NTNU | Norwegian University of Science and Technology

Implementing Z-ordering as Spatial Index in MySQL

TDT 4900 - Computer Science, Master's Thesis

Author: Gaute Håhjem Solemdal

Supervisor: Norvald Ryeng

Faculty of Information Technology and Electrical Engineering
Department of Computer Science
29.06.2021

Abstract

In this thesis, Z-ordering is implemented and integrated into MySQL Server to serve as a prototype spatial index. The prototype is compared to MySQL's R-tree for window queries on a large (4 million) data set of geographic points.

The experiments show that the R-tree outperforms the prototype. However, it is also discovered that the prototype index exhibits poor and inconsistent filtering accuracy. Without an algorithm that prunes the search range of Z-values in a window, it is susceptible to crossing 'unlucky' borders in the grid, leading to a potentially enormous amount of rows being returned from the index - even though the amount of rows in the result is small.

Although not tested through experiments, the indexing times recorded during the experiment setup showed that the prototype's indexing time was significantly faster than the R-tree's time.

Additionally, the grid that is manifested as a result of applying the Z-order curve to Earth is explored. Various aspects of the grid are presented and discussed. In particular, the varying cell sizes and non-geodesic cell sides/edges are shown to be problematic. In short, the curvature of the Earth mandates special considerations for geometry decomposition and k-NN search.

Preface

This master's thesis was conducted at the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway during the spring semester of 2021. The project was written for and in cooperation with Oracle and MySQL. Norvald Ryeng, Software Development Director at Oracle and adjunct associate professor at NTNU, supervised the project.

The thesis is accompanied by a patch file (created with *git diff*) that contains the additions and modifications of the MySQL source code. It can be applied on top of the correct commit (see section 5.1.2) to recreate the state of the code as of the experiments.

The data set used in the experiments is derived from OpenStreetMap data (© OpenStreetMap contributors) which is available under the Open Database License (ODbL 1.0). Figures 9, 14, 19, 20 and 25 are made using the website OpenStreetMap WKT Playground [5], which also uses OpenStreetMap data to produce its maps.

I would like to thank Oracle and MySQL for providing the opportunity to write a thesis with their support. Special thanks to Chaitra Gopalareddy for her invaluable help with understanding the range optimizer and Torje Digernes for helping me with various GIS stuff and keeping me sane during my endeavours with MySQL's code base. Last but not least, I would like to thank Norvald Ryeng for excellent supervision. Answers to my questions were only a quick message away throughout the entire project.

Contents

List of Figures

List of Tables

Listings

1 Introduction

As location-aware applications have become ubiquitous and spatial data have become abundant - so has the support for it in general-purpose database management systems (DBMSs). An integral part of that support is spatial indexes. For any data type, having an index to efficiently search through the data set is beneficial compared to evaluating the whole set - and the benefit grows with the size of the set. For spatial data, many of the functions involve expensive geometric operations, leading to an even greater benefit than usual.

Preliminary work of the author, Solemdal [30], found that the most popular, general-purpose relational DBMSs use one of two approaches to spatial indexing: either a variant of the R-tree or some scheme involving a space-filling curve (SFC) with multi-level grids.

In this thesis, the Z-order curve is used to implement a prototype of the latter variant, albeit with a single-level grid, that can index Point data. The implementation is integrated into MySQL and its performance compared to MySQL's R-tree using a large set of geographic Point data. Various characteristics of the grid, manifested from the implementation, are also explored.

1.1 Motivation

Even though the industry seems to have settled on two variants for spatial indexing, few resources provide empirical data comparing their performance. In addition, *"... the index performance depends heavily on the nature of the data and the type of applications."* as stated by Fang et al. [6].

Additionally, the author found little information regarding all quirks and caveats appearing when creating a grid on the Earth using an SFC. As shown in this thesis, applying an SFC to a sphere creates several technical and complex problems. All but a few resources seen by the author only talk about SFCs in a manner of Cartesian space. The notable exceptions are the two systems actually implementing an SFC for spatial indexing; S2 Geometry [9] and Microsoft SQL Server [18].

1.2 Goals and research questions

To try to remedy the shortcomings mentioned above of the literature this thesis attempts to achieve the following goals:

- GOAL 1: Provide empirical evidence furthering the understanding of Z-ordering's performance compared to R-trees.
- GOAL 2: Give a detailed description of how the Z-order curve was implemented and integrated into MySQL.

- GOAL 3: Explore and present interesting grid characteristics stemming from applying the Z-order curve on Earth.

The thesis attempts to answer the following research question in order to achieve goal 1:

- RESEARCH QUESTION: How does Z-ordering compare against the R-tree for window queries on Point data?

1.3 Scope

This thesis is only concerned with two-dimensional geographic data. Spatial indexes can be seen as a subset of multi-dimensional indexes, but the terms are often used interchangeably, and data does not have to be geographic. For example, the Z-order curve can be used to index pixel color data in an image; RGB values corresponding to a three-dimensional Cartesian space.

The implementation limits the experiments to only test point data. Other geometry types can produce different results.

Obviously, the thesis is also limited to the Z-order curve as space-filling curve and MySQL as the DBMS whose R-tree is used as a benchmark. Other SFCs and other DBMSs can produce different results.

1.4 Thesis structure

Chapter 2 gives an overview of the theory and definitions needed to understand the rest of the thesis. It introduces a taxonomy for spatial extents and spatial queries, and briefly explains the concept of spatial indexing and how Z-ordering and R-trees achieve it.

Chapter 3 presents the essential details of the implementation and challenges with integrating it into MySQL. With goal 3 in mind, chapter 4 explores the manifested grid.

Chapter 5 details the experiments and presents the results, with chapter 6 following with an evaluation of the results.

Chapter 7 discusses various aspects of the thesis, and chapter 8 rounds off by concluding the findings of the thesis.

2 Background Theory

This chapter gives an overview of the theory and definitions used in the thesis. It attempts to be concise and focuses on the particular technologies used in the implementation and those already present in MySQL. As a consequence, many technicalities and alternatives that are strictly not necessary are omitted.

The reader is referred to Rigaux et al.'s [25] textbook on spatial databases for a more extensive introduction to spatial indexing, and to Gaede and Günther's [7] landmark survey paper for an overview of various methods for indexing multi-dimensional data.

Preliminary work of the author, Solemdal [30], examined which approaches the most popular, commercial database management systems (DBMS) used for spatial indexing and gave a theoretical overview of them. Sections 2.3.1 and 2.3.2 are partially adapted from that overview.

2.1 Geometries

A data object can often be associated with a *spatial extent*. The spatial extent is a representation of an object's location and potentially its surface. There are multiple ways of defining a spatial extent, but the thesis (and MySQL) follows the Open Geospatial Consortium's (OGC) OpenGIS Simple Feature Access standard [11]. The standard describes the common architecture for simple feature geometry. It defines a class hierarchy with a base class; Geometry. Subclasses such as Points, Linestrings, and Polygons are more specialized variants of Geometry.

Put simply, the standard defines a Point by its coordinates and uses Points as building blocks for Linestrings and Polygons. A Linestring is a sequence of line segments represented as a sequence of Points. A Polygon is a surface represented by a sequence of Points making its outer boundary, and potentially many sequences making internal boundaries (holes in the surface). The three geometry types are shown in figure 1.

Throughout the thesis *geometry* and *geometries* are used to refer to Geometry objects/spatial extents. Points with a capital P are used when the actual Point geometry type is at hand.

The OpenGIS Simple Feature Access standard [11] also defines a textual representation of geometries that is human-readable; Well-known Text (WKT) Representation. For example

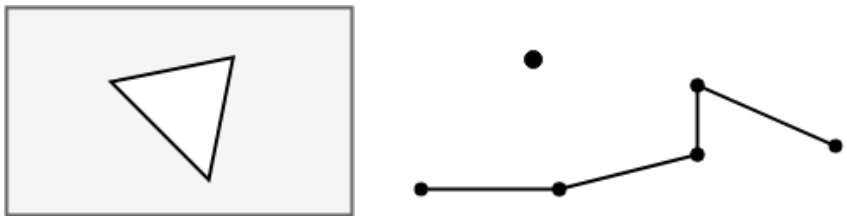


Figure 1: Geometries: Point, Linestring and Polygon (with an internal hole).

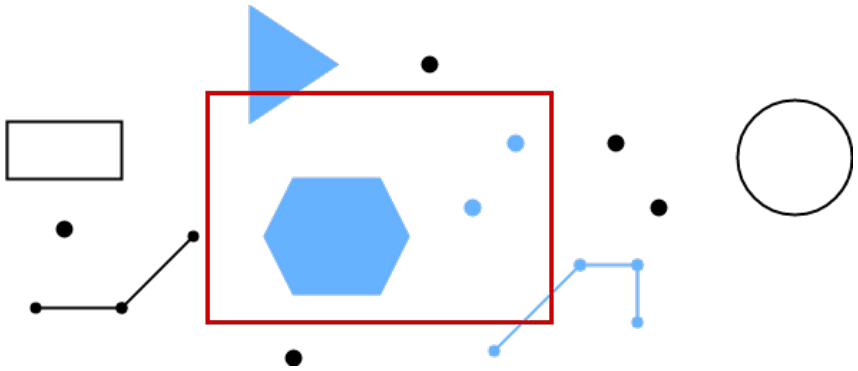


Figure 2: Window query. The window is red and intersected geometries highlighted in blue.

"LINESTRING(0 0, 3 0, 6 1, 6 2, 7 1)", which resembles the Linestring in figure 1.

2.2 Spatial queries

Let a *spatial query* be a query involving some operation on one or more geometries. For example, if one wants to find all objects that intersect[1] a Polygon X, then one must compare every object's geometry to X to determine whether they have some point or area in common.

There are many interesting spatial queries - but also many different ways to define them. Gaede and Günther [7] provides formal definitions of 9 queries using set notation. The intersection query is reproduced in equation 1. *o* and *o'* denotes objects, and *o.G* and *o'.G* their respective geometries.

$$IQ(o') = \{o | o'.G \cap o.G \neq \emptyset\} \tag{1}$$

Several other queries are similar, only expressing slightly different (stricter) forms of overlap. While the intersection query only requires the intersection between the two geometries to be non-empty, the containment query reproduced in equation 2 requires the intersection to be equal to *o*'s geometry.

$$CQ(o') = \{o | o'.G \cap o.G = o.G\} \tag{2}$$

The other relevant queries for this thesis are the *k* nearest neighbors query (k-NN) and the window query. The k-NN query finds the *k* nearest neighbors of an object. The window query is simply an intersection query where the query object is a *window* - an axes-aligned rectangle. It's shown in figure 2.

[1] Intersection is unfortunately an overloaded word, used for a specific query, a mathematical term in set theory, as well as its layman usage.

The Dimensionally Extended 9-Intersection Model (DE-9IM) [33] is a rigorous model for describing spatial relations between two geometries, but is strictly not necessary for this thesis and hence only mentioned. It is important for defining how two geometries interact in an intersection - especially for geometries with a different amount of dimensions, e.g., Linestring (1) and Polygon (2).

2.3 Spatial indexing

Let an *index* be a data structure that facilitates a fast search through a collection of data. One example is the B+tree with a logarithmic time complexity for search, O(log n), where n is the number of elements in the collection. For large collections, having a search time of log n is a drastic improvement over scanning the whole collection.

Processing spatial queries can be very expensive, making the benefit of an index even greater. Gaede and Günther [7] state that *"..., although the computational costs vary among spatial database operators, they are generally more expensive than standard relational operators."* Rigaux et al. [25] call them *"... complex and costly geometric operations."*

Hence, it is desirable to index spatial data. However, because spatial data is two-dimensional (or more), one cannot directly use one-dimensional indexes such as the B+tree. A naive approach using two consecutive one-dimensional indexes (e.g., first latitude then longitude) does not scale. Gaede and Günther [7] explains that *"Since each index is traversed independently of the others, we cannot exploit the possibly high selectivity in one dimension to narrow down the search in the remaining dimensions."*

There is a plethora of various indexing schemes that are designed to handle multi-dimensional data. Gaede and Günther's paper Multidimensional Access Method [7] is an elaborate survey of such methods. Earlier work of the author [30] found that (as of 2020) the most popular commercial DBMSs use some variant of the R-tree or a scheme involving space-filling curves and multi-layered grids. These are the two methods relevant to this thesis.

Both methods indexes approximations of geometries. This means that the index alone can lead to false positives. Such indexes are typically seen as a part of a two-step process. In the *filter step* the index is used to filter out objects that definitely do not fulfill a predicate. In the *refinement step* the actual geometries (of the reduced set) are evaluated, removing false positives. The filter step can be enough to send an object straight to the result set in the proper context.

In short, the idea behind spatial indexing is to efficiently filter out irrelevant data points based on information about location in space. Spatial data, being multi-dimensional, requires more sophisticated indexes or systems than the typical B+tree and hash indexes used for 'normal' data in a DBMS.

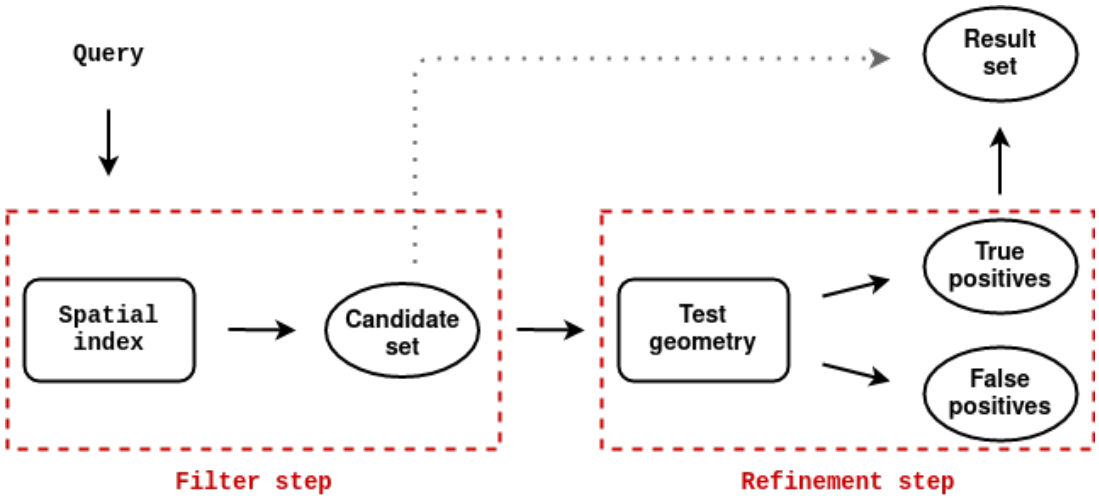


Figure 3: Two-step spatial query processing. Inspired by Gaede and Günther [7].

