are in. The variable s is added in order to move the region according to the shifted layer as described in Section 5.3. The d_{xpos} and d_{ypos} variables are used when the self maintenance approach is used. This takes into account that the layers can be displaced with respect to each other.

```
1  function range(q_x1, q_y1, q_x2, q_y2, D)
2    foreach(d ∈ D)
3      if (d ∈ {n_3 | n ∈ ℕ}) S←[0, 1/3, 2/3]
4      else S←[0]
5      foreach(s ∈ S)
6        q'_x1←⌊(d·q_x1 − d_xpos) / d_xsize + s⌋, q'_y1←⌊(d·q_y1 − d_ypos) / d_ysize + s⌋
7        q'_x2←⌊(d·q_x2 − d_xpos) / d_xsize + s⌋, q'_y2←⌊(d·q_y2 − d_ypos) / d_ysize + 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 ∪ {"iv=o"}
17   w'_R←implode(w_R," OR ")
18   foreach(o ∈ r)
19     w_r←w_r ∪ {"iv=o"}
20   w'_r←implode(w_r," OR ")
21   t←"(w'_R) OR ((w'_r) AND intersects(region(q_x1,q_y1,
         q_x2,q_y2)))"
22   return t
```

Listing 7: Range query algorithm

Lines 8–14 iterate through cells intersected by the range. For each cell that is iterated by Lines

8–14, Line 10 calculates the Hilbert space-filling number. Lines 11–14 examine whether a cell is completely inside the specified range. If it is, the space-filling number is added to the set R otherwise it is added to the set r.

Lines 15–21 create the actual SQL statement that specifies the range query. Lines 15–20 include the index values intersected by the range and add them to w_r and w_R respectively and `OR` them together. This is done by the `implode` function in Line 17 and Line 20. The `implode` function 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 in Line 22. In Section 8 tests have been performed using the `BETWEEN` predicate when possible. This approach requires that the sets r and R are sorted.

7.2 k-NN Queries

Listing 8 shows the algorithm for the 2D k-NN query. Again the focus is on building the `WHERE` clause. The function takes as input (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 is assumed. $|db|$ 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, because uniform object distribution is assumed.

Lines 4–8 are a loop that makes a range query and keeps expanding it until it contains k objects. c is a constant used for increasing the probability of getting a correct result in each iteration. In Line 5 the variable l is increased slightly by using c. A c value of 1.05, which is an expansion of 5%, is used as default. 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 by using the function in Listing 7. Line 7

```
1  function k−NN(k, q_x, q_y, D)
2    l ← sqrt( sqrt((w_x · w_y) · k/|db|) / 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(point(q_x,q_y)) LIMIT
         k"
```

Listing 8: The algorithm for k-NN queries

examines whether a is zero and in that case it is set to 0.5 in order to prevent division by zero in Line 5.

Line 9 executes the range query with the coordinates calculated in Lines 4–8. In Line 10 the result is sorted according to the distance to the query-point. Then the kth element is selected in Line 11, and the distance to this object is calculated in Line 12. In 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. Otherwise, a new range query is executed in Line 14. The result set may be too large, but it is guaranteed to include at least the closest k objects. In Line 15 the result is sorted by distance and only the k closest objects are selected. As the result is sorted, the algorithm guarantees, as distinct from the R$^+$-tree, that the result of a k-NN query always is sorted according to distance.

8 Performance Study

This section examines how the LSS technique performs when modifying a database. Furthermore, tests are carried out using the range and k-NN queries from Section 7.

8.1 Test Setup

The tests are performed on a 3.0 GHz Intel Pentium 4 (HT) processor with 2 GB RAM. The operating system is Microsoft Windows Server 2003 Enterprise Edition running the Oracle Database 10.2

Enterprise Edition DBMS in the default configuration.

The proposed LSS technique is implemented using the Oracle DBMS and its performance is compared to (a) the Oracle spatial indexing technique based on R^+-trees [12], (b) the canonical Hilbert indexing approach from Section 3.1, where the world is a single layer divided into cells, and if a region overlaps cell borders, the region is stored for each overlapping cell with the respective space-filling numbers, (c) indexing using the self maintaining LSS, and (d) the use of no spatial indexes. The last comparison is used for finding the overhead of using a given index structure.

In a relational database there are two obvious ways of implementing the canonical Hilbert approach. The first solution is to have a single table containing all data, and if a region overlaps several cells it is stored multiple times in the table [1]. The other solution is to have two tables where data is put in one and the Hilbert index is put in the other. Tests showed that the first implementation performs the best with respect to modifications and is used in the following. For calculating all the Hilbert space-filling curves the method described in [9] is used.

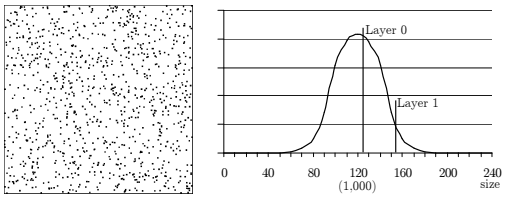
In the tests the world is $50,000 \times 50,000$. The area of the regions has a normal distribution with a mean $\mu = 125,000$ and a standard deviation $\sigma = 20,000$. This means that 1,000 regions together on average occupy 5% of the world. The distribution of the regions is uniform and illustrated in Figure 15(a). This paper has focus on moving regions so the tests are carried out with only regions in the test data. If a region is zero-sized it is still considered, and treated, as a region. The data sets used for testing are generated using the STDW data generator [6].

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

8.2 Finding Divisions

In order to use the standard LSS technique 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 *division set 1*. These numbers are chosen as the divisions are doubled on each layer. *Division set 2* is the divisions $[31, 17, 8, 5, 3, 1]$, which are all relative primes. These numbers are chosen as the divisions are close to *division set 1*.

Division set 3 is the divisions $[50_3, 41_3, 1]$. These numbers are chosen based on Equation 8, which is calculated on the smallest 50% of the regions and the smallest 96%. The numbers are slightly adjusted to ensure they are relatively primes. This is illustrated in Figure 15(b). The x-axis represents the size of the regions and the y-axis represents the probability. The reason for choosing 50% is that it is here there are most regions of the same size. The reason for choosing 96% is that it is a layer that should be able to contain almost all regions.



(a) 1,000 objects plotted

(b) Choosing divisions for *Division set 3*

Figure 15: Illustration of the test data

When comparing to the canonical Hilbert space-filling curve 8 divisions are used; this is *division set 4*. The division is chosen as tests has shown that a higher number of division is more time consuming when modifying. This comes at the expense of slower queries, but the main focus of this paper is on modifications.

The first test is to find the distribution of regions over the layers for the four division sets with multiple layers. The test is conducted with 50,000 regions in the database. *Division set 1* has the distribution $[61\%, 19\%, 10\%, 6\%, 3\%, 1\%]$, i.e., 61% of the regions are on Layer 0 (the bottom layer), 19% on Layer 1 and so on. Note that 1% of the regions are on the single-celled top layer. *Division set 2* has the distribution $[62\%, 30\%, 7\%, 0\%, 0\%, 0\%]$. Note that the top three layers contain very few regions (due to rounding the numbers do not sum up to 100). This shows that using relative primes for divisions can decrease the number of layers. *Division set 3* has the distribution $[96\%, 4\%, 0\%]$. Even though there are no regions on the top layer, it is not guaranteed that this cannot occur.

Figure 16 shows the dead space and index selec-

Division set	Dead space	Selectivity
1: $[32, 16, 8, 4, 2, 1]$	96.6%	2.780%
2: $[31, 17, 8, 5, 3, 1]$	96.5%	0.292%
3: $[50_3, 41_3, 1]$	87.6%	0.038%
4: [8] (canonical)	99.7%	1.722%

Figure 16: Dead space and index selectivity

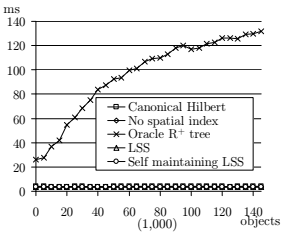
tivity as described in Section 3.1. The figure shows clearly that *division set 3* has the best dead space and index selectivity. *Division set 3* has 87.6% dead space, which is less than the upper bound described in Section 5.3. This is because the regions can fill the whole cell as the divisions are not calculated based on the maximum size. Division set 3 is used in the following for comparison and is called *LSS* and the Hilbert space-filling curve is used.

8.3 Modification

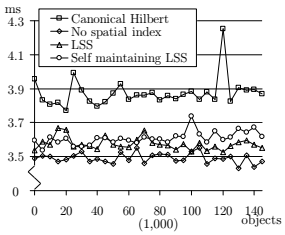
The following tests will show how inserts, updates, deletes, and a mix of the three perform.

8.3.1 Insert

The insert test is conducted on an empty database. The data set consists of 150,000 regions, which are all inserted one per transaction. Every 5,000 inserts are timed.



(a) Elapsed time per insert



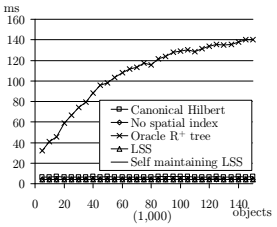
(b) Same as Figure 17(a) excluding Oracle R$^+$-tree

Figure 17: Insert

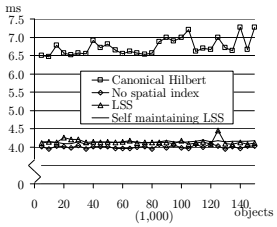
Figure 17 shows that the Oracle R$^+$-tree is approximately 2579% slower than LSS on average. In Figure 17(b), it shows that no spatial index and both LSS and self maintaining LSS are about the same. Canonical Hilbert is about 8% slower than LSS, as it occasionally saves a region multiple times. Note that in Figure 17(a) some measurements are about the same and thereby placed on top of each other, these are magnified in Figure 17(b).

8.3.2 Update

The update test is conducted on an empty database. 5,000 inserts are conducted and then 5,000 updates are performed and this is repeated. The testing is done by timing the 5,000 updates and calculating the average. The update test is done up till 150,000 regions.



(a) Elapsed time per update



(b) Same as Figure 18(a) excluding Oracle R$^+$-tree

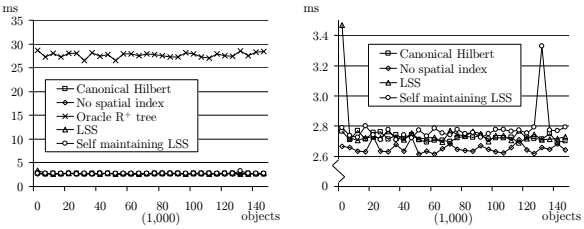
Figure 18: Update

Figure 18 shows that the Oracle R$^+$-tree is 2472% slower than LSS for updates. The canonical Hilbert (division set 4) is 62% slower than LSS as a delete must be conducted followed by a number of inserts to do the update. LSS and self maintaining LSS is about the same as with no spatial index. Note that in Figure 18(a) some measurements are about the same and thereby placed on top of each other, these are magnified in Figure 18(b).

8.3.3 Delete

The delete test is conducted on a database with 150,000 preloaded regions. Every region is deleted, one per transaction, 5,000 deletes are timed and the average is calculated. The x-axis shows how many regions there are in the database when the delete is conducted.

Figure 19 shows that the Oracle R$^+$-tree is approximately 906% slower than LSS for deletion. There is no significant difference between LSS, self maintaining LSS, canonical Hilbert, and no spatial index; these are only separated by 4%. Note that in Figure 19(a) some measurements are about the



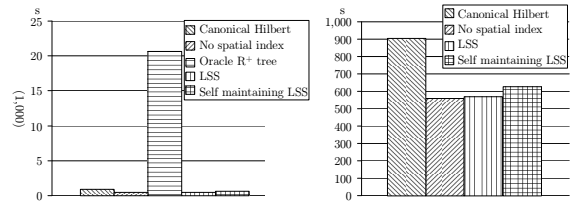
(a) Elapsed time per delete (b) Same as Figure 19(a) excluding Oracle R^+-tree

Figure 19: Delete

same and thereby placed on top of each other, these are magnified in Figure 19(b).

8.3.4 Mixed Modification

In order to test how the different indexes perform when heavily modifying the data, a mixed workload is created with 10% inserts, 80% updates, and 10% deletes. 160,000 modifications, each in a separate transaction, are executed on a database preloaded with 150,000 regions.



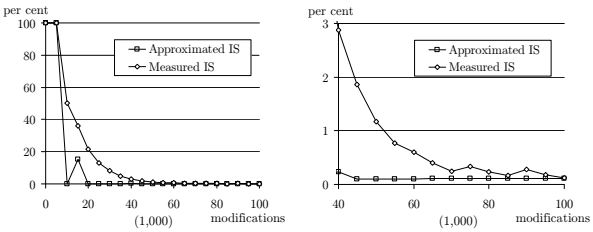
(a) Elapsed time for 160,000 modifications (b) Same as Figure 20(a) excluding Oracle R^+-tree

Figure 20: 10% Insert, 80% update, 10% delete

Figure 20 shows that the best performance is achieved with no spatial index, because it does not need to maintain an additional index. However, LSS is only 2% slower than no spatial index. The self maintaining LSS is 10% slower than LSS. The canonical Hilbert is 61% slower than LSS. This is due to the high number of updates where the canonical Hilbert is slower. The Oracle R^+-tree is 3542% slower than LSS.

8.4 Self Maintenance

This test is performed in order to examine how well a self maintaining LSS is to approximate selectivity. Figure 21 shows the approximated and the actually index selectivity from 100,000 modifications. The data set looks as follows. First 10,000 objects are inserted. Next 90,000 updates are performed on these objects. The objects are inserted with an average area of 125,000. Once an object has been updated five times it has an average area of 360,000. The objects maintain approximately this area for any further updates.



(a) Approximated and measured index selectivity (b) Same as Figure 21(a) from 40,000 to 100,000 modifications

Figure 21: Index selectivity

The test shows that the measured index selectivity comes very close to the approximated index selectivity, which indicates that the approximation method is very accurate. It ends up with an approximated index selectivity of 0.108934484 and a measured index selectivity of 0.120303732.

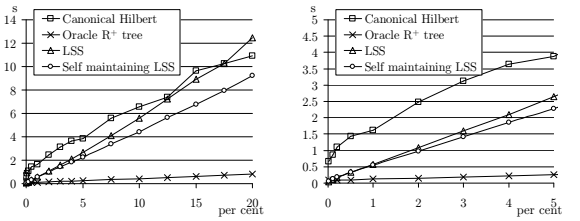
8.5 Querying

The LSS indexing technique described in this paper is designed to speed up modifications at the expense of slower spatial queries. This section examines the slowdown on range and k-NN queries on a database preloaded with 150,000 regions.

8.5.1 Range Query

The performance of range queries is found by executing 100 range queries at random positions and calculating the average response time. The percentage of the world selected by the range is varied, and is shown on the x-axis in Figure 22.

Figure 22 shows that LSS is up to 1100% slower than the Oracle R^+-tree. The self maintaining LSS is only 948% slower than the Oracle R^+-tree. This is because that the self maintaining LSS only has added one layer (division 47_3), which yields shorter queries and therefore returns the result faster than



(a) Elapsed time per range query

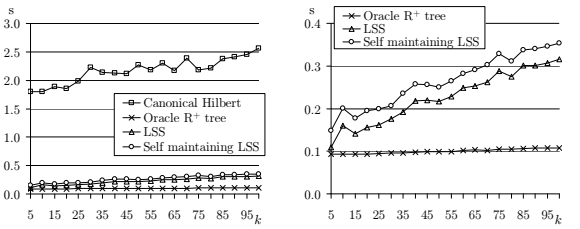
(b) Same as Figure 22(a) from 0% to 5%

Figure 22: Range queries

LSS, even though the self maintaining LSS has worse index selectivity than LSS. The canonical Hilbert performs worse than LSS because the table used for canonical Hilbert contains duplicates that must be filtered out (using `DISTINCT`). The canonical Hilbert also has worse selectivity, see Figure 16. Tests have also been performed on tables with no spatial index. These tests are not shown in Figure 22 as a query always takes approximately 30 seconds (a full table scan).

8.5.2 k-NN Query

For testing k-NN queries, 100 random query points are selected. These will be used for all the tests. The number of regions to be returned (k) is changed, to see how it scales. k is shown on the x-axis in Figure 23.



(a) Elapsed time per query

(b) Same as Figure 23(a) excluding canonical Hilbert

Figure 23: k-NN queries

Figure 23 shows that LSS is 126% slower than the Oracle R$^+$-tree. This is because that it is only very small range queries that the k-NN uses. The canonical Hilbert is very slow, which is the result of poor selectivity, i.e., here few regions are returned (k is low) and there are many more regions index by a single cell for the canonical Hilbert. Self maintaining LSS is 16% slower than LSS, this is because of worse selectivity, as with the canonical Hilbert, only a very few objects must be returned. Again results for no spatial index are omitted as it always results in a full table scan.

8.6 Summary

The self maintaining LSS indexing technique is overall faster than using an R$^+$-tree if the modification/spatial query ratio is 7 or above (7 modifications for each query) if the modifications consists of 10% inserts, 80% updates, and 10% deletes and the spatial range or k-NN queries select 2% of the world or less.

9 Conclusion

This paper proposed a new indexing technique, LSS, based on layered Hilbert space-filling curves for indexing 2D moving regions. Going from 2D to 3D has been shown to be straightforward. The notion of shifted layers was introduced to reduce the number of layers. A self maintenance model was introduced in order to be able to index data without prior knowledge of size and distribution. The LSS indexing technique has been implemented using the Oracle DBMS.

Tests showed that shifted layers are very efficient for achieving better selectivity and reducing dead space. The performance study showed that the overhead for maintaining the proposed index technique is significantly lower than the R$^+$-tree and the canonical way of using the Hilbert space-filling curve. The tests also showed that there is no significant cost overhead when using the self maintenance model. The speedup comes at the expense of slower k-NN and range queries. However, when indexing moving regions the number of modifications is often orders of magnitude larger than the number of queries. It is estimated that there should be 7 modifications for each spatial query for the proposed index technique to be overall faster than using R$^+$-trees.

Future includes implementing the LSS technique in Oracle using Oracle Extensible Indexing.

Acknowledgements

We would like to thank Kristian Torp for guidance and help throughout the project period. Without his feedback this paper could not have been realized.

References

[1] 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*, vol. 181, no. 1, pages 3–15, 1997.

[2] 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*, pages 322–331, 1990.

[3] C. Böhm, G. Klump, and H. Kriegel. XZ-Ordering: A Space-Filling Curve for Objects with Spatial Extension. *Symposium on Large Spatial Databases*, vol. 1651, page 75, 1999.