Note that the literature often uses the term *spatial access method* (SAM). However, the commercial DBMSs surveyed by Solemdal [30] use *spatial indexing*. For clearance, spatial indexing is a system that uses a spatial index.

2.3.1 R-trees

The original R-tree was introduced by Guttman [10]. It has gotten many extensions and adaptations, most notably the R*tree introduced by Beckmann et al. [2] and the R+tree introduced by Sellis et al. [28]. MySQL's implementation is of the original R-tree.

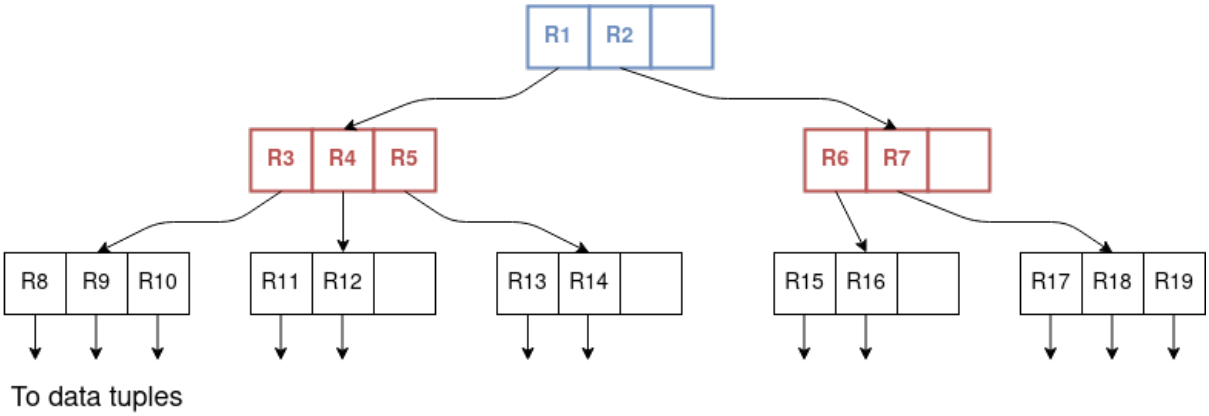
For R-trees, a geometry is approximated by a *bounding box*, the smallest axes-aligned rectangle that contains the geometry. The tree consists of two types of nodes:

- Leaf nodes - Containing bounding boxes of objects and pointers to the objects' locations on disk.
- Internal nodes - Containing bounding boxes of leaf nodes or lower-level internal nodes.

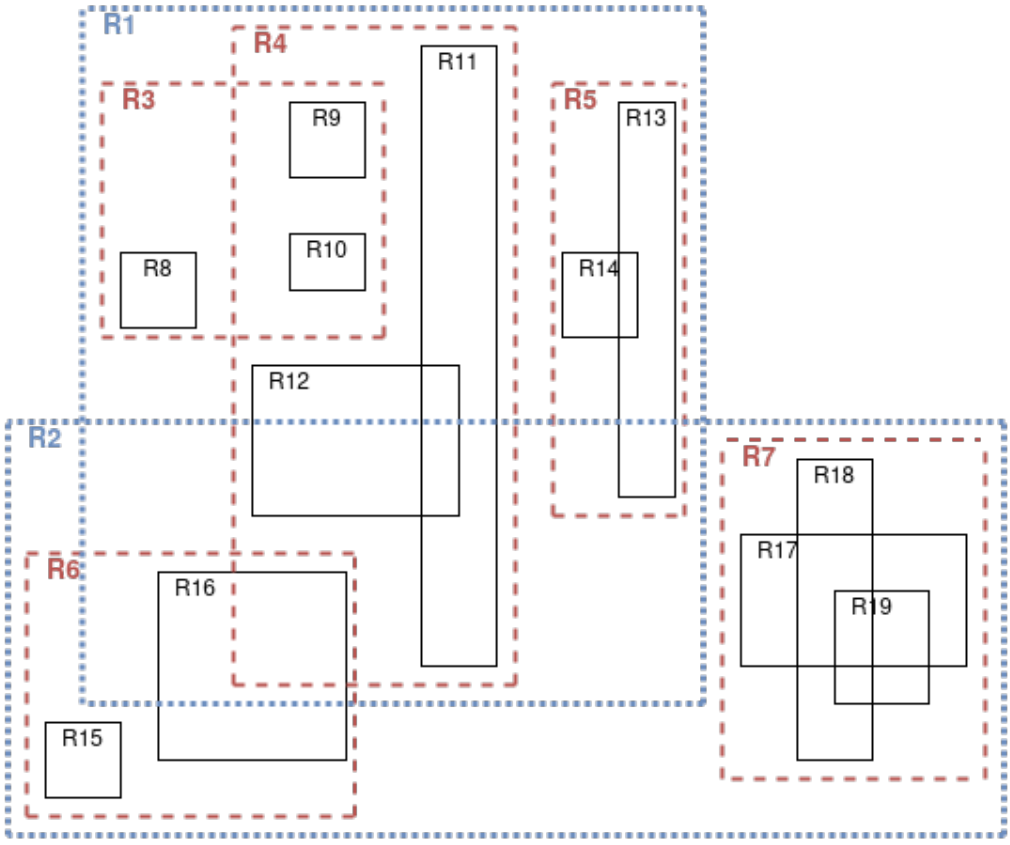
In figure 4b leaf node bounding boxes, R8-R19, are drawn with black, whole lines. The two next levels of internal bounding boxes are drawn with red dashed lines and blue dotted lines, respectively. The resulting hierarchy in tree form is shown in figure 4a. The idea is to group boxes that are close and envelop them in a higher-level bounding box. For example, R7 is the parent bounding box of R17, R18, and R19. Further upwards, R2 is the parent of the internal nodes R6 and R7.

When processing queries, a candidate region is approximated by a bounding box, then the tree is traversed (starting at the root) and the candidate box compared to nodes' boxes. This filters out irrelevant regions by stopping the traversal down the branch.

Sizes of nodes can be chosen such that it matches a desirable disk page size. Furthermore,



(a) Tree structure of the R-tree.



(b) Bounding boxes of the R-tree nodes in space.

Figure 4: R-tree. Adapted from Guttman [10].

one can specify a lower bound m of the number of objects in a node, enforcing a minimum number of objects in a node. The upper bound M is the maximum number of objects possible in a node. These two parameters influence the height and width of the tree, as well as how often reorganization might be triggered.

Similar to the B-tree, the R-tree is height-balanced and dynamic. As stated by Guttman [10] *"The index is completely dynamic; inserts and deletes can be intermixed with searches and no periodic reorganization is required"*. Reorganization is automatically triggered by insertions leading to overfilled nodes and by deletions leading to underfilled nodes. In the former case, a node split occurs and objects are redistributed among the two nodes. In the latter case, a node is deleted and its remaining objects are redistributed to other nodes.

The reader is referred to Guttmann's [10] paper for a high-level overview of algorithms and to Rigaux et al. [25] for a textbook explanation and pseudocode.

2.3.2 Space-filling curves

The Encyclopedia of GIS [19] defines:

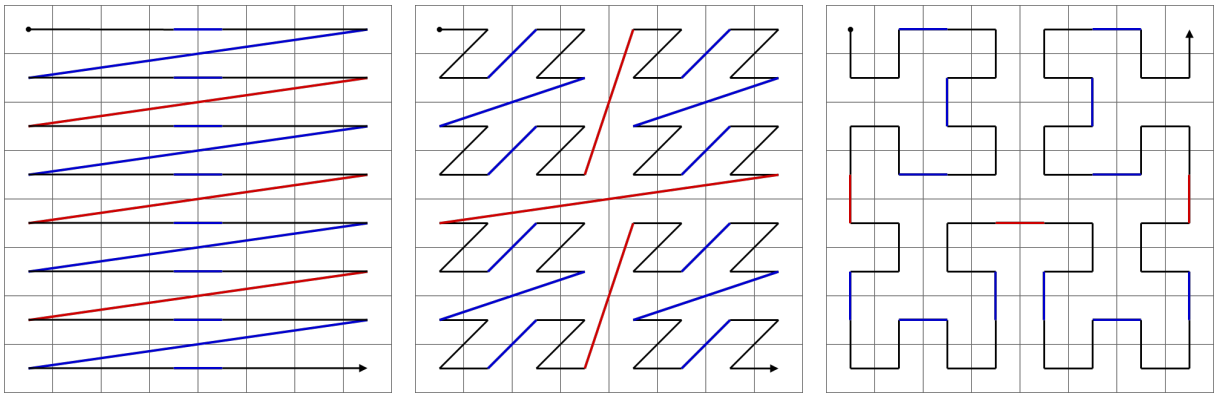
> *"A space-filling curve (SFC) is a way of mapping a multi-dimensional space into a one-dimensional space. It acts like a thread that passes through every cell element (or pixel) in the multi-dimensional space so that every cell is visited exactly once."*

An SFC gives a *total ordering* of the space, meaning that any two elements can be compared. In other words, it can be used to reduce the multi-dimensional nature of spatial data to one-dimensional, and thus enabling the use of a B+tree. This is what makes using an SFC for spatial indexing a compelling alternative; the reuse of resources that are already prevalent in DBMSs.

There are many possible space-filling curves. The Z-order curve and Hilbert curve are shown in figure 5, along with a simple row-wise variant. Both van Oosterom [32] and Samet [27] illustrate other curves and summarize desirable properties SFCs should have.

Two important properties are:

- Easy mapping between space and ordering
  - It should be easy to find the ordering given a location in space and to find a region in space given an element in the ordering.
- Preservation of spatial proximity
  - Cells that are close to each other in space should also be close to each other in the ordering. It is inevitable to have some cells that are neighbors in space yet far apart in the ordering - but some curves are better in general.



(a) Row order curve (b) Z-order curve (c) Hilbert curve

Figure 5: Space-filling curves on 8x8 grids. '4-groups' are connected by blue segments, and '16-groups' are connected by red segments.

Both van Oosterom [32] and Gaede and Günther [7] summarize academic comparisons of SFCs. Z-ordering and the Hilbert curve are considered to be the best alternatives.

The Hilbert curve is regarded to have superior clustering, largely due to no long jumps in space to get to the next cell. The next cell in the order will always be an adjacent one. On the other hand, the Hilbert curve's mapping process is, as stated by Samet [27], considerably more complex.

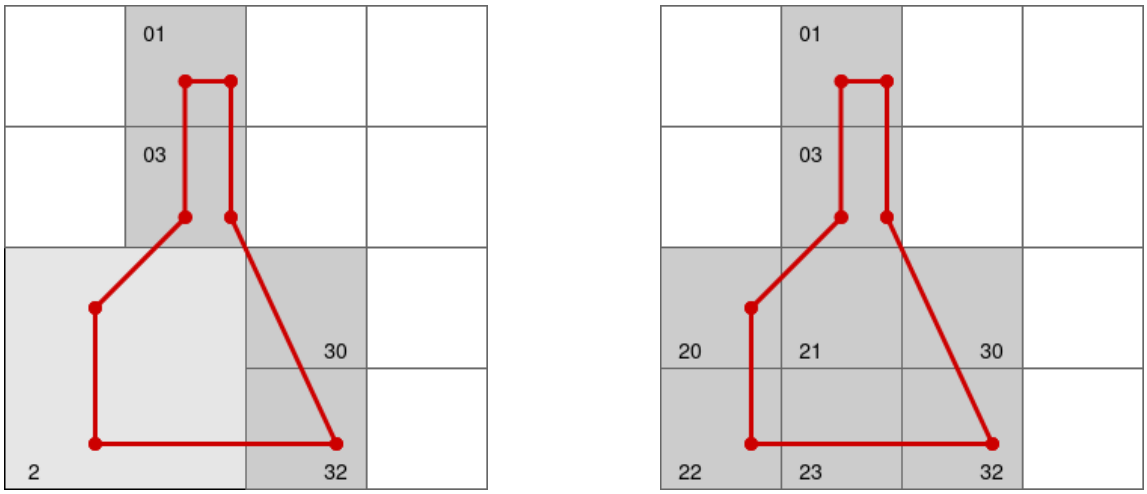
2.3.3 Z-ordering

Let *Z-ordering* be a spatial indexing scheme that uses the Z-order curve to order cells in a grid. Let the *Z-value* be the number in the ordering that maps to a cell. The Z-value is found by simply interleaving the binary representation of the cell's coordinates, i.e., interleaving two bitstrings. Each cell has its own unique Z-value.

A finite bitstring representing a Z-value may not be sufficiently long to uniquely represent all points in the underlying space. In such situations, the cells will contain several points. Hence, a point maps to one Z-value, and a Z-value maps to a region in space. This distinguishes Z-ordering usage in continuous spaces from discrete spaces.

In Z-ordering, a geometry is approximated by the set of cells it intersects. For each cell intersected, an entry (cell-id, object-ID) is added to the index, where cell-id is the cell's Z-value. The set of cells is called the geometry's *decomposition*. Some sources use the term *tessellation*.

This is a simple interpretation of Z-ordering suited for representing Z-values as unsigned integers. It corresponds to what Rigaux et al. [25] call the *raster variant of Z-ordering with redundancy*. They explain Z-ordering in a different way, somewhat as an extension to quadtrees.



(a) Quadtree-like decomposition.

(b) Raster decomposition.

Figure 6: Two variants of decomposing geometries in Z-ordering.

Their quadtree-like variant represents Z-values as strings consisting of characters [0, 1, 2, 3] - one for each quadrant. This can be beneficial as a group of sub-quadrants can be aggregated into a parent quadrant, as seen in figure 6a. The quadrants 20, 21, 22, and 23 are replaced by a larger quadrant 2, resulting in three fewer entries to the index. On the other hand, whereas integers are easily comparable, these strings must be compared in a lexicographical order that handles variable-length strings. For example, $020 < 2 < 20$.

The argument for fewer entries and thus a smaller index is strong, especially for large collections of spatial data and/or large geometries. However, actual implementations of spatial indexing using space-filling curves, such as S2 Geometry [9] and Microsoft SQL Server 2019 [18] solves this problem by using multi-level grids.

The redundancy part of the name is due to multiple entries in the index pointing to the same object. This is also a problem for the quadtree-like variant. The redundancy creates a duplicate removal problem that must be handled.

Rigaux et al. [25] mention a variant without redundancy. It only sends to the index the smallest quadrant that wholly contains the geometry. This removes the duplication problem and leads to a potentially much smaller index but can also lead to remarkably poor approximations when geometries overlap borders dividing large quadrants.

2.3.4 Important distinctions between Z-ordering and the R-tree

There are two important distinctions to be made. Firstly, by providing a one-dimensional ordering of cells, Z-ordering can reuse the existing B+tree index. This benefit is twofold. There is no need to implement a new indexing mechanism, and it also comes with transactional

properties[2] such as concurrency control, recovery, etc.