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

[5] C. Faloutsos and S. Roseman. Fractals for Secondary Key Retrieval. In *ACM PODS*, pages 247–252, 1989.

[6] J. Frøkjær, P. B. Hansen, I. V. S. Larsen, and T. Oddershede. STDW: A Multi-dimensional Spatiotemporal Data and Workload Generator. June 2006. Technical Report.

[7] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *ACM SIGMOD*, pages 47–57, 1984.

[8] K. Kalpakis, J. Behnke, M. Pasad, and M. Riggs. Performance of Spatial Queries in Object-Relational Database Systems. *NASA/CESDIS Technical Report TR-00-226*, January 2000.

[9] J. K. Lawder and P. J. H. King. Querying Multi-dimensional Data Indexed Using the Hilbert Space-filling Curve. *ACM SIGMOD*, pages 19–24, 2001.

[10] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the Clustering Properties of Hilbert Space-filling Curve. *IEEE TKDE*, vol. 13, no. 1, pages 124-141, 2001.

[11] B. C. Ooi and K. Tan. B-trees: Bearing Fruits of All Kinds. In *Australasian Conference on Database Technologies*, vol. 5, pages 13–20, 2002.

[12] Oracle Spatial, Locator, and Location-Based Services. www.oracle.com/technology/products/spatial/, June 2006.

[13] 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*, pages 214–221, 1993.

[14] R. Peck, C. Olsen, and J. DeVore *Introduction to Statistics and Data Analysis*, 2. edition. Duxbury Press, page 319, 2004. ISBN 0-534-46710-5.

[15] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *ACM SIGMOD*, pages 71–79, 1995.

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

[17] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *ACM SIGMOD*, pages 331–342, 2000.

[18] H. Samet. Object-Based and Image-Based Object Representations. *ACM Computing Surveys*, vol. 36, no.2, pages 159–217, 2004.

[19] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects. *Very Large Data Bases*, pages 507–518, 1987.

[20] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*, volume 4. Mc Graw Hill, page 759, 2002. ISBN 0-07-228363-7.

[21] European Commission's Transport. Galileo: European Satellite Navigation System. europa.eu.int/comm/dgs/energy_transport/galileo/index_en.htm, 2005.

[22] Wikipedia. Law Of Averages — Wikipedia, The Free Encyclopedia. en.wikipedia.org/w/index.php?title=Law_of_averages&oldid=52243429, June 2006.

[23] X. Yu, K. Q. Pu, and N. Koudas. Monitoring K-Nearest Neighbor Queries Over Moving Objects. In *ICDE*, pages 631–642, 2005.

STDW: A Multi-Dimensional Spatio-Temporal Data and Workload Generator

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

June 9, 2006

Abstract

When developing new indexing techniques, their performance must be examined. Synthetic data is often preferred as it is easy to control the conditions. This paper proposes both a data and workload generator for generating n-dimensional data. The data generator operates with worlds and blocked spaces which enables the user to setup a realistic scenario for the objects to move within. The data generator operates with different statistical distributions which makes it possible to control, e.g., the placement, destination, and size according to the distributions. When testing an index it is often desired to test a mix of modifications and queries. Therefore, this paper also provides a workload generator which produces a mix of inserts, updates, deletes, and spatial queries. The workload generator enables use of spatial and temporal noise which makes the output more realistic. Furthermore, the workload generator is able to produce workloads from external data generators. A performance study shows that STDW is able to produce 27,000 object snapshots per second on a standard PC.

1 Introduction

As new technologies arise for obtaining the geographical positions of moving objects, the need for handling this type of data grows. This means that efficient index structures for handling moving objects are needed.

There are different kinds of moving objects which are typically referred to as moving points and moving objects with spatial extension. An example of a moving point could be a car driving on the highway. Other moving objects, such as clouds, could be modeled as having spatial extension in two or three dimensions.

When new data structures are introduced, it is necessary to investigate the performance of these. Therefore, tests must be performed on data that resembles the conditions for which the data structure is designed. These tests can be done using either real or synthetic data sets.

Real data sets are often not accessible. Furthermore, when using real data sets, it is very difficult to control single variables like size and velocity, which is often desired. Using synthetic data sets it is easy to control single variables and to generate large amounts of data in order to simulate a specific situation.

When testing an index it is often desired to test a mix of different operations, that is inserts, updates, deletes, and queries. This is often not part of a data generator [7, 10, 11]. The mix of modifications can be generated using a workload generator. This paper presents a data and workload generator called STDW. The data generator is able to produce *pure data*, which consist of exactly one modification on each object in each snapshot. The workload generator is able to produce a mix of modifications and relevant spatial queries. Temporal and spatial noise

can be added to modifications by the workload generator. The reason why STDW is split into two different generators is that sometimes pure data is preferred; then only the data generator is used. The workload generator is able to operate with external data sources and therefore, this can be used with different data generators.

The focus of this paper is to develop a fast data and workload generator that can operate with n-dimensional (nD) data. Furthermore, different parameters are presented in order to ensure complex simulations.

This paper is structured as follows. Section 2 presents related work within the area of data and workload generation. Section 3 introduces the design criteria and the architecture. The data generator is presented in Section 4. Section 5 presents the workload generator. Visualization of the generated data is presented in Section 6. Section 7 shows the performance studies, and finally Section 8 concludes the paper.

2 Related Work

This section presents related work on non-network data generators [4], i.e., in network data generators objects move along paths [2], e.g., cars driving on roads.

In accordance to [4] the main contribution to the area of non-network data generators is made by GSTD [10], G-TERD [11], and Oporto [7]. These papers present different methods for generating spatio-temporal data sets. Common to these generators is that they all produce 2D data.

GSTD and G-TERD are the ones most similar to STDW. They all use a collection of different statistical distributions to simulate different scenarios. The third generator, Oporto, differs from STDW, GSTD, and G-TERD by mainly using an attraction and repulsion model. GSTD and G-TERD share most characteristics with STDW, but STDW differs overall from the three mentioned generators by the generation of nD data and workloads with relevant spatial queries.

GSTD supports generation of both points and minimum bounding rectangles (MBR). Points are supported as MBR with no spatial extension. Objects do not interact with each other and are allowed to leave the world. The generated objects have the following properties. *Duration*, $shift$, and *resizing*. *Duration* controls how often an object is updated. $Shift$ controls object velocity. *Resizing* controls the object size. These properties can be used in conjunction with the three statistical distributions. Uniform, Gaussian, and Zipf (skewed).

GSTD supports three different techniques for handling objects about to leave the world. Radar, adjustment, and toroid. Radar means that objects leaving the world are still monitored, since they could reenter. Adjustment, if an object tries to leave the world; it is repositioned inside the world. Toroid, the world is seen as a ball, leaving the world in one side means reappearing on the other side. Like GSTD, STDW supports an adjustment approach for dealing with objects leaving the world. Unlike GSTD, STDW has the notion of restricted areas, where no objects are allowed inside. STDW produces all objects in parallel to ensure they are ordered by time. GSTD, on the other hand, generates the objects in serial to save memory.

[8] proposes an extension to GSTD that applies infrastructure. The infrastructure is supported by randomly positioned and sized rectangles which restrict object movement. Like [8], STDW also have a notion of blocked spaces. Unlike [8] the object restrictions in STDW are user controlled and allows both encapsulation and restriction of objects.

G-TERD and GSTD uses a collection of user inputs and a number of statistical distributions to simulate different scenarios. Objects in G-TERD are a collection of sub-objects which may or may not be connected. These sub-objects are allowed to change their properties like speed and position, over time, independent of one another. The size of an object is the MBR of its sub-objects. G-TERD supports interaction between objects. Objects do not bounce on one another. However, an object can be configured to avoid overlapping other objects.

The world size in G-TERD is not limited to the monitored area. G-TERD uses a scene-observer, which means that an object can be followed as from an airplane. Only the part of the world at any time covered by the observer is presented in the output.

Like G-TERD, STDW uses statistical distributions and user input to simulate scenarios. Where G-TERD produces a sequence of raster images as output, STDW can produce both text output as

well as visualize 2D and 3D in external tools in order to verify the data. STDW generates nD MBRs. With STDW it is possible to make complex simulations by using restricted areas and nesting of worlds.

In general Oporto differs from STDW, GSTD and G-TERD by being based on attraction and repulsion. It simulates both moving points and moving objects with spatial extension. It simulates a fishing scenario.

Fishing ships head for a shoal and after some time it heads towards a harbor. Plankton act as good spots for shoals. Shoals act as good spots which attract the fishing ships. Fishing ships are repelled by storms. The Oporto generator is written in 16 bit. This has the consequence that the number of objects with different ids is limited to 2^{16}. STDW does not have this limitation. Furthermore, Oporto only supports 2D objects, where STDW is able to produce nD data.

3 Design

This section presents the design criteria and architecture for both the data and workload generator.

3.1 Design Criteria

The overall criteria for the data and workload generator are the following.

- The generator should be able to produce data in one to n dimensions
- Data and workload generation should be fast
- The output should be in an easily parsed text format

The data generator outputs *pure data* which consists of s *snapshots* each containing the position and size of all o objects. A snapshot contains all object position and sizes at a specific time. o is the total number of objects in the data load. The data is pure, because is contains the exact position and size for each object in every snapshot. The data generator is able to do the following.

- Simulate different scenarios using different statistical distributions
- Setup different areas in which objects can or cannot move

The workload generator produces a *workload* which is a number of *start inserts* followed by a number of *loads* and finally followed by a number of *end queries*. A load is a collection of i inserts, u updates, d deletes, k k-NN queries [6], and r range queries [5]. Note that some of these values can be zero, for example a load could consist of only inserts. The workload generator is able to produce the following.

- A mix of modifications and queries
- Expected results for spatial queries
- Object positions with spatial noise
- Temporal noise

3.2 Architecture

The data and workload generators consist of different elements which are depicted in Figure 1. The solid lines illustrate data flow. The boxes illustrating output are emphasized, that is **pure data** and **workload**.

The data generator can be controlled by the workload generator. When using the workload generator, it is also possible to use an external data generator. Control is illustrated by the dashed lines.

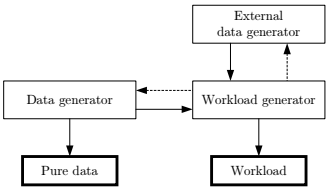


Figure 1: STDW architecture

The workload generator controls the data generator. This way, only the movements of the objects needed for the workload are generated. The workload generator is also able to process data from external data generators. In order to make this work, the external data generator must implement the STDW-workload interface. This interface requires the methods `getNewObject()` and `getNextShapshot(`*objectId*`)`.

4 Data Generation

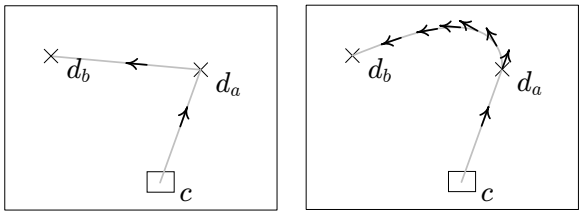
This section describes how the data generator produces the spatio-temporal data. The data generator operates with three kinds of nD entities. *Cuboids*, *worlds*, and *blocked spaces*.

4.1 Cuboids

Moving objects are called cuboids. Each cuboid has a set of properties which can be manipulated throughout the cuboid's lifetime. The position of a cuboid is its center. The cuboid can change size according to its growth rate. Collision detection between cuboids would compromise the design criteria of high performance.

To make a cuboid able to move, it is given a *destination*, a *velocity*, and a *heading*. The cuboid will try to reach its destination. When a cuboid reaches its destination it is simply given a new one. The velocity is the speed which the cuboid is currently traveling. Finally, the heading is the direction in which the cuboid is currently traveling.

In Figure 2, a cuboid, c, reaches its destination, d_a. After it has reached d_a, it is assigned a new destination, d_b. In the figure, the heading of a cuboid is shown as a black arrow. The grey line illustrates the path on which the cuboid travels. In order to make a realistic movement pattern, it is possible for a cuboid to have a heading that does not point directly towards its destination. In Figure 2(a) the cuboid always travels directly towards its destination. In Figure 2(b) the cuboid has a heading which does not at all times point directly towards its destination. This allows the cuboid to travel off the linear course towards its destination.



(a) Straight line movement (b) Cuboid with heading

Figure 2: Movement towards new destination

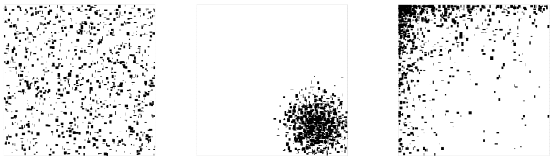
Formally, the heading is defined in Equation 1, where $\hat{\vec{a}}$ is the unit vector of $\vec{a}$, calculated $\frac{\vec{a}}{|\vec{a}|}$. $\overrightarrow{heading}$ is the new heading where $\overrightarrow{heading'}$ is the current. Pos' is the coordinate of the current position, and $Dest$ is the coordinate of the destination. $t_r \in [0; 1]$ is the turn rate. A larger value results in a tighter turn, i.e., if $t_r = 1$ it would result in the movement from Figure 2(a).

$$\overrightarrow{heading} = \hat{\overrightarrow{Pos'Dest}} \cdot t_r + \hat{\overrightarrow{heading'}} \cdot (1 - t_r) \quad (1)$$

The new position, Pos, is given by Equation 2, where Pos' is the current position, $\overrightarrow{heading}$ is given from Equation 1, and $velocity$ is the velocity of the cuboid.

$$Pos = Pos' + \hat{\overrightarrow{heading}} \cdot velocity \quad (2)$$

The start size, start position, target size, and destination are modeled as mathematical distributions. The following distributions are supported. Uniform, Zipf [13], and Gaussian [1]. These are shown as 2D data in Figure 3. Figures 3(a), 3(b), and 3(c) each show single snapshots of cuboid positions. Figures 4 shows a series of snapshots where cuboids are born uniformly spread all over the world and moving towards the upper left corner. This is accomplished by using a uniform distribution for assigning start positions, and a Gaussian distribution for assigning destinations to the cuboids.



(a) Uniform (b) Gaussian (c) Zipf

Figure 3: Distribution of cuboids

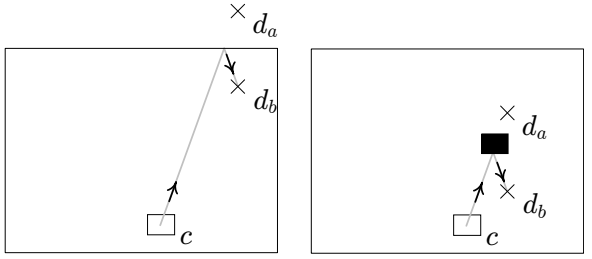
Figure 4: Using two distributions

4.2 Worlds and Blocked Spaces

Cuboids are born in stationary worlds, in which their movement is restricted. This means that a cuboid cannot leave one world, unless it enters another. Blocked spaces are stationary and placed in worlds. Cuboids cannot enter blocked spaces, and if a cuboid hits a blocked space, it *bounces*.

Worlds and blocked spaces have similar behavior. (a) They are both stationary. (b) They restrict object movement. (c) They are positioned and given size as per cent of the universe size. The universe is the entire area in which worlds and blocked spaces can be positioned.

If a cuboid is given a destination outside its world, this must be handled. This is illustrated in Figure 5(a) where the cuboid, c, has the destination d_a. The cuboid travels towards this destination, but cannot reach it because it positioned outside the world. At some point the cuboid hits the side of the world and bounces back. This is realized by (a) mirroring the destination on the side of the world, and (b) setting the cuboid's heading to point directly towards the destination. The cuboid then travels towards d_b.

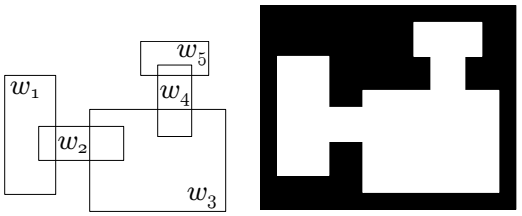


(a) Bounce on world-side (b) Bounce on blocked space

Figure 5: Cuboid bouncing

Worlds can contain blocked spaces. In Figure 5(b) a blocked space is shown as a black rectangle. The cuboid has a destination, d_a, which makes the cuboid's path cross the blocked space. As when a cuboid hits the side of the world, the cuboid bounces of the side of the blocked space. Again the destination is mirrored on the side of the blocked space, making the cuboid travel towards d_b.

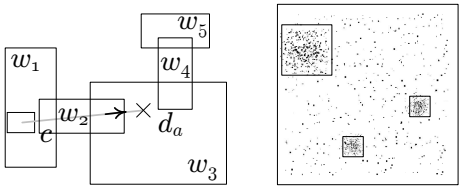
The motion of cuboids is restricted in two ways. Either when hitting world sides or blocked spaces. As shown in Figure 5(b) worlds and blocked spaces can be used together. It is possible to make scenarios where cuboids move between large worlds connected by narrow worlds. In Figure 6(a) the cuboid movement is restricted by the five worlds. In Figure 6(b) movement is restricted by blocked areas. In both scenarios the same space for cuboid movement is present.



(a) Restriction using worlds (b) Restriction using blocked spaces

Figure 6: Cuboid movement restriction

Figure 7(a) shows a cuboid, c, born in w_1 and having a destination, d_a, in w_3. On its path it first leaves the world w_1. As c is allowed to change worlds, it does not bounce on the side of w_1.



(a) Cuboid changing worlds (b) Four worlds

Figure 7: Cuboid movement restriction

Worlds can be used to make complex simulations. Figure 7(b) shows a scenario containing four worlds. Each world contains the same amount of cuboids. This simulates a map with three cities. The cities have a higher density of cuboids than the area surrounding them.

4.3 Parameters

In this section the parameters for the data generation algorithm are presented. The parameters for the algorithm are divided into four groups. Data size, environment, initialization of the cuboids, and movement of cuboids.

4.3.1 Data Size

These parameters control the amount of data produced by the generator. Two parameters control the data size.

Number of cuboids The number of cuboids generated by the data generator

Number of snapshot per cuboid The number of occurrences of a given cuboid in the data set

4.3.2 Environment

These parameters specify the information about the environment in which the cuboids move. The environment is controlled by the following parameters.

Number of dimensions This specifies the dimensionality of the data set. All cuboids in the generated data set is of this dimensionality

Universe size The size of the entire area in which worlds and blocked spaces can be positioned

Worlds A set of worlds including their sizes and positions

Blocked spaces A set of blocked spaces including their sizes and positions

4.3.3 Initialization of Cuboids

Each world holds statistical distributions that control the initialization of the cuboids within the world. The two distributions for initialization are the following.