Secondly, R-trees approximate a geometry by a bounding box, whereas Z-ordering approximates a geometry by a set of cells. No matter the shape or size of a geometry, R-trees approximate it by one bounding box (two points for 2D). For Z-ordering, the set of cells can grow large as the geometry grows larger and/or gets a more complex shape. Kothuri et al.'s [12] comparative analysis found that *"Storage requirements for a Quadtree are nearly the same as those for R-trees when the data is points and more otherwise."*

The effect of this difference in geometry representation on query speed is not that easy to distill because it's also influenced by other aspects such as comparisons during tree traversal; simple arithmetic greater/lower than comparisons for the B+tree and bounding box overlap for R-trees.

[2] In the author's opinion, a reasonable assumption for any general-purpose, commercial DBMS

3 Z-order Implementation

Sections 3.1 through 3.4 explain the most important details of, and choices made for, this implementation of Z-ordering. Section 3.5 sheds light on some hurdles that were encountered when trying to integrate the index into the MySQL code. The additions and modifications to the MySQL source code enable indexing of point geometries and usage of window queries which is presented in section 3.6.

Section 3.7 gives an overview of some unimplemented features that would be important for an implementation of Z-ordering to become a full-fledged spatial index.

3.1 N-shaped curve

The Z-order curve can take many shapes depending on which coordinate goes first in the interleaving and in which directions the coordinates increase. Figure 7 illustrates three variations. The typical Z-shape occurs when the Y-coordinate goes first (most significant bit), and the upper left corner is the base. A mirror image Z-shape occurs when the Y-coordinate goes first and the lower left corner is the base. An N-shape occurs when the X-coordinate goes first and the lower left corner is the base.

For this thesis, the N-shape was chosen. Having the base in the lower left corner and dimensions increasing upwards and rightwards follows the typical representation of coordinate systems, and having a reversed shape was not preferred. This is also more in line with geographical coordinates, which increase northwards (upwards) and eastwards (rightwards). Figure 8 shows the N-curves for a 2-bit, 4-bit, and 8-bit string, respectively.

The bitstring was chosen to be 32 bits long, leaving both latitude and longitude with 16 bits accuracy. By simply interpreting/storing the strings as unsigned integers, we get a range of values, [0, 4 294 967 295], in a ubiquitous number type.

Having an equal amount of bits for both latitude (-90, 90) and longitude (-180, 180) leads to wide cells as longitude must cover twice the range of latitude. However, as is shown in section 4.1 neither shape nor size of cells are constant as the width narrows towards the north/south pole, resulting in tall, slim rectangles despite the starting "advantage" at the Equator.

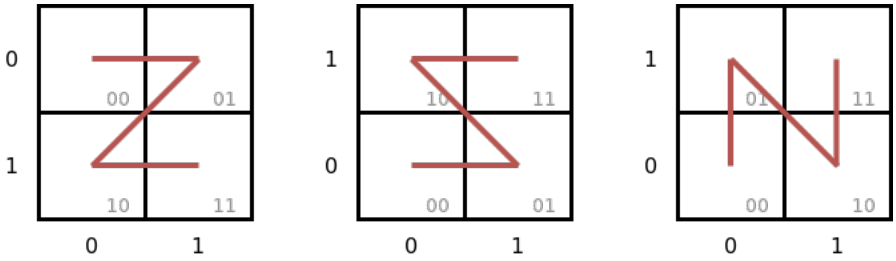


Figure 7: Various shapes of the Z-order curve.

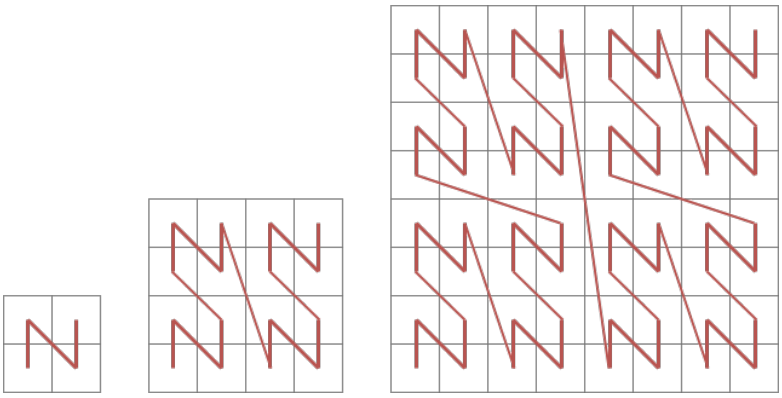


Figure 8: 2x2, 4x4 and 8x8 N-curves.

3.2 Mapping from coordinates to cell

The typical explanation of Z-ordering tells that the bitstrings of two coordinates are interleaved, and the resulting bitstring is the value that represents the containing cell's place in the ordering. This works perfectly fine for systems with integer coordinates, such as computer graphics with raster representations.

The challenge becomes more complex for geographical coordinates, which can be both negative and positive - and have a (theoretically unbounded) fractional part. Such coordinates should be stored in some floating-point format, typically having a sign bit, exponent part, and mantissa part. One cannot simply interleave bitstrings of such formats and expect the result to be sensible.

To solve this problem the method first converts a coordinate to an integer representation. It does so by iteratively building the bitstring. In each iteration, the coordinate is compared to the middle value of a lower and upper bound (starting at -90 and 90 or -180 and 180). If the coordinate is smaller than the middle, the bit should be 0 (1 for the opposite case). The middle value then becomes the new lower (upper) bound. After 16 rounds, we have a bitstring of length 16 representing the coordinate's location along one dimension as an integer value. This can be seen as a mapping of a coordinate from a real number in [-90, 90] (or [-180, 180]) to an integer in [0, 65535].

The implemented function simultaneously calculates the next bits for both latitude and longitude and directly builds the interleaved bitstring. It can be seen in appendix A.

3.3 Mapping from cell to coordinates

When mapping from coordinates to cell, one starts with *one* point, which is represented by *one* cell (Z-value). But a cell is a container with spatial extent - it contains (theoretically infinitely) many points. Mapping from a cell to coordinates should therefore give a set of coordinates. It can be expressed by four bounds, lower and upper for both latitude and longitude.

Starting with bounds at -90 and 90 (or -180 and 180), the method iterates through the bitstring, adjusting one of the bounds based on the bit value. A bit with value 1 implies a location in the higher half of the range between the bounds - resulting in the lower bound being set to the middle of the range. After all 32 bits of the string are evaluated, the bounds envelop the cell.

By taking the middle point of the bounds, one gets the middle point of the cell. The implemented function finds this point, but a simple adjustment can give the bounds. It can be seen in appendix A.

3.4 Neighbor functions

One of the major benefits of Z-ordering is its structure, in particular, that if a coordinate's value is incremented, the resulting interleaved bitstring represents the next cell in that coordinate's direction. This neat characteristic is used to create *neighbor functions*.

By (1) deinterleaving a Z-value into its latitude and longitude parts, then (2) adding/subtracting 1 from one of the coordinates, and (3) interleaving the parts again, one can find the cell's neighbor in all four directions. For example, adding 1 to the latitude part results in the northern neighbor cell.

Figure 9 shows a point, that point's containing cell, and that cell's neighbor cells. The cells are represented as Linestrings between the corner points. The corner points are computed based on the lower and upper bounds found when mapping from cell (Z-value) to coordinates as explained in section 3.3.

The (de)interleaving of steps (1) and (3) above use functions provided by the Libmorton C++ library [1]. By using sophisticated techniques, the (de)interleaving is done very fast, leading to the neighbor functions also being fast as they only require one additional simple arithmetic operation. Note that Libmorton expects a Z-shaped curve. To accommodate this, the implementation must add/subtract to the other coordinate. The function that finds the northern neighbor can be seen in appendix A.

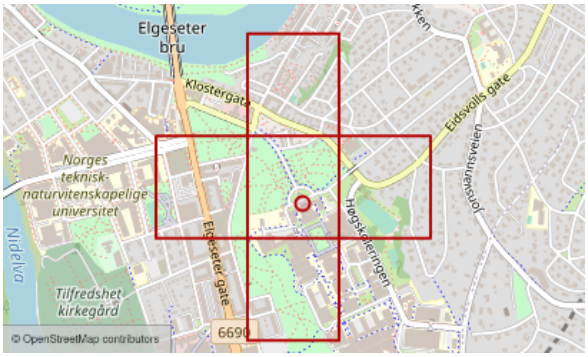


Figure 9: A point's cell and its neighbor cells.

3.5 MySQL integration

One of the sales points of using space-filling curves for spatial indexing in a DBMS is that they can reuse a lot of code that is ubiquitous in those systems. The one-dimensional ordering one gets from applying the curve can be indexed by already implemented B+trees. On the other hand, spatial indexes such as the R-tree need its own specialized index implementation.

It was experienced through integrating the Z-ordering prototype into MySQL that it still is no straightforward task, and several hurdles had to be overcome to get a functional prototype.

3.5.1 Creating an index

In MySQL one can create an index on a column with the following SQL statement:

```
CREATE INDEX name ON table(column);
```
(3)

A spatial index (R-tree) is created by adding the SPATIAL keyword before INDEX[3]. It would therefore be desirable to create a Z-order index in a similar fashion:

```
CREATE ZPATIAL INDEX name ON table(column);
```
(4)

Unfortunately, this would entail far too much work than would be feasible during this thesis. The problem stems from trying to index a derived value - and not the column itself. MySQL expects to find the value (or range of values) in the index that matches what is searched for. This leads to a mismatch when the column to be searched through has some Geometry type, and the values of the Z-ordering index are integers.

The R-tree obviously overcomes this hurdle, but it is treated differently than B+trees. Ryeng [26] explains that to match this model, a B+tree with SFC values would have to implement an R-tree interface for window search - which is no easy task.

The alternative, fundamentally changing how MySQL understands indexes, is arguably much more work. As a consequence, the prototype was implemented so that it requires the Z-values to be in a dedicated column. The index can then be created like a normal index as done in statement 3.

3.5.2 Optimizer awareness of the Z-value column

Having Z-values in a dedicated column introduces a new problem. Code must now be added to the existing range optimizer to make it "switch" - make it filter out rows based on values

[3] Newer versions of MySQL recognizes that the column type is some geometry type and creates an R-tree without the SPATIAL keyword, but it's kept here for an illustrative purpose.

in the Z-value column even though the query uses a spatial function on the geometry column. The range optimizer was extended to make this switch, but it did not choose a range scan on the Z-column for the following query:

```
SELECT * FROM table WHERE Z_CONTAINS(@qw, g);          (5)
```

where @qw is a variable set to a geometry of type Polygon (representing the query window), and g is the geometry column.

There is no mention of the Z-column in the query, resulting in the range optimizer not being aware of that the key (index) is usable for this query, so it disregards it and defaults to a table scan. A quick fix was to add a second condition to the WHERE clause, which triggered the range optimizer to use the key:

```
SELECT * FROM table WHERE Z_CONTAINS(@qw, g) AND z > 0;          (6)
```

where z is the indexed column containing the Z-values.

This triggered the range optimizer to use the key. Other attempts, such as mentioning the column in the SELECT list, did not work. A solution like this would of course pollute any results in a performance analysis as it adds an unnecessary condition that must be evaluated.

A proper solution was found by passing the Z-column to the function as an argument and forcing the optimizer to add the key of that column to its candidate set of keys when it recognizes the spatial function Z_CONTAINS. The following query then results in the range optimizer choosing a query execution plan that uses a range scan on the Z-column:

```
SELECT * FROM table WHERE Z_CONTAINS(@qw, g, z);          (7)
```

3.6 Window query

As mentioned in section 2.2 a window query is an intersection query where the query geometry is an axes-aligned rectangular Polygon. Since only point geometries can be indexed, the intersection function can be substituted with the containment function. A more specific wording of the query would then be: find all points contained by the window.

Given a table with a geometry column *g* and a corresponding Z-value column *z*, along with a Polygon *@qw* the query can be used in the form seen in statement 7 from section 3.5.2. The z column must be indexed as shown in statement 3. If not, there will not be an index to use, and a table scan will be executed.

Z_CONTAINS mimics the functionality of MySQL's original ST_CONTAINS and gives the same result, but differs in the execution plan due to different indexes being used to improve query speed.

The query can take advantage of the index by translating the window into a single range of Z-values and using it to perform a range scan on the z column. The lower bound is the value of the cell containing the lower left corner, and similarly, the upper right corner gives the upper bound. This follows from the definition of the curve in section 3.1. The bounds will be in different corners for different configurations of the curve.

In order to ensure a correct result, the window must be adjusted because of the sphere curvature of Earth. This is done using the same function as is used for the bounding boxes of the R-tree. The output of the function is a slightly larger box. Section 4.2 explains why such an adjustment is necessary.

MySQL's range optimizer is a very complex piece of software. Trying to explain its inner workings will most likely be more confusing than beneficial for both the author and reader. It's simply stated that it uses a complex graph structure and red-black trees to represent the possible keys and values that can be used to filter rows. This system accommodates several values for the same key, meaning that the optimizer can be fed virtually any amount of ranges. This is in fact how the implementation is set up, but it only passes in one range, making for an easy extension if one has an algorithm as mentioned in section 3.7.2 that produces several ranges.

3.7 Unimplemented features

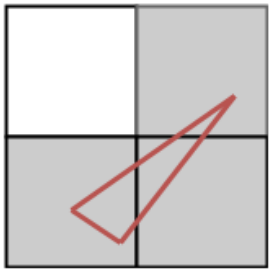
The following three sections briefly explain and motivate three features an implementation of Z-ordering should have in order to be a full-fledged spatial index.

3.7.1 Geometry decomposition

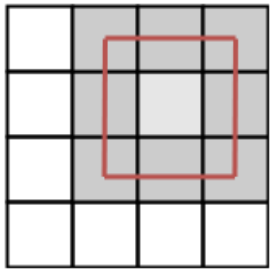
To be able to index a geometry, one must be able to find its decomposition - the set of cells that is intersected by the geometry. Point geometries map to only a single cell, while Linestrings and Polygons can map to many.

The problem lies in finding all the cells intersected from a limited number of points that represent the geometry. This is simple for axes-aligned rectangles such as the query window, but becomes much more difficult when the geometry is more complex and its sides are not axes-aligned.

The essential challenge is to calculate which cells are intersected by the line between two points. The triangular Polygon in figure 10a is defined by the three corner points, located in the lower left and upper right cells. The lower right cell does not contain any points of the



(a) Decomposition of a triangular Polygon.



(b) Decomposition of an axes-aligned rectangular Polygon.