Start position Controls where in the world the cuboids are born according to the given distribution

Start size Controls the initial size of the cuboids according to the given distribution

4.3.4 Movement of Cuboids

As with initialization of cuboids, the movement of cuboids is controlled by two different distributions for each world. The two distributions are the following.

Destination Controls where cuboids select their destinations according to the given distribution

Target size Controls the size that the cuboids resize towards according to the given distribution

4.4 The Algorithm

The data generation algorithm is divided into two phases; initialization and movement. In the initialization phase cuboids are assigned properties, e.g., size, position, and destination. In the movement phase cuboids change position, size etc. Different distributions can be used for each of the phases for destination and target size.

```
1  procedure initialize(conf)
2    worlds ←createWorlds(conf)
3    blockedspaces ←createBlockedSpaces(conf)
4    foreach(w ∈ worlds)
5      foreach(obj ∈ w)
6        obj_size←startSize(w)
7        obj_targetSize←newTargetSize(w)
8        obj_postion←startPostion(w)
9        obj_destination←newDestination(w)
10       obj_heading← startHeading(obj)
11       obj_velocity←startVelocity(obj)
```

Listing 1: Initialization

Listing 1 shows the algorithm for initialization. In Lines 2–3 worlds and blocked spaces are created. Lines 4–11 iterate over all the worlds. Lines 5–11 iterate over all the cuboids in every world. In Lines 6–11 cuboids are assigned their properties.

In the movement phase the following is done in each snapshot. First, a new size, velocity, position, destination, and heading are given to a cuboid. A cuboid may get the same destination in more than one snapshot because a new destination is only chosen when the distance to its current destination is less than a certain threshold. Then three things are examined. Should the cuboid bounce on a blocked space? Should the cuboid change world? Or should it bounce on a world side?

Lines 2–24 in Listing 2 run the algorithm *snapshots* times. Lines 3–24 iterate over all the worlds. Lines 4–24 iterate over the cuboids in each world. Lines 5–6 update cuboids with new sizes and positions. Lines 7–12 check whether the distance to the destination has not gotten smaller within a given number of snapshots. A new destination is

```
1  procedure moveCuboid(worlds,blockedspaces,
       snapshots)
2    foreach(snapshots)
3      foreach(w ∈ worlds)
4        foreach(obj ∈ w)
5          obj_size ←newSize(obj, w)
6          obj_position ←newPosition(obj, w)
7          if (obj_disToDest ≈ obj_lastDisToDest)
8            obj_notCloser←obj_notCloser + 1
9          else obj_notCloser←0
10         if (obj_notCloser = closerThreshold ∨
               obj_disToDest ≈ 0)
11           obj_destination←newDestination(w)
12           obj_notCloser←0
13         obj_heading← newHeading(obj, w)
14         obj_velocity ←newVelocity(obj)
15         foreach(b ∈ blockedspaces)
16           while (intersects(obj,b))
17             bounceOnBlockedSpace(obj,b)
18         if (w_mayLeave)
19           foreach(w' ∈ worlds\{w})
20             if (w'_mayEnter ∧ canPass(obj,w,w'))
21               w←w\{obj}, w'←w' ∪ {obj} //change
                     world
22               w←w', changedWorld←true
23         if (¬changedWorld ∧ outside(obj,w))
24           bounceInWorld(obj, w)
```

Listing 2: Movement phase

also given to the cuboid, if it has reached its current destination in Line 10–11. Lines 13–14 update the heading and velocity of the cuboid. Lines 15–17 iterate over *blockedspaces*. Lines 16–17 check if the cuboid has to bounce on a blocked space. Line 18 examines if the cuboid is allowed to leave its world. If it is then Lines 19–22 iterate over *worlds* to check if a cuboid has to change its world. Line 20 checks if a potential new world, w', has been configured to accept new cuboids. If this is the case and if the cuboid intersects this world and can fit, then it is moved from w to w'. Lines 21–22 move the cuboid to its new world. Lines 23–24 check if the cuboid has not left its world and if it has it must bounce on the world.

4.5 Output

The output from the data generator can be written to a file. A fragment of an output file from the data generator is shown in Figure 8, showing two cuboids in two snapshots in 2D. The first column is the cuboid's id. The second column is the time stamp. The last two columns are the lower left, *lc*, and upper right, *uc*, corner of the cuboid. This output corresponds to the definition of a spatial object in [9].

id	*time*	*lc*	*uc*
1	1	11,40	14,42
2	1	3,19	11,33
1	2	12,42	15,43
2	2	3,20	11,34
...			

Figure 8: Data fragment

5 Workload Generation

When testing the performance of an index structure it is necessary to control the ratio of the different modifications and queries in order to fully understand the results of the tests. The workload generator is able to output a number of inserts, followed by a number of loads composed of a mix of inserts, updates, deletes, k-NN queries, and range queries. Last a number of k-NN queries and range queries can be put in the workload.

When dealing with GPS-data the positions are not always accurate [3]. In order to make the synthetic data more realistic, the workload generator can add noise to size, position, and time stamps of objects.

5.1 Parameters

In the following the parameters for the workload generator are presented. These parameters are split into three groups. Data size, spatial queries, and spatial and temporal noise.

The workload generator is able to produce two kinds of output.

- A workload that consist of n loads
- At least n operations per object on a collection of o objects

Therefore, the workload generator uses two different algorithms as shown in Appendix A. The two algorithms use the same parameters except the ones that control the data size; that is *number of loads*, *number of objects*, and *operations per object*.

5.1.1 Data Size

The following parameters control the amount of data produced by the workloads algorithms.

Number of loads The number of loads that will be generated by the workload generator. This parameter is only used by the number-of-modifications algorithm shown in Listing 4 in Appendix A

Number of objects The number of objects generated by the workload generator. This parameter is only used by the number-of-modifications algorithm shown in Listing 5 in Appendix A

Operations per object The number of operations generated by the workload generator. The parameter is used only by the number-of-loads algorithm shown in Listing 5 in Appendix A

Number of start inserts The number of start inserts added to the workload before any loads. Note this number can be zero

Inserts, updates, deletes The number of inserts, updates, and deletes in a load. Note these numbers can be zero

When configuring the workload generator the ratio of start inserts (i_s), inserts (i), and deletes (d) in a load must be considered. Otherwise, a situation having more deletes than inserts could occur. Equation 3 describes the ratio that must be respected in order to guarantee a successful generation in which l is the number of loads.

$$i_s + i \cdot l \geq d \cdot l \tag{3}$$

The *number of objects parameter* in the number-of-loads algorithm is derived from $i_s + i \cdot l$.

5.1.2 Spatial Queries

The following parameters control the spatial queries in the workload.

k-NN and range queries The number of k-NN and range queries in a load

k-NN and range size Controls the size of k-NN and range queries

End queries The number of k-NN and range queries to be placed last in the workload.

5.1.3 Spatial and Temporal Noise

The following parameters control how spatial and temporal noise is added to the objects. Spatial noise is a distribution that controls how a controlled inaccuracy is added in a given radius of an object. Temporal noise controls both the length of the loads and the distribution for selecting time stamps in the timeslots, which are elaborated on in Section 5.4. The parameters for noise are listed below.

Spatial noise A distribution that controls the amount of noise added to objects

Temporal length of loads A distribution controlling the temporal length of loads

Placement in timeslot This is a distribution that controls where in a timeslot the object is placed

5.2 Algorithm

This section describes the core algorithms in the workload generator. As mentioned in Section 5.1 there are two algorithms.

5.2.1 Number-of-Loads Algorithm

The algorithm for generating *n loads* can be seen in Listing 4 in Appendix A.1. This algorithm outputs a file with $conf_{startInserts}$ inserts. This is followed by $conf_{loads}$ loads each consisting of $conf_{inserts}$ inserts, $conf_{updates}$ updates, and $conf_{deletes}$ deletes. The load also contains $conf_{knn}$ k-NN queries and $conf_{range}$ range queries. Finally, $conf_{endRange}$ range queries and $conf_{endKnn}$ k-NN queries are output.

The workload generator fetches a new object from the data generator when an insert is made. The object is put in the set of currently inserted objects named *objects*. When an object is to be updated, the workload generator randomly selects an object from *objects*, and asks the data generator for the next snapshot for this object.

Deletes are conducted by randomly selecting an object from *objects* and removing it.

5.2.2 Number-of-Modifications Algorithm

The algorithm for generating a workload with at least n modifications per object on a collection of o objects can be seen in Listing 5 in Appendix A.2.

It outputs a file containing $conf_{startInserts}$ followed by a number of loads which are followed by a number of end queries. This algorithm guarantees a workload containing $conf_{objectCount}$ objects each having at least $conf_{minModifications}$ modifications. An object can be inserted, a number of updates conducted on it, deleted, and then inserted again. Because modifications are carried out on randomly chosen objects, a maximum number of modifications per object cannot be guaranteed, but the algorithm aims towards having at most $conf_{minModifications} + 1$ modifications on each object.

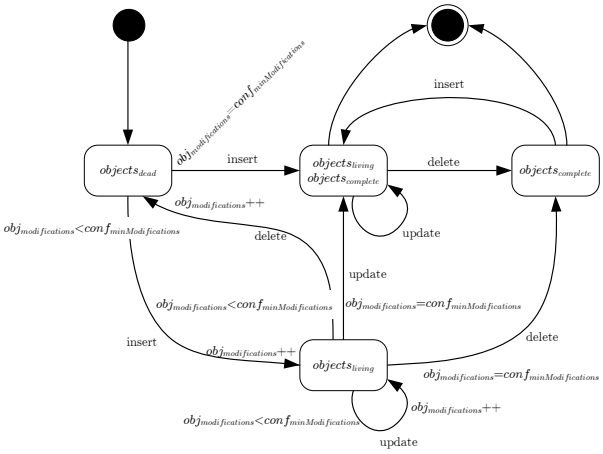


Figure 9: State diagram for an object

The algorithm utilizes three sets of objects. $objects_{living}$, $objects_{dead}$, and $objects_{complete}$. $objects_{living}$ contains the currently inserted objects. $objects_{dead}$ contains objects that are not currently inserted, but still has not reached $conf_{minModifications}$. Objects having reached $conf_{minModifications}$ are put in $objects_{complete}$. These objects are however not excluded from also being in $objects_{living}$. Generally $objects_{complete}$ is used to prevent deadlock situations. This could occur if there are no objects in $objects_{dead}$, and the workload generator was to issue an insert. Then no object could be chosen for this modification. This is avoided by choosing an object only residing in $objects_{complete}$. A state diagram for an object can be seen in Figure 9. It illustrates how an object can move between the three sets.

When the workload generator is to output an insert it chooses an object from $objects_{dead}$. If this is not possible it chooses an object from $objects_{complete}$ which is not currently inserted, although it has reached $conf_{minModifications}$ modifications. If possible an update is conducted on an object from $objects_{living}$ which is not in $objects_{complete}$. Otherwise, an object that occurs in both sets is chosen. When a delete is to be made, an object is chosen from $objects_{living}$ which is also in $objects_{complete}$ if possible. This means that an object which has reached $conf_{minModifications}$ is deleted. If no objects occur in both $objects_{living}$ and $objects_{complete}$, an object is chosen from $objects_{living}$. When deleted, the object is removed from $objects_{living}$. Furthermore, if the object is not in $objects_{complete}$ it is added to $objects_{dead}$.

5.3 Spatial Noise

Spatial noise means that the coordinates of an object are distorted. This is often useful when simulating GPS data, as there are inaccuracy in the measurements.

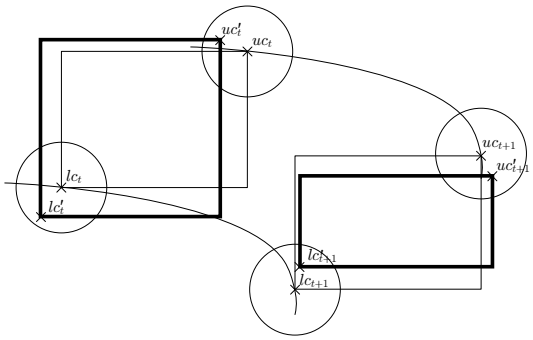


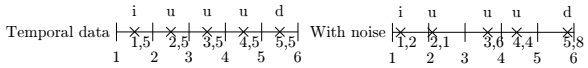
Figure 10: Adding spatial noise

In Figure 10, the original coordinate pair (lc_t,uc_t) is moved to (lc'_t,uc'_t). This is done for the lower left corner and upper right corner of the MBR of the object in the 2D example. The emphasized box in the figure indicates the new size and shape of the object. The new position for the corner is selected inside a circle with regard to the given distribution (uniform, Gaussian, or Zipf). The default distribu-

tion is Gaussian in the center of the circle as this according to [3] as this approximates GPS-noise well. The size of the circle symbolizes the maximum amount of noise added to a coordinate. Object noise is calculated for each modification, and does not affect other modifications for the same object.

5.4 Temporal Noise

When looking at the output from the data generator in Section 4.5, each object at the same snapshot has the same time stamp. This is however not very realistic. An example could be a taxi company having all their taxis send their position every ten seconds. It would be unrealistic that all positions would arrive at the same time at the server. Therefore, it is necessary to model that time between modifications varies. A load is divided into a number of *timeslots*. A timeslot is the time in which a given operation must occur.



(a) Operations placed naively in their timeslots

(b) Operations placed uniformly in their timeslots

Figure 11: Adding temporal noise

Figure 11(a) shows how a load is split into timeslots and operations are placed regularly in the middle of the given timeslots. Figure 11(b) shows how operations are uniformly placed in their timeslots and that time between operations varies. To simulate periods with varying activity, it is possible to vary the temporal length of a load. This is shown in Figure 12.

```
1  function timeStamp(operations,temporalLength)
2    time←temporalLength / operations
3    times←∅
4    for(i←0 to operations − 1)
5      r←distributeTime(i · time , (i + 1) · time)
6      times←times ∪ {r}
7    return times
```

Listing 3: Algorithm for temporal noise

Listing 3 shows the algorithm for calculating time stamps for operations in a load. Line 1 takes in the number of operations and the time the load spans. Line 2 calculates the length of a timeslot and assigns it to *time*. Lines 4–6 iterate *operations* times. Line 5 generates a time stamp for the current timeslot according to the chosen distribution. In Line 6 the time stamp is saved in the set *times*. Line 7 returns the time stamps for the load.

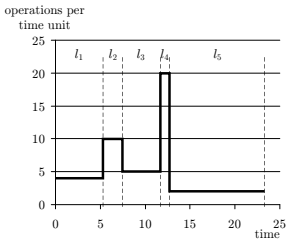


Figure 12: Different length of loads

5.5 Query and Result Generation

This section describes how relevant spatial queries and their result sets are produced. The focus will be on positioning the two important spatial queries range and k-NN. Instead of positioning the queries randomly in space, the main part of the queries should be placed where objects are present. The spatial queries and their result sets will be embedded in the workload file. This is done to ensure that the result set of a query corresponds to the objects contained in the database at the given time.

When generating workloads the STDW data generator is controlled by the workload generator. The workload generator positions queries according to the statistical distribution that controls the movement of objects. This is done by asking the world for a new destination which is basis for the range and k-NN query. This solution, however, cannot be used with an external data source.

For choosing query points when using an external data generator the approach is slightly different. When a query is positioned, a random object is chosen. Now the center of this object is the query point. This approach will reflect the distribution, as there statistically will be more query points where there are many objects.

5.5.1 k-NN Query

A k-NN query is defined by a point in space, and a number, k. The k-NN query returns the k nearest neighbors to the given point. The output from

the workload generator of a k-NN query is the center point, then k, followed by two lists. The first list contains the object ids of those nearest objects which are bound to be in the result set. The second list contains object ids that might be in the result set. In a 5-NN query the nearest three objects may have different distances to the query point. These object ids are put in the first list. But the fourth closest object can be one of three objects, all having the same distance to the query point. In this situation 5 objects cannot be chosen based on distance. Therefore, these three objects are put in the second list. The result set is split in two lists as the workload does not know how the DBMS chooses the 5 objects. A list of object ids which must be returned by the DBMS, and a list of object ids that may be returned.

5.5.2 Range Query

The range query is defined by a rectangle in space. It returns objects that intersect with this rectangle. The output from the workload generator for a range query are the corners of the rectangle and the result set. The user inputs the size of a range query, which is a percentage of the universe.

5.6 Output

The output from the workload generator is written to a file. A fragment of an output file from the workload generator is shown in Figure 13. A line starting with `i` represents an insert. It is followed by an object id, the time stamp, the lower corner of the object, and upper corner of the object. `u` is an update with the same output as an insert. `d` is a delete. `d` is followed by the object id and the time. `r` represents a range query. It is followed by the lower corner, the upper corner of the range and the result set. `k` symbolizes a k-NN query, which is followed by the point of the query, the number k, and two lists.

```
...
i    35     10.3    10,43     13,44
u    35     11.7    11,40     14,42
d    17     12.4
r    5,7    20,10   1,6,14
k    27,5   5       2,19,22   5,13,36
...
```

Figure 13: Workload fragment

6 Visualization

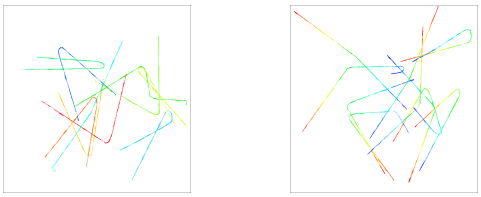
When the data is generated it is very useful to visualize it. This way the user can get a visual impression of how the generated data looks. It is easy to determine whether changes in, e.g., size and position of the objects fulfill the demands of the user. Visualization is possible in both 2D and 3D.

It is possible to visualize all object positions in one image. This means that the whole path of one object is drawn as one line in the image. This makes it easy to get an impression of the objects movement in space. Either each object has its own color, or each time stamp has its own color.

Visualization is especially useful for finding the parameter values needed for the desired scenario. This way a small sample can be generated and visualized. When the correct parameters are found the whole data set can be generated.

6.1 Two Dimensions

Visualization in 2D is quite simple. This can be done by drawing the regions directly to a bitmap image. This is implemented into STDW. An example of a 2D visualization is shown in Figure 14. Note that the Figures 14(a) and 14(b) show two different simulations.



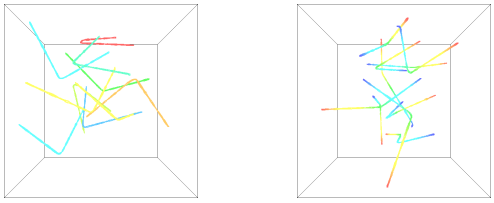
(a) One color per object (b) One color per snapshot

Figure 14: Visualization in 2D

6.2 Three Dimensions

3D objects are difficult to visualize in a bitmap image. Therefore, the output is VRML [12] which

can be shown in an existing external viewer. Viewing objects in a VRML viewer makes it possible to rotate, zoom and move around in the data. An example of a 3D visualization is shown in Figure 15. Note that the Figures 15(a) and 15(b) show two different simulations.