Figure 10: Decomposition of two geometries.

geometry, but it is intersected by a line between two of them. Finding the intersected cells seems fairly simple in a Cartesian coordinate system. In a geographic coordinate system it is not. Section 4.1 shows how the cells' size and shape has a non-linear relationship w.r.t. latitude.

The key implication is that going a distance of X to the east does not always amount to the same number of cells. Close to the Equator it can be two cells, while further north it can be four cells. This is even further complicated by cell edges not being geodesics. See section 4.2 and in particular figure 17.

In addition, one must find the internal cells, those that are not intersected by the border of the geometry but lies inside it. See the cell in figure 10b with a lighter grey color.

An algorithm that decomposes a geometry is necessary for indexing other geometries than points.

3.7.2 Algorithm for efficient range search

A naive implementation of Z-ordering will just represent the query window as a single range of Z-values. However, as is shown in section 4.4 this can lead to a vast amount of Z-values in the range, while only a fraction actually intersects the window. Unfortunately, that is the penalty of linearizing a two-dimensional space into a one-dimensional ordering.

There are, however, improvements to be made. The overarching idea is to represent geometries/areas as sets of ranges instead of a single, large range. One possible method of pruning the search range is using Tropf and Herzog's [31] BIGMIN and LITMAX values used to filter out larger subranges that are in the range but outside the window. A similar approach is presented by Ramsak et al. [24] using the term *next intersection point*.

An algorithm that can reduce the search range may improve query speed.

3.7.3 Multi-level grid

An entry (cell-ID, geometry-ID) is inserted into the index for every cell a geometry intersects. With a high-resolution grid (small cells) and large geometries, the index size can become massive. A way to counteract a large index and still keep a good approximation of the geometry is to have a multi-level grid.

The idea is to have a cell in a higher level be the parent of some cells in the lower level. If all child cells of a parent are part of a geometry's decomposition, the parent cell can be inserted into the index in their stead - without degrading the approximation of the geometry. If most child cells, instead of all, are part of the decomposition, the parent can still take their place, but that representation now includes a larger area than strictly necessary. This creates a trade-off between accuracy of approximations and index size.

A multi-level grid can lead to the index scaling better.

4 Exploring the Z-order grid

This chapter explores various characteristics of the manifested grid resulting from the implementation described in chapter 3.

Section 4.1 explores the varying cell size - and shape - stemming from applying the grid to a spheroid and the consequences it leads to for k nearest neighbor searches and geometry decomposition. Section 4.2 briefly describes how cell edges not necessarily being geodesics complicates finding the correct intersection of geometries. Section 4.3 briefly explains that points along a cell's borders do not necessarily map to that cell. Rounding off, section 4.4 illustrates that small changes to a query window can have significant effects on the range of Z-values that window contains.

4.1 Varying cell size and shape

A *meridian* is a line of constant longitude, stretching from pole to pole, while a *parallel* is a circle of constant latitude parallel to the Equator [34]. The difference in longitude degrees between two meridians is constant, but because they converge towards the poles, the actual distance between two meridians depends on the latitude.

Since the cells of the Z-order implementation are derived from geographic coordinates the cells have a trapezoid shape. Both the southern and northern edges of a cell start and end at the same meridians, but one is closer to a pole and thus will be shorter. As shown in section 4.1.1 this effect is marginal for a cell and it appears rectangle-like, but accumulated, the effect is substantial, leading to varying size and shape as illustrated in section 4.1.2.

Sections 4.1.3 and 4.1.4 explain how this variation may affect nearest neighbor searches and geometry decomposition.

4.1.1 Cell size statistics

By taking as base a cell just above the Equator and repeatedly calling the north neighbor function (see section 3.4, a data set consisting of all cells in a column stretching from the Equator to the North Pole was constructed. This set was inserted into a MySQL database and analyzed using the built-in GIS functions ST_AREA [15] and ST_DISTANCE [16]. Both functions compute the answer with regards to an ellipsoid defined by the semi-major and semi-minor axes. The ellipsoid depends on the spatial reference system [14], which in this case was WGS 84 (also known as EPSG:4326). Further details of the data set is given in appendix B.

The data set was examined, and the following was found to be true for all cells: (1) The west edge distance equals the east edge distance, and (2) the south edge distance does **not** equal the north edge distance.

	Area of cell	South edge	West edge
Minimum	0.36 m^2	0.10 m	303.69 m
Maximum	185 714.72 m^2	611.50 m	306.78 m

Table 1: Extrema of cell area and side lengths.

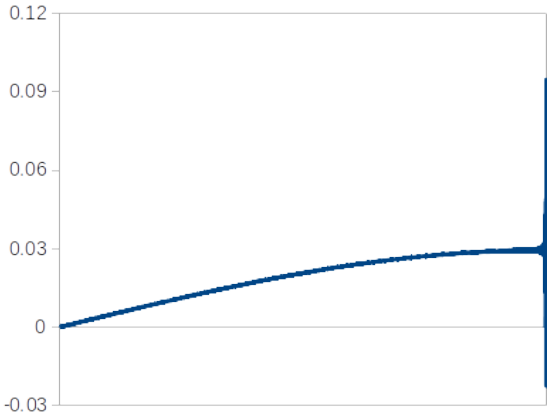
Table 1 lists the extrema of cell width (south edge distance), height (west edge distance) and area. It should be noted that the northernmost cell's area had to be disregarded. The value was far beyond reasonable. One would expect cells close to the pole to be among the smallest, yet it dwarfed all other cells.

Results like this are not unheard of in the GIS world, as systems tend to struggle with geometries crossing over the poles or the (-)180 ° meridian. This is, along with floating-point arithmetic, the presumed reason behind the fluctuating results in figure 12 near the pole as well.

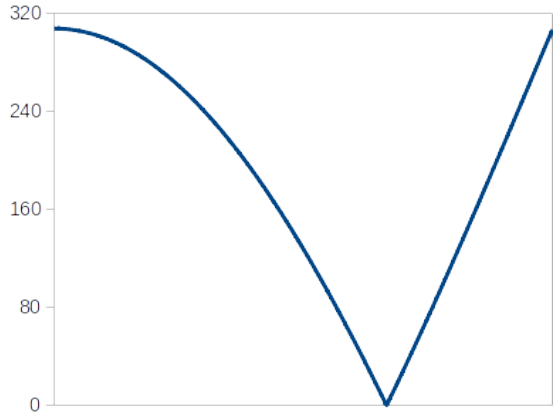
Data points in the graphs of figures 11, 12 and 13 are ordered on the X-axis from the Equator (left) towards the North Pole (right).

Figure 11a shows the difference between the south and north edge of the cells. We deduce two key points from this graph: (1) the difference is marginal, converging to about 3 cm, and (2) the difference increases towards the pole - in a non-linear fashion.

Figure 11b shows the absolute difference between the south and west edge of the cells, giving an indication of how stretched the rectangle (cell) is. At $y = 0$, the sides are equal and the cell has a square shape. To the left, the shape is a wide rectangle, while to the right it is a tall rectangle. Being (partly) derived from the south edge, this metric also has a non-linear relationship.

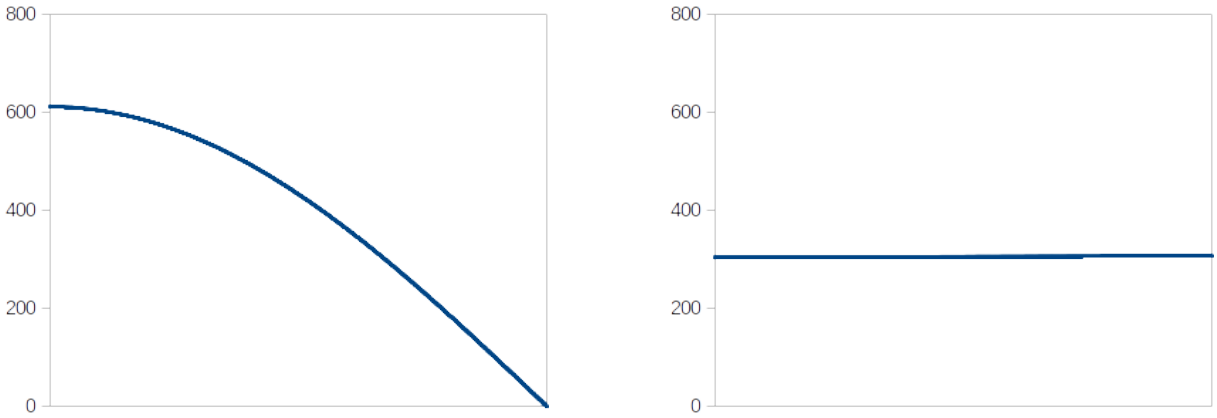


(a) South edge - north edge (m).



(b) Absolute value of south - west (m).

Figure 11: Differences in grid cell side lengths.



(a) South edge (m).

(b) West edge (m).

Figure 12: Grid cell side lengths.

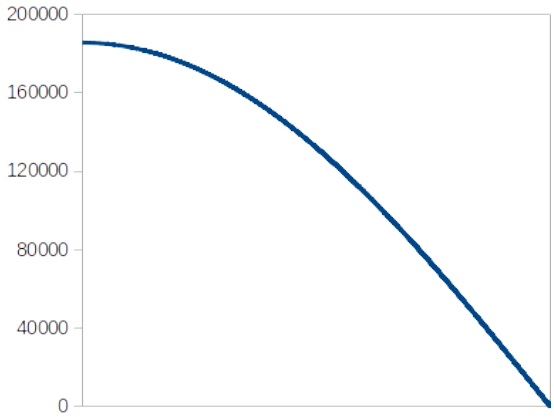


Figure 13: Grid cell area (m^2).

Figure 12 simply shows the distances of the south and west edges. To showcase the disproportionate changes in height and width, the Y-axis is kept at an equal scale. A variant of figure 12b revealing a sigmoid curve shape is presented as figure 27 in appendix B.

Figure 13 shows the area for all the cells in the set. As expected, the area declines towards the pole in accordance with the declining distance of the southern edge in figure 12a.

4.1.2 Cell sizes from Lagos, Trondheim and Longyearbyen

Three cities are used to exemplify the differences in cell size and shape; Lagos in Nigeria, Trondheim in Norway, and Longyearbyen at Svalbard in the Arctic Ocean. Cell measurements are listed in table 2.

We see that the height of the cell (west edge) only has a minuscule change from Lagos to Longyearbyen - a mere 0.96 % increase. On the other hand, we see a substantial difference in the width of the cell (south edge) and consequently the area. Trondheim's cell is only 45.15 % of the width of Lagos' cell, while Longyearbyen's is just 20.61 %. Figure 15 visualizes this

(a) Longyearbyen at Svalbard, Norway (in the Arctic Ocean)

(b) Trondheim, Norway (near the Polar Circle)

(c) Cells in Lagos, Nigeria (near the Equator)

Figure 14: 3x3 grids in Lagos, Trondheim and Longyearbyen.

	Latitude	Area of cell	South edge	West edge
Longyearbyen	78.224 °	38 488 m^2	125.21 m	306.65 m
Trondheim	63.420 °	84 002 m^2	274.36 m	306.16 m
Lagos	6.463 °	184 570 m^2	607.64 m	303.74 m

Table 2: Cell size data from Lagos, Trondheim and Longyearbyen.

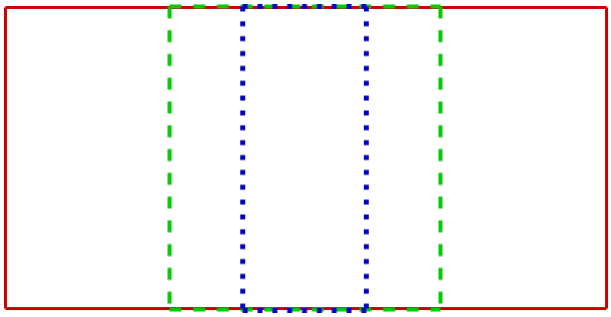


Figure 15: Grid cell sizes from Lagos (red), Trondheim (green dashed), and Longyearbyen (blue dotted).

by constructing rectangles based on the south edge and west edge distances from table 2, and centering them around a common point.

The differences in cells' spatial reach are also striking in figure 14, which shows a 3x3 grid in each of the cities. A 3x1 column in Lagos will cover more ground than a 3x3 grid in Longyearbyen.

4.1.3 Implications for k-NN search

A k nearest neighbor search is a popular use case for applications using spatial data. Intuitively this seems fairly easy to support using a Z-order index, by iteratively searching the next "ring" of cells in an outwards direction from the geometry.

However, due to the rectangle shape of the cells, one would have to search the next ring in order to guarantee a correct result. When k points are found, there still might be points just outside the ring that are closer than points situated in a corner of a cell in the ring.

This problem is even further complicated by the severe differences in cell width. When a cell's width is more than twice (or less than half) of the height, it is no longer sufficient to search one extra ring. See figure 16. A point is found in the first ring (shaded gray), but the closest point is situated directly north and is not even within the next ring.

A correct implementation will have to account for the cell width to adjust how many rows up and down, and columns left and right, a ring should consist of. This is not necessarily trivial because of cell width and height having a non-linear relationship w.r.t. latitude.

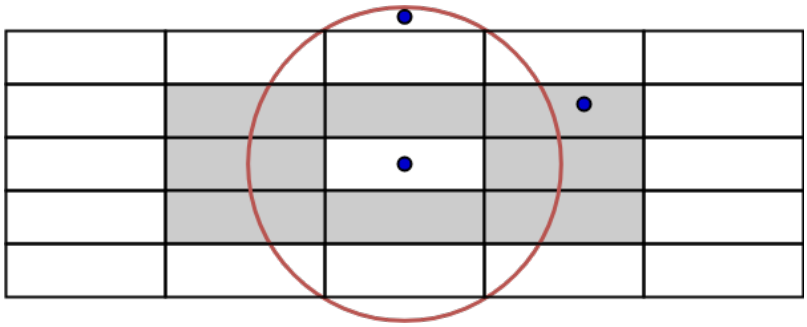


Figure 16: Problematic k-NN search due to varying cell widths.

4.1.4 Implications for geometry decomposition

For spatial indexes with a multi-level grid, such as the one presented in Spatial Indexing in Microsoft SQL Server 2008 [6], it is useful to have some mechanism to control the granularity of the decomposition, i.e., a threshold of how many cells are used to approximate a geometry. Put simply, too coarse a decomposition leads to poor approximations and hence many false positives, while too fine a decomposition leads to a large index reducing its effectiveness. Thus, a threshold guiding the decomposition may balance this trade-off.