(a) One color per object (b) One color per snapshot

Figure 15: Visualization in 3D

	Oporto	*GSTD*	*G-Terd*	*STDW*
2D	√	√	√	√
3D				√
n dimensions				√
Points	√	√	√	√
Spatial Extension	√	√	√	√
Workload				√
Text output	√	√		√
Large number of objects		√	√	√
Visualization	√	√	√	√
Controlled inaccuracy				√
Spatial queries with results				√
Restricted areas	√	(√)		√

Figure 16: Feature matrix

6.3 Feature Matrix

Figure 16 shows a comparison of features between the reviewed data generators and STDW. It shows that only STDW is able to produce nD data. All of the data generators are able to produce 2D data. Also points and spatial extension are supported by all the generators. Only the STDW generator is able to produce a workload. Text output is supported by all the generators but G-TERD as it produces raster images. As Oporto is 16 bit, it is not able generate more than 2^{16} objects. All the generators are able to visualize the generated data. Only STDW is able to add noise to the generated result and add relevant spatial queries. Finally, only Oporto and STDW are able to restrict certain areas for object movement. Furthermore, [8] proposes an extension to GSTD that allows restricted areas.

7 Performance

This section will test the performance of the described data and workload generator. The tests are performed on a 1.8 GHz Intel Pentium 4 processor with 1 GB RAM. The operating system is Microsoft Windows Server 2003 Enterprise Edition with service pack 1.

The data and workload generators are implemented in C# (approximately 4,000 lines of code). Size, position, heading, and destination for moving cuboids are implemented as vectors. This makes the generators able to easily expand the data to any dimensionality.

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

7.1 Data Generator

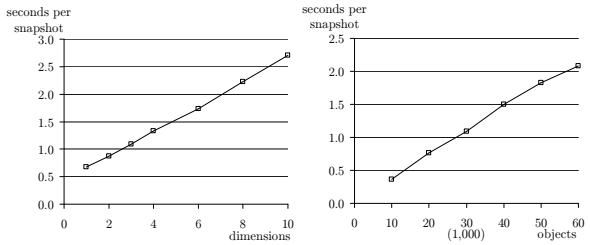
This section will test the performance of the data generator. The tests will show how the generator scales with regard to dimensionality, number of objects, number of snapshots, and number of worlds.

Figure 17 shows the performance of the data generator.

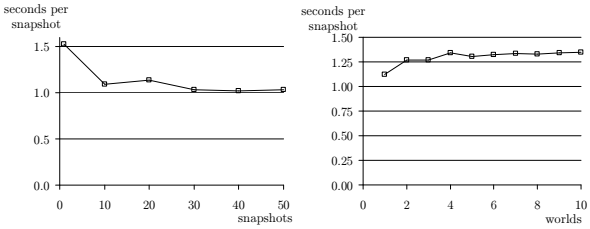
Figure 17(a) shows the performance with regard to the number of dimensions. The measurements are performed on the 10th snapshot generating 30,000 objects. The x-axis shows the number of dimensions and the y-axis shows the time spent. The figure shows that the time consumption is linear in the number of dimensions.

Figure 17(b) shows the performance with regard to the number of objects. The measurements are performed on the 10th snapshot in 3D. The x-axis shows the number of objects generated and the y-axis shows the time spent. The figure shows that the time consumption is almost linear in the number of objects.

Figure 17(c) shows the performance with regard to the number of snapshots. The measurements are



(a) Number of dimensions (b) Number of objects



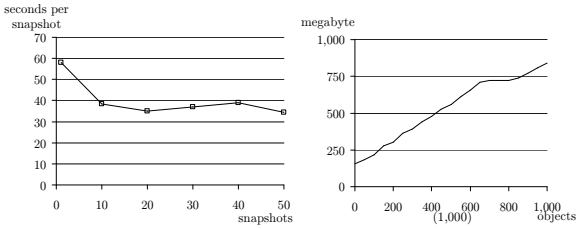
(c) nth snapshot (d) Number of worlds

Figure 17: Data generator performance

performed on 3D data with 30,000 objects. The x-axis shows the nth snapshot and the y-axis shows the time spent. The data generation is split in two phases, initializing and movement. As it can be seen from the chart, the first snapshot, which belongs to the initializing phase, consumes more time than the following snapshots. In the movement phase the time consumption per snapshot is constant. Due to the design criteria that output should be easily parsed, objects are generated in parallel. This requires all objects to be initialized at the beginning of data generation which results in the first snapshot being generated slower than the following.

Figure 17(d) shows the performance with regard to the number of worlds. The measurements are performed on the 30,000 objects in 3D where the 10th snapshot is timed. The x-axis shows the number of worlds and the y-axis shows the time spent. The figure shows that using only one world is less time consuming than using multiple worlds. When using more the one world the time consumption is almost constant.

Figure 18(a) shows the time used for generating the nth snapshot for 1,000,000 objects in 3D. Similarly to Figure 17(c) it can be seen that initialization of objects requires an overhead in snapshot one. In the succeeding snapshots the time consumption stabilizes.



(a) 1 million objects in 3D (b) Memory consumption

Figure 18: 1 million objects test

Figure 18(b) shows the memory consumption of the data generator. The test is performed with up till 1,000,000 objects in 3D. The measurements are performed after the first snapshot. The test shows that the memory consumption is linear in the number of objects.

The tests have shown that it takes about 30 minutes to generate 50 snapshots with 1,000,000 objects in 3D. This is 50,000,000 object snapshots, which is about 27,000 modifications per second. The generator can generate about 11 gigabytes per hour.

7.2 Workload Generator

This section will test the performance of the workload generator. The tests will show how the generator scales with regard to the two different workloads algorithms from Appendix A. Furthermore, testing generation with queries and generation with queries including results are carried out.

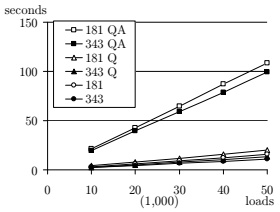


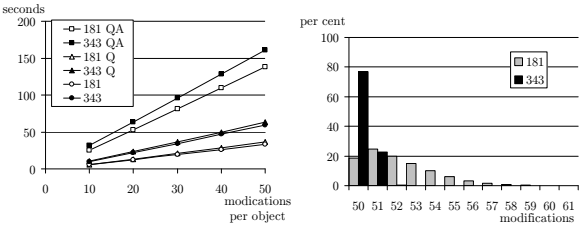
Figure 19: Number-of-loads test with different loads

Figure 19 shows the time used for generating from 10,000 loads to 50,000 loads using the number-of-loads algorithm. E.g., 181 represents a workload containing loads with one insert, eight updates and

one delete. Q is when a load also contains one range query, and one 100-NN query, but without query results. QA means the load also contains query result sets.

As it can be seen from the figure the most time consuming tests are to generate workloads with queries and result sets. This is due to the fact that the workload generator has to scan all objects in order to investigate which objects are in the result set. Figure 19 illustrates that the overhead when generating loads with queries, compared to loads without queries is very small.

It can also be seen that loads containing three inserts, four updates, and three deletes are faster than loads containing one insert, eight updates, and one delete. This is because each insert and update requires computation of the next object position and size where deletes do not need these calculations. This means that in a 181 load nine new positions and sizes are computed. In a 343 load only seven new object positions and sizes are computed. This results in less computation in a 343 load than in a 181 load.



(a) Different loads (b) Modifications per object

Figure 20: Number-of-modifications test

Figure 20(a) shows the time used to generate a workload using the number-of-modifications algorithm. This means at least n operations on 10,000 objects. n is varied on the x-axis. It can be seen that generating loads consisting of three inserts, four updates, and three deletes are slower than loads containing one insert, eight updates, and one delete. This is in contrast to number-of-loads test. This is because inserts and deletes always move objects between different sets, as illustrated in Figure 9. Again, generating queries with answers are very time consuming as all objects must be scanned.

Figure 20(b) shows how many modifications there are on each object when using the number-of-modifications algorithm. The test is performed with 5,000 start inserts and 10,000 objects configured for minimum 50 modifications on each object. The figure shows that a 181 workload has more objects over 50 modifications than the 343 workload. This is due to the lower number of inserts.

8 Conclusion

This paper has presented STDW, a multi-dimensional spatio-temporal data and workload generator. The data generator simulates cuboids moving between worlds and around blocked spaces. This makes it easy for the user to create a realistic universe in which cuboids move.

The workload generator can use any data generator that implements a simple interface. It can generate realistic scenarios where objects are modified in a mix of inserts, updates, and deletes, and spatial queries. The workload generator is also able to return the result sets for the queries in order to ease the verification of tests.

The data is visualized in both two and three dimensions. This helps users to better understanding of the consequences of changes in settings.

A comparison to the three dominant data generators shows that STDW is the only one able to produce nD data. Furthermore, STDW is able to generate workloads from external data souces.

The performance shows that the generator is able to produce 27,000 object snapshots per second. It also shows that the time consumption is linear in the number of dimensions and objects.

Future work is to implement a spatial index into STDW. This way, the generation of result sets should become faster.

Acknowledgements

The authors would like to thank José Moreira et al. for providing us with the Oporto [7] source code. The authors would also like to thank Mario Nascimento for his help during the data generation. We would also like to thank Kristian Torp for guidance and help throughout the project period. Without his feedback this paper could not have been realized.