The variation in cell sizes does not directly affect this trade-off, as a cell is a cell no matter its size, and the index only consists of (cell-ID, geometry-ID) pairs. But the further away from the Equator, the less space does a cell cover. A geometry close to the Equator needs fewer cells than an equal-sized geometry far from the Equator. It could therefore be problematic if a granularity threshold would limit geometry decomposition far north/south - resulting in poor approximations.

4.2 Non-geodesic cell sides

A geodesic is the curve of the shortest path along a surface between two points. On Earth, the shortest path between two points both having the same latitude will not be a parallel (line parallel to the Equator)[4]. On the northern hemisphere the path will "curve upwards", and downwards on the southern hemisphere.

The edges of grid cells are not geodesics. By definition they follow latitude and longitude borders.

Consequently, the shortest path between the two upper/lower corner points of a cell will **not** follow the edge of the cell[4]. In the right context, this can lead to Linestrings starting and ending in cells at the same latitude (same row in the grid) while some part of the Linestring intersects cells in rows further north/south than the starting and ending cells. This is illustrated, albeit exaggerated, in figure 17. The issue also pertains to Polygons.

[4] Unless the points lie on the Equator

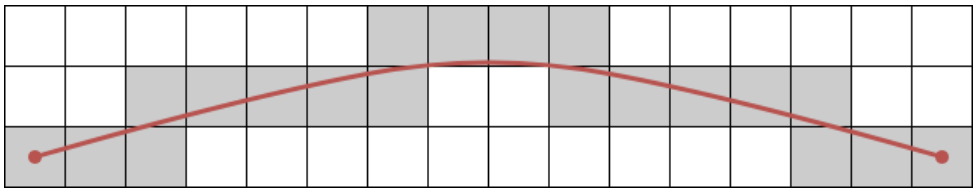


Figure 17: Cell intersection of geodesic Linestring.

ST_DISTANCE (@pt1, @pt2)	273 988.14 m
ST_LENGTH (@ls)	274 071.94 m
Difference	83.80 m

Table 3: Distances of a geodesic and a parallel approximated Linestring.

Exploratory testing using a parallel-approximating Linestring confirmed an expected difference in lengths. The results are presented in table 3.

Starting with the cell containing point (10.4018962 63.4195280), the east neighbor function was used to build a Linestring of 1000 Points, all being the middle point of the cells. The Linestring, along with the starting Point and ending Point, was inserted into variables in a MySQL database, again using WGS 84 as the spatial reference system. Distances was then calculated using the built-in spatial functions ST_DISTANCE [16] and ST_LENGTH [17].

Even though each segment of the Linestring is the shortest path between those points, forcing the string to follow the parallel yields a sum of segments that is slightly longer than the shortest path between the starting and ending point.

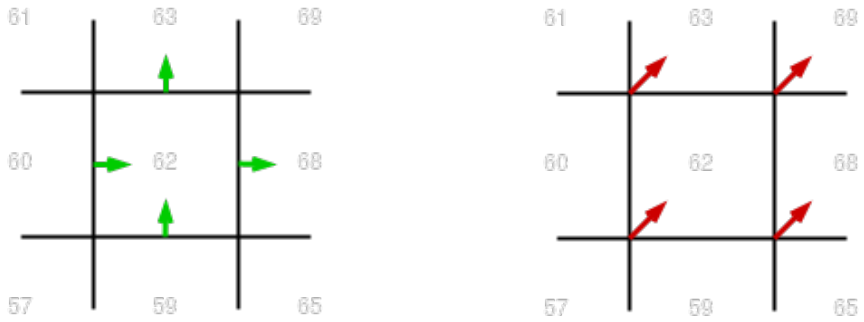
Repeating the test but with the south neighbor function instead yielded, as expected, equal distances.

4.3 Cell borders

The function mapping from Z-value to cell boundaries from section 3.3 gives the edges of the cell. The edges, however, are not necessarily inclusive. Points along a cell's edge defined by two of its corner points can map to an adjacent cell depending on which edge it is.

Figure 18 illustrates for points along a cell's edges which cells they fall into. Points along the northern and southern edges fall upwards, while points along the western and eastern edges fall rightwards. Corner points differ from the rest and fall both up and right, leading to the upper right corner falling into the northeast neighbor.

A cell shares its edge with the adjacent neighbor. This also includes corner points. The upper corners of a cell will be the lower corners of its northern neighbor. Furthermore, a corner point



(a) Middle points of edges. (b) Corner points.

Figure 18: Points along cell edges and the cells they map to.

is shared by all four cells. Nonetheless, a point maps to only one cell - leading to non-inclusive cell edges.

Subtracting the smallest amount possible, 0.00000000000001 (13 zeroes), from a point along the northern and eastern edges pushes it into the cell. Adding another zero to the amount yields no change in the value stored as a C++ double.

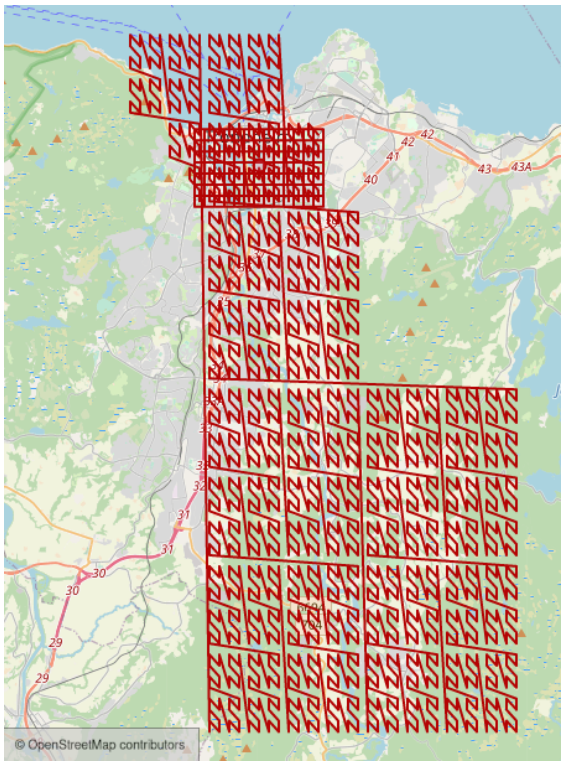
The key takeaway is that cells are disjoint.

4.4 Query windows and Z-value ranges

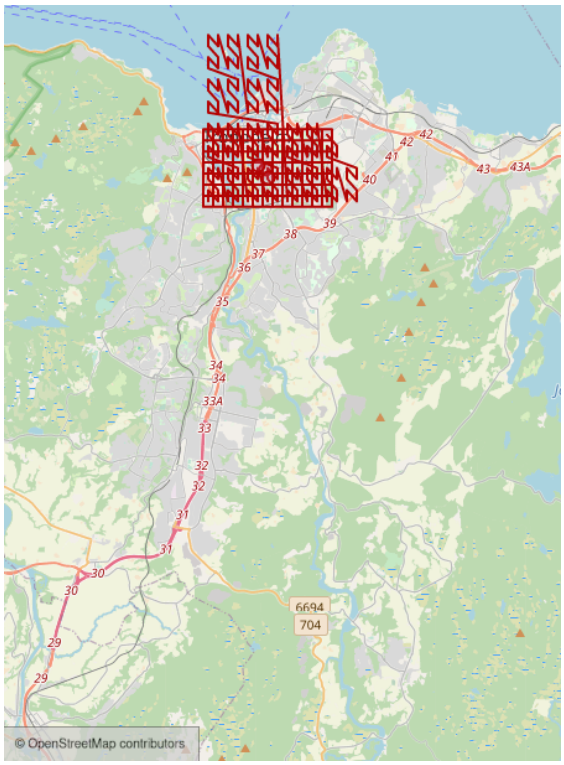
The essence of using space-filling curves for spatial indexing is that one gets a one-dimensional ordering of space, in which cells close to each other in space should also be close to each other in the ordering. So when decomposing a geometry into a set of cells, those cells' values ought to be fairly similar.

A simple implementation of such a spatial index could do a range search between the lowest and highest value, filtering out the vast majority of cells. However, as will be shown in this section, searching the whole range can still lead to a considerable amount of unnecessary cells being regarded as potential candidates, resulting in huge amounts of false positives sent to the refinement step.

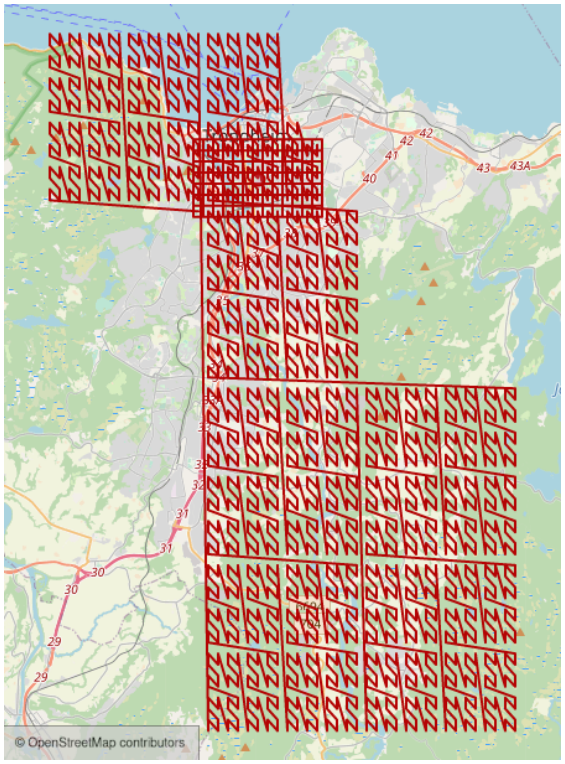
Figure 19 illustrates how large the ranges may be compared to the geometry - and how a little shift in location may result in a vastly smaller/larger range. The figure shows four variants of a query window which is represented by a Polygon that is decomposed into a 13x7 grid. It also shows the Z-order curve drawn from the cell in the window with the smallest Z-value through to the highest. If one takes figure 19a as base, then the window in 19b is shifted one column eastwards, 19c one row southwards, and 19d both one column eastwards and one row southwards.



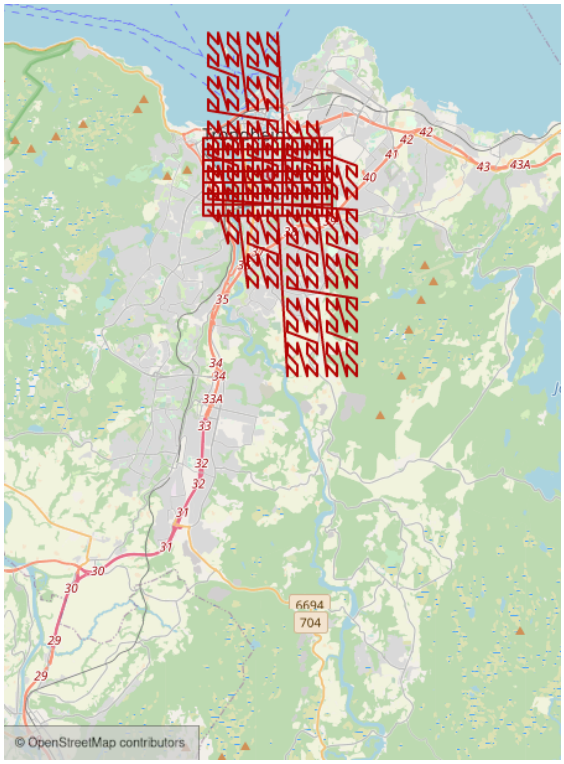
(a) Northwest variant of window.



(b) Northeast variant of window.



(c) Southwest variant of window.



(d) Southeast variant of window.

Figure 19: 4 almost identical query windows and their Z-value ranges.

		Cells in range	Cells in window	Coverage
Northwest	(19a)	1525	91	1676 %
Northeast	(19b)	181	91	199 %
Southwest	(19c)	1693	91	1860 %
Southeast	(19d)	349	91	384 %

Table 4: Coverage statistics of query windows from figure 19

The enormous difference in range sizes is due to the western variants, 19a and 19c, just tipping over a "large dividing line" - the middle vertical line of an N-group containing 4096 cells. This makes them pick up several unnecessary segments, such as the 1024-block to the southeast and most of the 256-block just south of the windows.

Table 4 shows the number of cells in each range from figure 19 and the respective superfluous coverage of the window. Even though most cells in the space are filtered out, searching through 18 times more cells than strictly necessary could substantially impact performance, especially if some of the superfluous area is densely populated with geometries.

The problem was illustrated using a rectangle-shaped Polygon as a query window, but also Linestrings and Polygons of any shape may suffer. Their decomposition may require cells that just barely fall over a line in space where there is a large jump in the ordering, such as the large east-west divider in figure 19. For an N-shaped curve, the geometries most at risk are those with a spatial extent stretching in the northeast/southwest direction. For a Z-shaped curve, it would be the northwest/southeast direction.

5 Experiments and Results

In order to answer the research question stated in section 1.2 empirical data on the performance of the Z-ordering prototype compared to MySQL's R-tree must be produced. Experiment 1 provides this data; query timings for both indexes on a large (4 million) data set of points, for three different window sizes. Experiment 2 and 3 drills further down, providing data that could explain the results from Experiment 1.

The hardware, MySQL Server version, data set, query windows and more that was used to conduct the experiments are detailed in section 5.1.

5.1 Setup

5.1.1 Hardware and OS

The experiments was conducted on a desktop computer with the following hardware:

- CPU - AMD Ryzen 5 3600 6-Core Processor
  - CPU max MHz: 4208
  - CPU min MHz: 2200
  - L1d cache: 192 KiB
  - L1i cache: 192 KiB
  - L2 cache: 3 MiB
  - L3 cache: 32 MiB
- RAM - 16 GB DDR4
  - 2x 8GiB DIMM DDR4 Synchronous Unbuffered (Unregistered) 3600 MHz (0,3 ns)
- SSD - Samsung SSD 970 EVO Plus 500GB
  - Up to 3500 MB/s read advertised
  - Up to 3200 MB/s write advertised

The operating system used was Fedora 33 with the specific release: 5.12.11-200.fc33.x86_64

5.1.2 MySQL Server

The MySQL Server version used was 8.0.24. The source code was downloaded from the public Github repo, with the most recent commit dated on March 19th 2021 having SHA-1 hash 3e90d07c3578e4da39dc1bce73559bbdf655c28c. With Boost version 1.73.0 and the additions of the Z-order implementation a debug build was configured using the following cmake command and built with the ninja command:

```
cmake ../mysql-server/ -DWITH_DEBUG=1 -DWITH_BOOST=~/path-to-
    boost/ -G Ninja
ninja -j8
```

With the following commands the MySQL Server was initiated (1), started (2), and a MySQL command line client started (3):

```
./bin/mysqld --initialize-insecure --datadir=/path-to-db-folder/
./bin/mysqld --datadir=/path-to-db-folder/
./bin/mysql -S/tmp/mysql.sock -uroot
```

Note that although a debug build was compiled the server itself was not started in debug mode, and that no parameters were specified, leading to a server with default configuration.

5.1.3 Data set

The data set used for the experiments is derived from OpenStreetMap's [21] data of Norway. A data dump (.pbf file format [22]) was downloaded from Geofabrik's mirror server [8] at June 11th 2021, containing data up to 2021-06-10T20:21:58Z. The dump was processed using Osmcode's Osmium Library [23] to extract a subset of the data and produce SQL-files used to insert geometries to a database.

The dump contains 152 800 887 *nodes*[5], so a subset was created by only allowing nodes that had at least one accompanying *tag*[5]. The resulting set consisted of 4 044 694 nodes - each with an associated latitude and longitude pair that was converted into a Point geometry.

Points were created using MySQL's ST_GEOMFROMTEXT function, a la:

```
ST_GEOMFROMTEXT("POINT(10.3950581 63.4304961)", 4326, "axis-order
    =long-lat")
```

The set of Points was inserted into the following simple table:

```
create table norway_hastag(
    oid bigint not null,
    g geometry not null srid 4326,
    z int unsigned not null,
    primary key(oid));
```

oid is OpenStreetMaps unique identifier (64-bit integer) of nodes, meaning that it serves as a unique identifier for the Points in the set. *g* is the Geometry column that holds the Points, and is specified to use the spatial reference system with ID 4326 (WGS 84/EPSG:4326). *z* is the Z-value that is derived from the Point's coordinates.

[5] OpenStreetMap idioms, see [20] for definitions.

Indexes was then created with the following queries:

```
create index zorder on norway(z);
create spatial index rtre on norway(g);
```

NB! Creating the B+tree index on the z column took only just over a minute - but creating the R-tree index took almost seven hours. The author suspects that the small memory allocation used by the default configuration of a MySQL server severely hampered the creation time. Increasing allocations through server parameters could alleviate the problem. For clearance, this is not an apples to apples comparison of index creation time as the Z-values were precomputed. Nonetheless, the Z-values were computed as part of extracting data from the .pbf-file and writing to an SQL file - which was timed to 33 seconds.

5.1.4 Query windows

Three sets of query windows were constructed, each as a 10x10 mesh centered in Oslo, Norway. They vary in window sizes, and hence the larger once cover more ground. The measurements of the lower left cell in each mesh are presented in table 5, and the meshes illustrated in figure 20.

This was repeated for Bergen and Trondheim, resulting in a total of nine 10x10 meshes. A 10x10 mesh is referenced as city-size, e.g. Oslo-medium meaning the set of 100 medium-sized Polygons around Oslo.

Each of the experiments uses some subset/combination of Polygons from these meshes. The chosen Polygons were inserted into variables using statements a la:

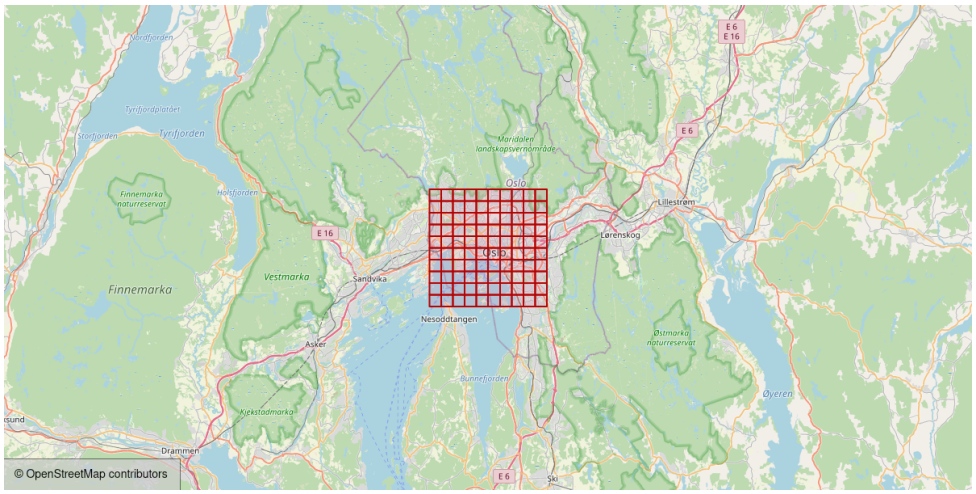
```
set @poly0 = ST_GEOMFROMTEXT("POLYGON((10.3284069 59.7168276,
   10.4084069 59.7168276, 10.4084069 59.7568276, 10.3284069
   59.7568276, 10.3284069 59.7168276))", 4326, "axis-order=long-
   lat");
```

The statements were inserted into an SQL file and executed with:

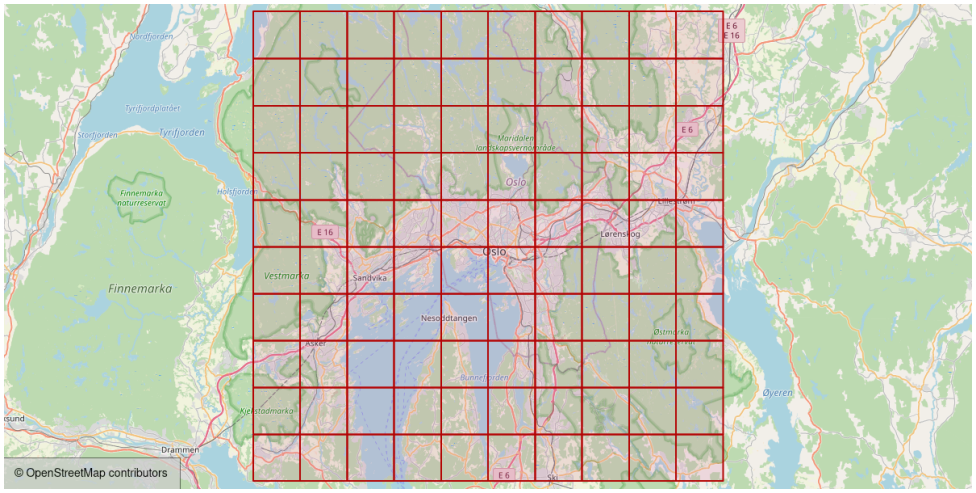
```
source /path-to-file/filename.sql;
```

	Small	Medium	Large
Width	1 120 m	4 502 m	18 330 m
Height	1 114 m	4 456 m	17 824 m
Area	1.25 km^2	20.05 km^2	325.94 km^2

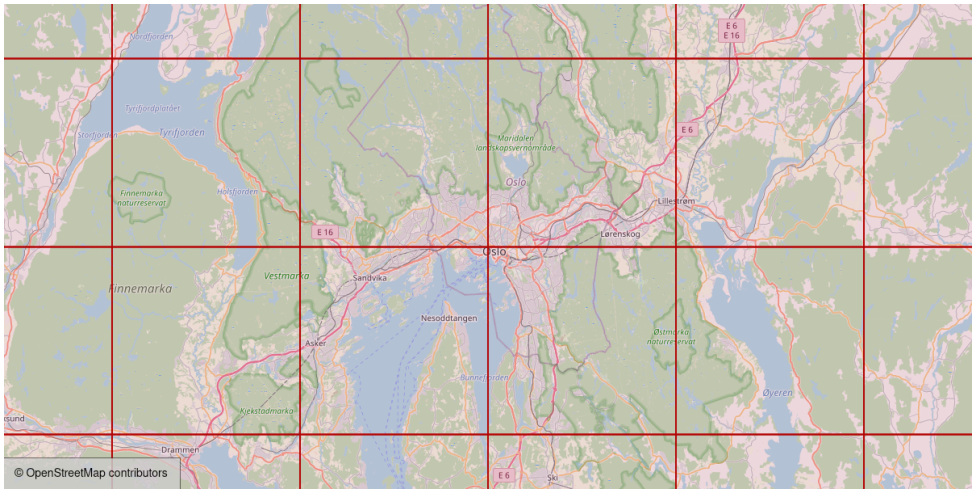
Table 5: Measurements of query windows in the 10x10 meshes centered in Oslo.



(a) Small windows.



(b) Medium windows.



(c) Large windows.

Figure 20: Three 10x10 meshes of query windows centered in Oslo.

5.1.5 Queries

The queries used in the experiments were of the form:

```
1 select oid from norway_hastag force index (zorder) where
    z_contains(@poly0, g, z);
2 select oid from norway_hastag force index (rtre) where
    st_contains(@poly0, g);
```

Note that although the wording of the function is 'contains', the specific usage of the queries in the experiments translates to the window query as explained in section 3.6.

It was discovered during initial testing that four of the 100 z_contains queries on the Oslo-medium set triggered a table scan instead of an index range scan using the indexed z column. This happens when the cost estimation indicates that scanning the whole table should be more efficient. However, rerunning these queries forcing use of the index resulted in significantly improved query times. A table scan took roughly 7 minutes and 40 seconds, whereas forcing the index yielded times of 44, 49, 45 and 46 seconds. Consequently, all queries were run with the *force index* clause.

A number of queries were inserted into an SQL file and executed with the following query from the MySQL command line client:

```
1 source /path-to-file/filename.sql;
```

5.1.6 Output

The output was written to CSV files using MySQL's *TEE* function which appends the content seen in the command line client to a file:

```
1 tee /path-to-file/filename.csv;
```

The statement *"notee;"* stops the writing to file.

5.2 Experiment 1 - Per window size

Experiment 1 was designed to examine the performance of the Z-ordering prototype compared to the R-tree - per window size. The idea was to catch if relative performance varied with window size.

Experiment 1 took all small windows from all three cities and merged them into one window set. For each window a corresponding query was created. The query set was shuffled using C++'s *std::shuffle* [4] function to create a random order to simulate more "natural" usage and mitigate risks of any locality of reference affecting the results. Two query sets were created,

		Mean	Minimum	Max	1st quartile	3rd quartile	Std dev
Small	z_contains	2.03 s	0.00 s	19.98 s	0.21 s	2.14 s	3.33 s
	st_contains	0.12 s	0.00 s	0.73 s	0.02 s	0.17 s	0.14 s
Medium	z_contains	8.93 s	0.00 s	107.52 s	0.45 s	8.79 s	19.44 s
	st_contains	0.33 s	0.00 s	6.65 s	0.04 s	0.32 s	0.65 s
Large	z_contains	21.25 s	0.00 s	242.70 s	1.18 s	14.03 s	42.48 s
	st_contains	1.02 s	0.00 s	18.09 s	0.15 s	0.93 s	1.94 s

Table 6: Statistics of query times, per window size.

one using z_contains and one using *st_contains* - both with the same seed (11) to have identical order.

This was repeated for medium and large windows, with seeds 22 and 33 respectively. The windows was then inserted into the client, the queries executed, and the output written to files.

To minimize the effect of random OS processes interfering with the results the experiment was run after a reboot of the PC and with the System Monitor application as the only other user-started application running. Also, to minimize effects of a cold start, both small query sets were run in advance to "warm up". Then the captured queries where run: large first, then medium, and finally small. In all cases the set with z_contains were run first.

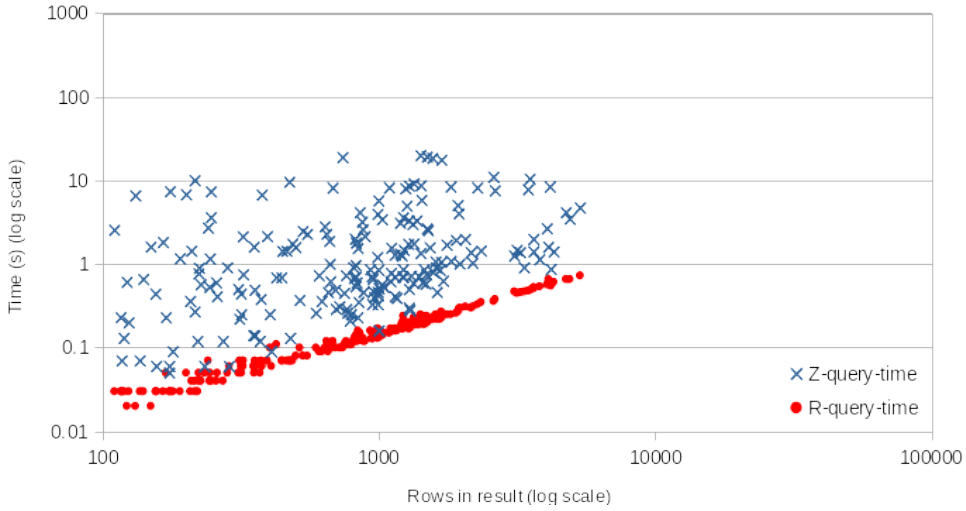
5.2.1 Results

Statistics of the query times are presented in table 6. For all three window sizes and all statistics the R-tree is better by one order of magnitude. The only positive for the Z-ordering prototype from these statistics is the minimum query time of 0.00 seconds. By inspection it was found that these times were achieved for some queries where the result set was empty. However, this was not the case for all empty results, and a high of 27.48 seconds was found.

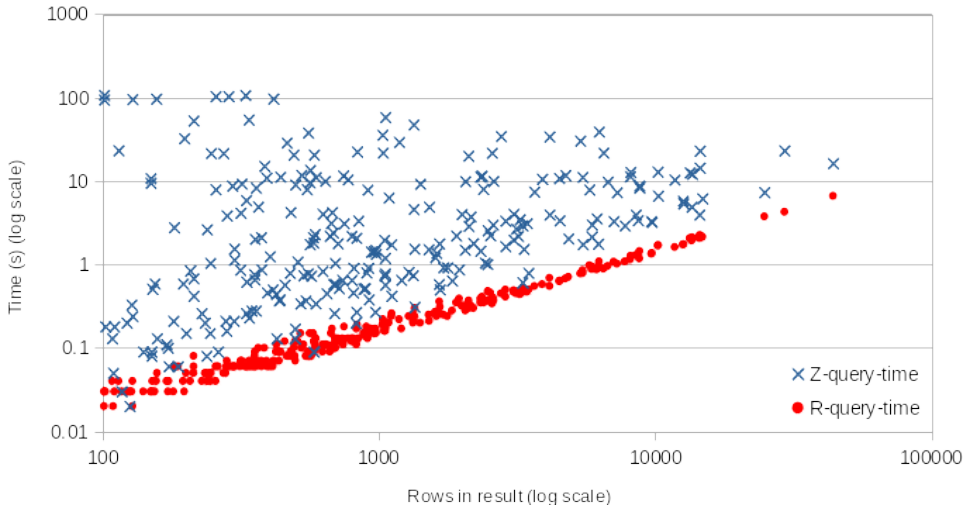
Figure 21 shows plots of the query times (per window size) w.r.t. the number of rows in the result set. The plots confirm the inference from table 6: the R-tree outperforms the Z-ordering. However, in several cases the Z-ordering is close, and in a few cases comparable. The plots also show that while the R-tree's query times looks to be consistent with the amount of rows in the result the Z-ordering query times are all over the place.