References

[1] G. E. P. Box and M. E. Muller. A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics*, pages 610–611, 1958.

[2] T. Brinkhoff. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, vol. 6, no. 2, pages 153–180, 2002.

[3] F. Van Diggelen. GPS accuracy: Lies, damn lies, and statistics. *GPS World*, vol. 9, no. 1, pages 41–44, 1998.

[4] M. A. Nascimento, D. Pfoser, and Y. Theodoridis. Synthetic and Real Spatiotemporal Datasets. *IEEE Data Engineering Bulletin*, vol. 26, no. 2, pages 26–32, 2003.

[5] 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*, pages 214–221, 1993.

[6] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *ACM SIGMOD*, pages 71–79, 1995.

[7] J. Saglio and J. Moreira. Oporto: A Realistic Scenario Generator for Moving Objects. *GeoInformatica*, vol. 5, no. 1, pages 71–93, 2001.

[8] Y. Theodoridis and D. Pfoser. Generating Semantics-Based Trajectories of Moving Objects. *Intern. Workshop on Emerging Technologies for Geo-Based Applications*, pages 59–76, 2000.

[9] Y. Theodoridis, T. Sellis, A. N. Papadopoulos, and Y. Manolopoulos. Specifications for efficient indexing in spatiotemporal databases. *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, pages 123–132, 1998.

[10] Y. Theodoridis, J. R. O. Silva., and M. A. Nascimento. On the Generation of Spatiotemporal Datasets. *International Symposium on Spatial Databases*, no. 6, pages 147–164, 1999.

[11] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. On the Generation of Time-Evolving Regional Data. *GeoInformatica*, vol. 6, no. 3, pages 207–231, 2002.

[12] W3C. VRML Virtual Reality Modeling. http://www.w3.org/MarkUp/VRML/, June 2006.

[13] G. K. Zipf. *Selected Studies of the Principle of: Relative Frequency in Language*. Harvard University Press, 1932.

A Workload Algorithms

This section presents the workload algorithms discussed in Section 5.2.

A.1 Number-of-Loads Algorithm

```
1  procedure nol(conf, dataSource)
2    for(1 to conf_startInserts)
3      obj←dataSource.newObject()
4      addTimeAndSpatialNoise(obj)
5      objects←objects ∪ {obj}
6      writeInsert(obj)
7    for(1 to conf_loads)
8      for(1 to conf_inserts)
9        obj←dataSource.newObject()
10       addTimeAndSpatialNoise(obj)
11       objects←objects ∪ {obj}
12       writeInsert(obj)
13     for(1 to conf_updates)
14       obj←random(objects)
15       dataSource.nextSnapshot(obj)
16       addTimeAndSpatialNoise(obj)
17       writeUpdate(obj)
18     for(1 to conf_deletes)
19       obj←random(objects)
20       objects←objects\{obj}
21       writeDelete(obj)
22     makeQueries(conf, objects)
23   makeEndQueries(conf, objects)
```

Listing 4: Algorithm for generating *n workloads*

In Line 1, the algorithm receives the parameters *conf* which contains parameters for the workload generator as described in Section 5.1 and *dataSource*. *dataSource* can either be the data generator described in Section 4 or any external data generator that implements the STDW-workload interface described in Section 3.2. Line 2 runs Lines 3–6 $conf_{startInserts}$ times. $conf_{startInserts}$ represents the number of inserts done before doing the loads. Line 3 acquires a new object from the data source. In Line 4 noise is added to the position of the object, and the time is changed. This new object is added to the set of currently inserted objects in Line 5. Line 6 handles output of the insert to the file. Line 7 runs the Lines 8–22 $conf_{loads}$ times. $conf_{loads}$ is the number of mixes of inserts, updates, deletes, and queries. Lines 8–12 are iterated $conf_{inserts}$ times. Line 9 acquires a new object from the data source. The time is changed and spatial noise is added to the object in Line 10. In Line 11 the new object is added to the set of objects which are currently inserted. Line 12 outputs the object to the file. Line 13 runs Lines 14–17 $conf_{updates}$ times in order to make $conf_{updates}$ number of updates. Line 14 selects a random object from the set of currently inserted objects. Line 15 asks *dataSource* for the next snapshot of that object. Line 16 changes the time and adds spatial noise to the object. Line 17 outputs the update to the file. Line 18 runs the Lines 19–21 $conf_{deletes}$ times. In Line 19 an object is randomly selected. This object is removed from the currently inserted objects in Line 20. Line 21 outputs this delete to the file. Finally in Lines 22–23 queries are produced and added to the file. These queries consist of a number of k-NN queries and range queries.

A.2 Number-of-modifications Algorithm

```
1  procedure nom(conf, dataSource)
2    for(1 to conf_objectCount)
3      obj←dataSource.newObject()
4      objects_dead←objects_dead ∪ {obj}
5    for(1 to conf_startInserts)
6      obj←random(objects_dead)
7      objects_dead←objects_dead\{obj}
8      addTimeAndSpatialNoise(obj)
9      objects_living←objects_living ∪ {obj}
10     writeInsert(obj)
11   while(|objects_complete| < conf_objectCount)
12     for(1 to conf_inserts)
13       if(objects_dead ≠ ∅) obj←random(objects_dead)
14       else obj←random(objects_complete\objects_living
           )
15       objects_dead←objects_dead\{obj}
16       dataSource.nextSnapshot(obj)
17       addTimeAndSpatialNoise(obj)
18       objects_living←objects_living ∪ {obj}
19       writeInsert(obj)
20       if(obj_modifications = conf_minModifications)
21         objects_complete←objects_complete ∪ {obj}
22     for(1 to conf_updates)
23       if(objects_living\objects_complete ≠ ∅)
24         obj←random(objects_living\objects_complete)
25       else obj←random(objects_living)
26       dataSource.nextSnapshot(obj)
27       addTimeAndSpatialNoise(obj)
28       writeUpdate(obj)
29       if(obj_modifications = conf_minModifications)
30         objects_complete←objects_complete ∪ {obj}
31     for(1 to conf_deletes)
32       if(objects_living ∩ objects_complete ≠ ∅)
33         obj←random(objects_living ∩ objects_complete)
34       else obj←random(objects_living)
35       objects_living←objects_living\{obj}
36       if(obj_modifications = conf_minModifications)
37         objects_complete←objects_complete ∪ {obj}
38       if(obj ∉ objects_complete)
39         objects_dead←objects_dead ∪ {obj}
40       writeDelete(obj)
41     makeQueries(conf, objects)
42   makeEndQueries(conf, objects)
```

Listing 5: Algorithm generating at least n modifications per object on a collection of o objects

The algorithm in Listing 5 resembles the one in Listing 4. In Line 1 the algorithm receives the parameters $conf$ which contains parameters for the workload and $dataSource$. $dataSource$ can either be the data generator described in Section 4, or any external data generator that implements the STDW-workload interface described in Section 3.2. Line 2 runs Lines 3–4 $conf_{objectCount}$ times. $conf_{objectCount}$ is the total number of objects. Line 3 gets a new object from the data source, and Line 4 adds this object to the set of objects currently not inserted, called $objects_{dead}$. Line 5 runs Lines 6–10 $conf_{startInserts}$ times. In Line 6 a random object is chosen from $objects_{dead}$. The object is removed from $objects_{dead}$ in Line 7. Spatial noise is added to the object position, and the time stamp is changed in Line 8. In Line 9 the object is added to the set of currently inserted objects. The object information is written to the file in Line 10.

Line 11 runs Lines 12–41 until all objects have reached $conf_{minModifications}$. Lines 12–21 are iterated $conf_{inserts}$ times. In Line 13 it is examined whether any objects are currently not inserted that have not reached $conf_{minModifications}$. If this is the case one of them is chosen. Else in Line 14 an object is chosen from $objects_{complete}$. This object is to be inserted, hence it cannot be in $objects_{living}$. The object is removed from $objects_{dead}$ in Line 15. In Line 16 the next snapshot for the object is calculated. Line 17 adds noise to the object. The object is added to $objects_{living}$ in Line 18. In Line 19 the object information is written to the file. Lines 20–21 checks whether the object has reached $conf_{minModifications}$. If this is the case the object is added to $objects_{complete}$.

Line 22 runs Lines 23–30 $conf_{updates}$ times. In Lines 23–24 a random object, that has not reached $conf_{minModifications}$, is chosen from $objects_{living}$. If this is not possible an object is chosen from $objects_{complete}$ in Line 25. The next snapshot for the object is calculated in Line 26. Spatial noise is added to the object's position, and the time stamp is changed in Line 27. The update is written to the file in Line 28. If the object has reached $conf_{minModifications}$ it is added to $objects_{complete}$ in Lines 29–30. Lines 31–40 $conf_{deletes}$ times. In Lines 32–33 a random object in $objects_{living}$, which has reached $conf_{minModifications}$, is chosen. If this is not possible a random object is chosen from $objects_{living}$ in Line 34. In Line 35 the object is removed from $objects_{living}$. If the object has reach $conf_{minModifications}$ it is added to $objects_{complete}$ in Lines 36–37. On the other hand, if the object has not reached $conf_{minModifications}$ it is added to $objects_{dead}$ in Lines 38–39. The delete is output to the file in Line 40. Finally in Lines 41–42 queries are produced and added to the file. These queries consist of a number of k-NN queries and range queries.