The key takeaways are that for some queries the prototype is competitive - but it's generally outperformed, and that its query times are wildly inconsistent.

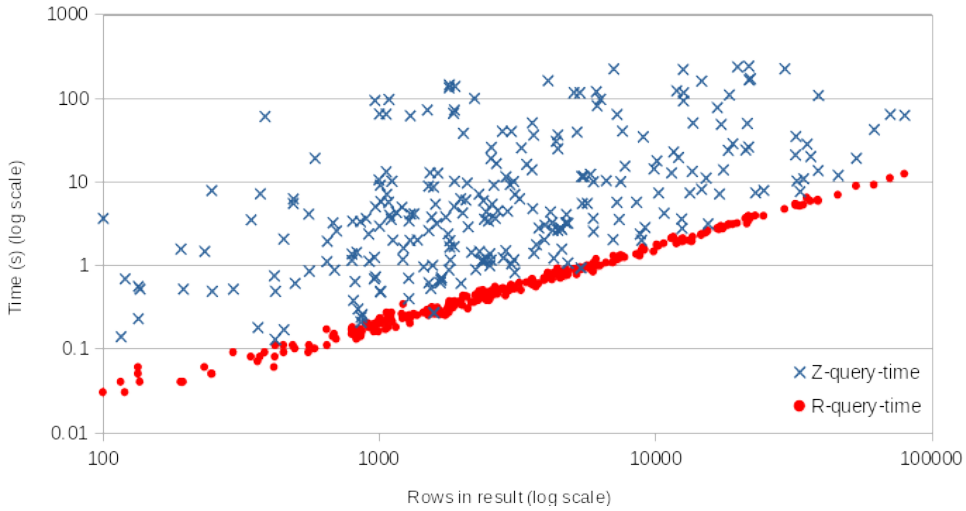
Note that the plots' X-axis starts at 100 and ends at 100 000. Only one query had over 100 000 rows in the result. The queries with under 100 are shown in figure 22.



(a) Small windows.



(b) Medium windows.



(c) Large windows.

Figure 21: Plots showing timing data, per window size.

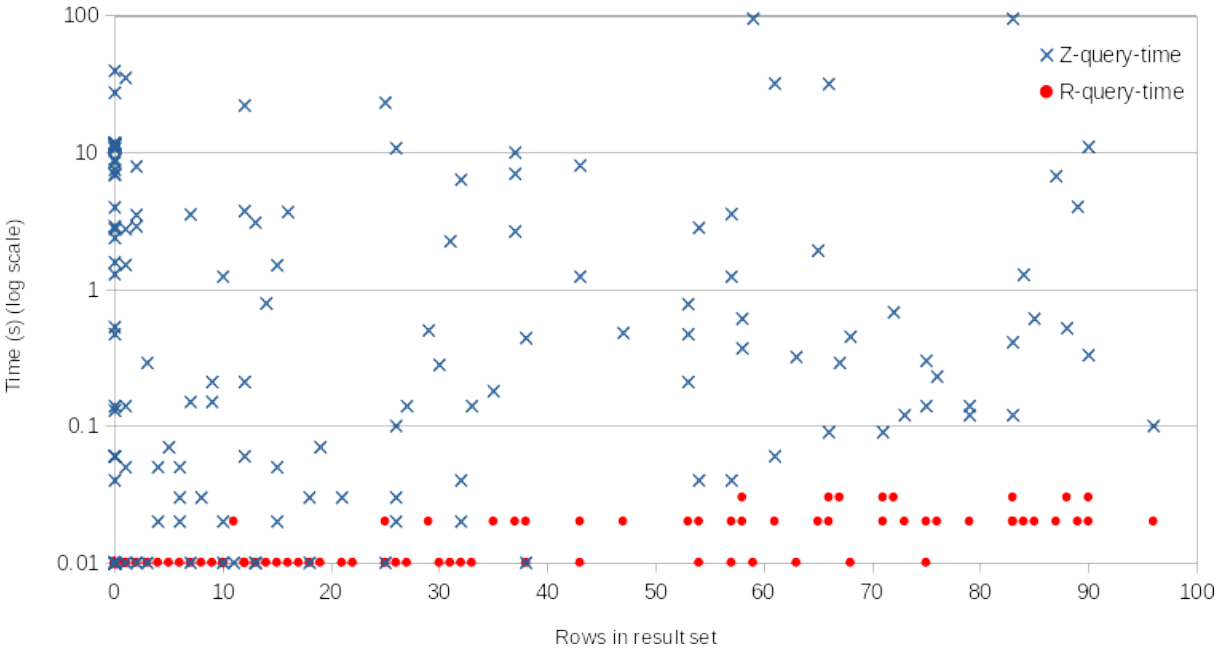


Figure 22: Plot showing timing data for queries with less than 100 rows in the result.

5.3 Experiment 2 - EXPLAIN ANALYZE

Experiment 2 was designed to further examine the seemingly random query times of the Z-ordering prototype.

Using MySQL's EXPLAIN ANALYZE [13] functionality one can obtain some metadata about the execution of a query. It has some overhead, so the timing should be slightly higher than running just the plain query. For this experiment the query windows used was the Oslo-medium set, and queries on the following form:

```
1 explain analyze select oid from norway_hastag force index (zorder
    ) where z_contains(@poly0, g, z);
2 explain analyze select oid from norway_hastag force index (rtre)
    where st_contains(@poly0, g);
```

Table 7 exemplifies the relevant data that was extracted from the query output. The motivation for this experiment was to examine how query time was affected by (1) the estimated number of rows, (2) the actual number of rows returned from the iterator (index) and (3) the actual number of rows in the result.

	Query time	Estimated rows	Index rows	Result rows
z_contains	1.38 s	18 168	10 095	2866
st_contains	0.51 s	1	2 867	2 866

Table 7: Example metadata extracted from EXPLAIN ANALYZE query output.

During preliminary testing it was discovered that the R-tree estimations were 1 for almost all windows of all three sizes. For a few of the large windows the estimation was 2, 3, or 4. Additionally, the number of rows returned from the R-tree index was for most windows equal to the number of rows in the final result. If not equal the difference was typically only a few rows. Consequently, looking at this data for the R-tree is arguably not interesting and only Z-ordering is considered further.

5.3.1 Results

Figure 23a corresponds to figure 21b, but only for the windows around Bergen and with a non-logarithmic Y-axis. Along with the data shown in Experiment 1 it substantiates two key observations: (1) the R-tree timings seem consistent w.r.t. the size of the result and (2) increasing with the size. In contrast, query timings from the Z-order index seem to be inconsistent and uncorrelated to the result size.

Figure 23b shows the number of rows estimated and returned from the Z-ordering index. The graph reveals two interesting observations. Firstly, the estimation seems uncorrelated to the size of the result - which is counter-intuitive to what one would expect. Secondly, the graph shows a striking resemblance to figure 23a.

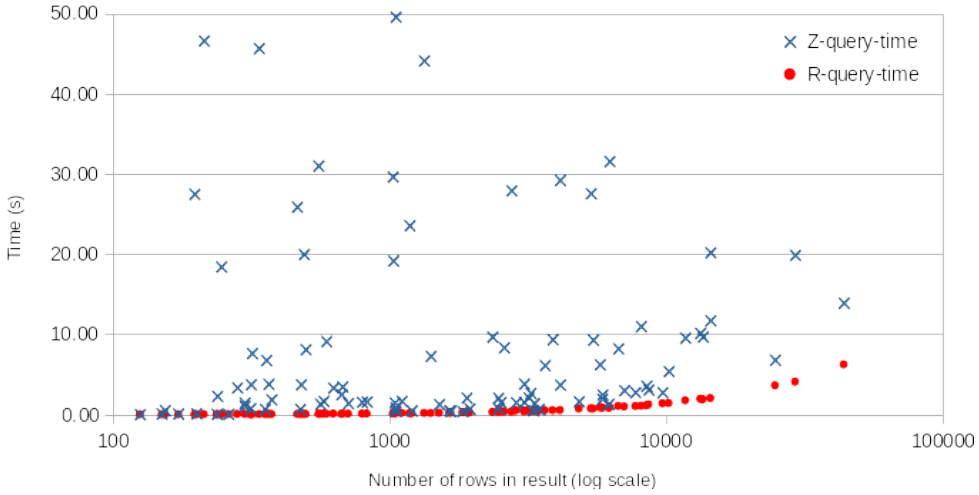
Figure 23c explains the resemblance of figure 23a and figure 23b as it clearly shows a linear trend of number of rows against query time. The obvious answer is that the rows returned from the index are compared using the expensive spatial function. The more rows that pass through, the longer the query will take. Furthermore, the graph shows that the ratio between estimated number of rows and number of rows returned by the index seem consistent, albeit off by some linear factor.

Figure 24 shows all the query times from Experiment 1 in one plot, along with a few benchmark timings (black squares and interpolated line). The benchmarks come from the following statements, with the query repeated with limit 1 000, 10 000, 100 000 and 1 000 000:

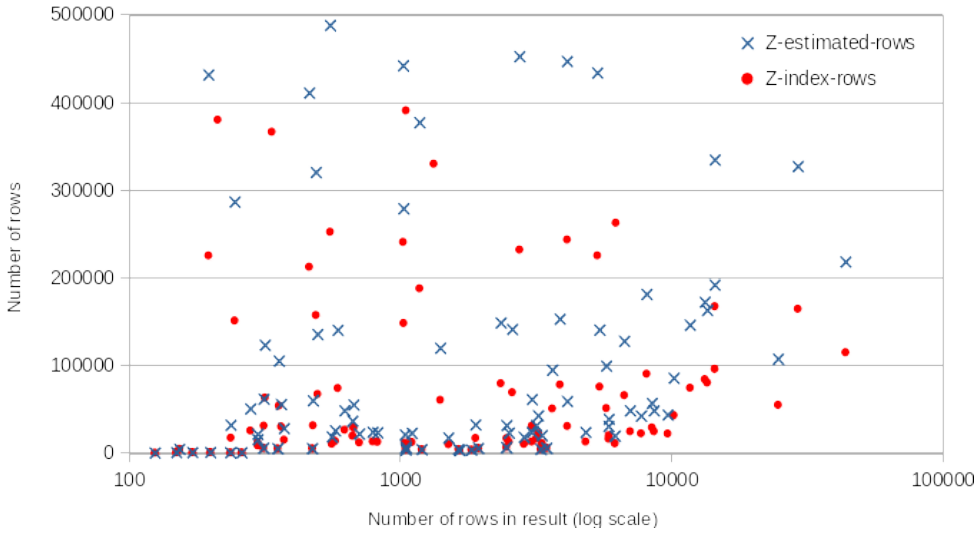
```
1 set @poly = st_geomfromtext('polygon((0 50, 50 50, 50 89, 0 89, 0
      50))', 4326, 'axis-order=long-lat');
2 select oid from norway_hastag ignore index (rtre, zorder) where
     st_contains(@poly, g) limit 100;
```

The Polygon is huge in order to have any Point in the data set be contained by it. The queries are forced to ignore the index, and limited to some amount of rows. The intent was to get timings of queries where rows are simply fetched, run through the spatial function, and then sent to the result set.

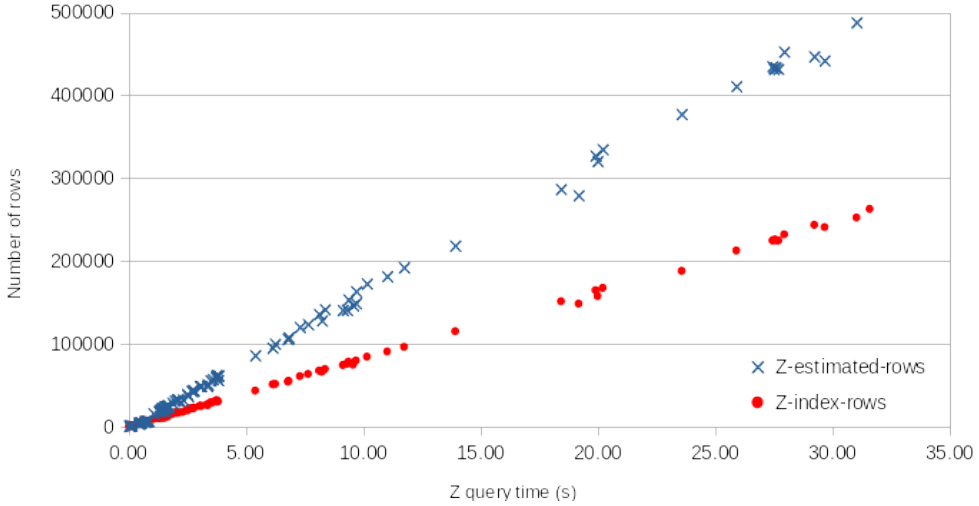
The idea behind the benchmarks is to provide some reference to how long processing the queries takes. The implication being that most of the query time is spent on processes other



(a) Query time vs number of rows in the result.



(b) Estimated and actual number of rows returned from the index vs number of rows in result.



(c) Estimated and actual number of rows returned from the index vs query time.

Figure 23: Plots showing data from EXPLAIN ANALYZE on the Oslo-medium set.

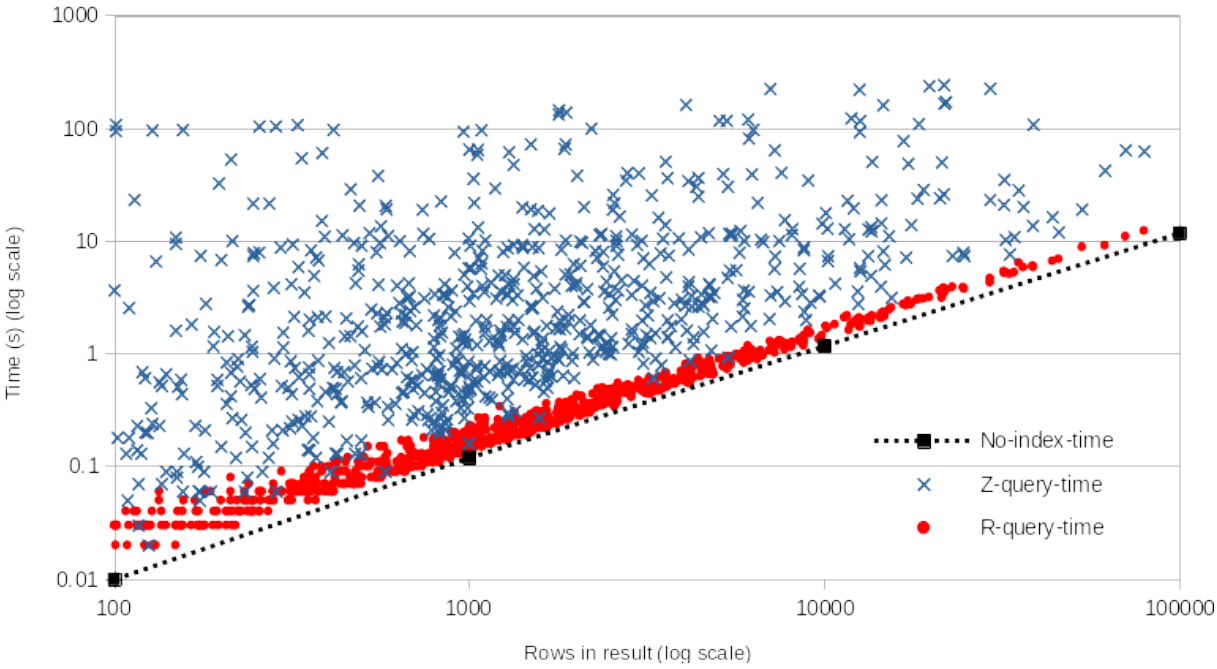


Figure 24: Plot showing timing data and no-index benchmark.

than using the index to pre-filter rows. It must be emphasized that the spatial part of the query is not necessarily the only process that contributes to higher times when the row count increases. For example, the physical retrieval of a row from memory to registers obviously contributes to increased time. This differentiation is not explored.

The major takeaway from this experiment is that for the Z-ordering prototype, the number of rows returned from the index is clearly correlated with query time (figure 23c) - but in spite of that, it seems to be uncorrelated to the number of rows in the result. The prototype seems to have a very poor, or at least inconsistent, filtering accuracy.

5.4 Experiment 3 - A closer look at Bergen-medium

Experiment 3 was designed to reveal the cause of very high query times despite a low amount of rows in the result set.

It was discovered during processing of the results for Experiment 1 that some results from the Bergen-medium window set stood out - and exactly 10 of them. Hence the set was rerun using EXPLAIN ANALYZE as in Experiment 2.

5.4.1 Results

Table 8 shows the collected data from the output for the 10 windows. Two extra windows were included, the ones just west and east of @berg_medium_8; 7 and 9. The data shows a shocking mismatch between the amount of rows returned from the index and the rows in

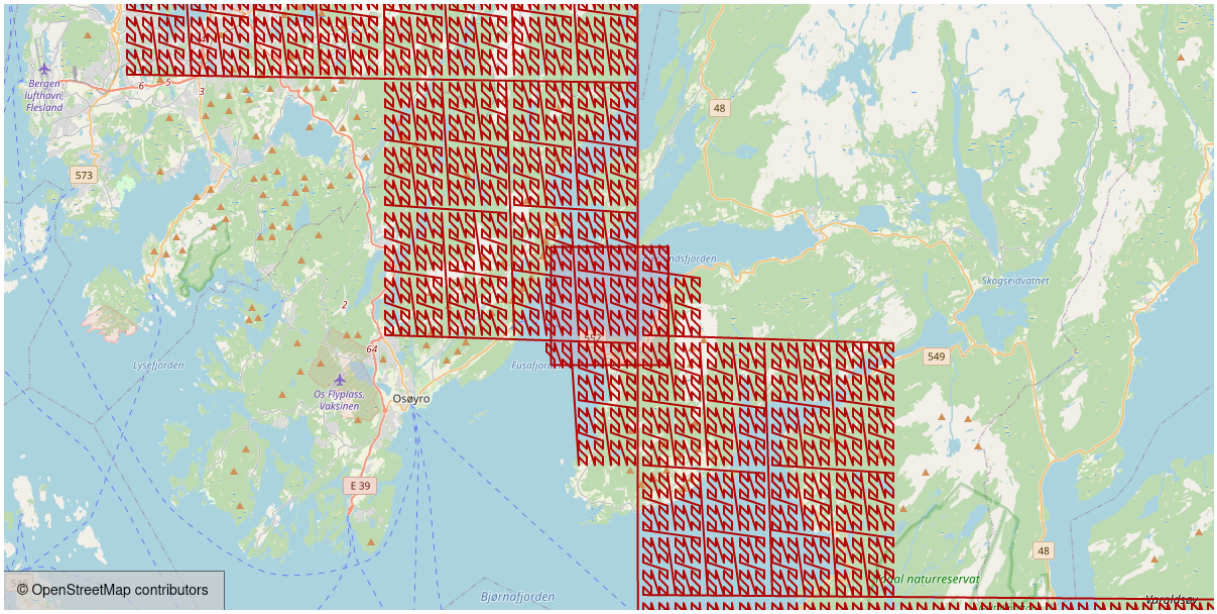
Window	Query time	Estimated rows	Index rows	Result rows
@berg_medium_8	95.47 s	1 451 268	724 792	285
@berg_medium_18	96.00 s	1 364 496	723 319	328
@berg_medium_28	95.53 s	1 343 006	725 015	101
@berg_medium_38	87.06 s	1 306 806	655 661	156
@berg_medium_48	86.88 s	1 306 806	656 072	59
@berg_medium_58	86.22 s	1 264 020	652 108	414
@berg_medium_68	94.24 s	1 376 728	709 651	256
@berg_medium_78	86.96 s	1 193 892	648 861	101
@berg_medium_88	86.16 s	1 290 740	648 639	128
@berg_medium_98	86.08 s	1 290 912	649 555	83
@berg_medium_7	0.38 s	2436	2436	519
@berg_medium_9	0.19 s	1356	1356	122

Table 8: Query metadata from a subset of the Bergen-medium set.

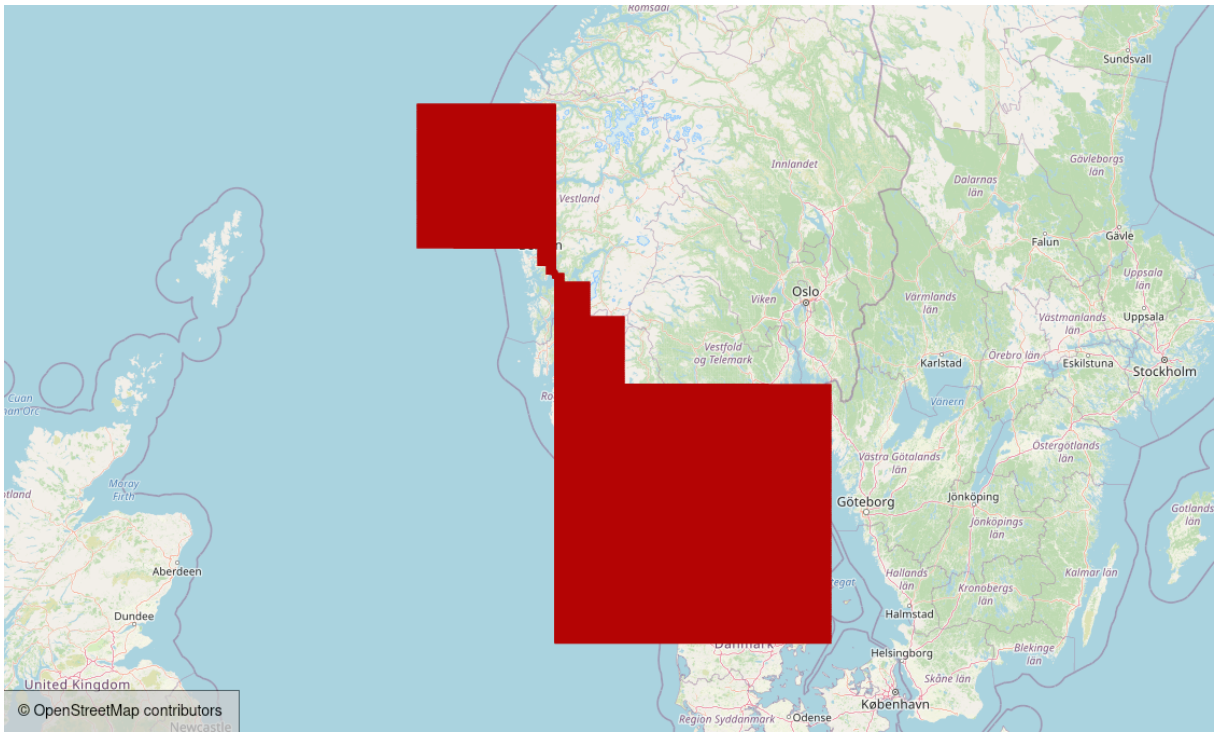
the actual result. The amounts of estimated rows are also high, yet they are consistent with figure 23c: roughly double of index rows.

Figure 25 reveals the reason for the mismatch. As explained in section 4.4, when a window crosses a "large divider" of the N-curve its Z-value range can end up covering several times as many cells as there are cells in the window. This is what happened for these 10 windows.

For clearance, the windows all belong to the same column in the 10x10 mesh and hence cross the large divider. @berg_medium_7 and @berg_medium_9 fall into each side of the divider and the queries on those two windows are consequently two orders of magnitude (!) faster. Nevertheless, the filtering accuracy is still quite off the actual result.



(a) Zoomed in.



(b) Zoomed out.

Figure 25: Polygon @berg_medium_8 and its Z-value range.

6 Evaluation

Section 6.1 evaluates the results of experiments 1, 2 and 3, while section 6.2 reasons about the R-tree's curiously close-to-perfect filtering accuracy.

6.1 Experiment results

Experiment 1 clearly shows that MySQL's R-tree outperform the Z-ordering prototype. The statistics of table 6 tell that averaged the R-tree is better by an order of magnitude. Interestingly, figure 21 and 22 shows that for some queries the prototype does perform well. Yet, in large they show that the prototype's query timings are significantly higher - and seemingly random, uncorrelated to the number of rows in the result.

An interesting observation from figure 21 is that query timings seem to be fairly uncorrelated to window size, and it's the number of rows in the result set that's the driver. A larger window is more likely to contain more points, so that can explain why timings shift a bit up- and rightwards for the medium and large windows.

It's intuitive that the more rows that are returned from the index, the longer the query will take because it has to do the expensive spatial computations for more rows. Experiment 2 confirms this is the case, as shown in figure 23c. More important, the experiment also shows that the number of rows returned from the prototype's index seems to be uncorrelated to the number of rows in the result. The inference being that the filtering accuracy of the index is very inconsistent and generally poor. On the other hand, as shown in figure 26, the R-tree is very consistent. The plot is made using the same data as Experiment 2.

The cause of the inconsistent filtering is revealed by Experiment 3. It showcases that the prototype is susceptible to windows crossing large dividers. This was showcased in section 4.4. Although not directly tested the results from the experiments, and in particular figure 26, imply that the 'unluckyness' of crossings is widespread and substantial.

6.2 R-tree filtering accuracy

From figure 26 it seems as if the R-tree is almost too consistent - and too precise. As mentioned in section 5.3 the amount of rows returned from the index either matched the amount of rows in the result exactly, or only differed by a few rows.

A plausible explanation stems from the experiments only using Point data. Unlike the other Geometry types points does not have a spatial extent, and hence does not suffer the problems associated with approximations. With no extent a Point cannot lie both inside and outside of a bounding box. It's either inside the box - or outside. This results in a selection from the tree that is in almost all cases correct - i.e. no false positives. With Linestring and Polygon

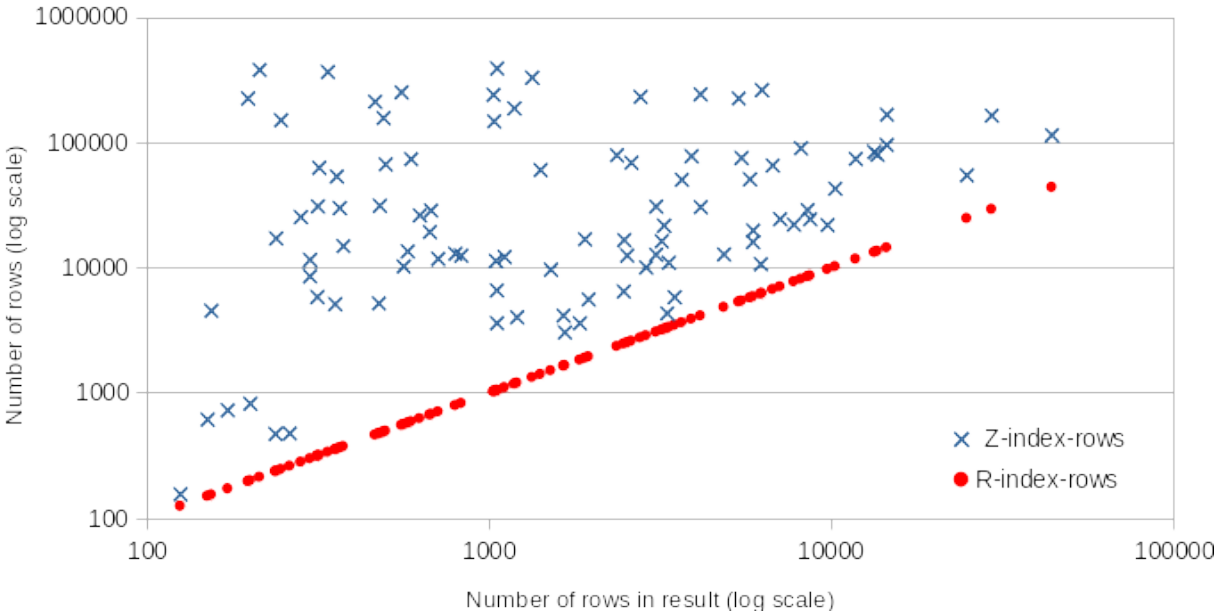


Figure 26: Plot showing no. of rows returned from the index vs no. of rows in the result.

data one should expect far more false positives because of the rectangle approximations.

The slight deviation of row count is suspected to come from the adjustment of bounding boxes, the increase towards poles to adjust for sphere curvature. Most likely the query window's bounding box sometimes contains points that are not actually inside the window.

As for the R-tree's estimation being only one row - or sometimes a few for the large windows - the cause lies in the formula. The estimation should approximately match [26]:

```
(number of rows in table * area of search box) / area of root node's box
```

Arguably 4 million points is not that much compared to the size of mainland Norway. But the data points are not constricted to the mainland and can extend far into the Norwegian sea, the Barents sea, and also around Svalbard. The area of the root node's bounding box is consequently massive.

7 Discussion

The following sections discuss various findings and observations from the thesis.

7.1 Problematically large search ranges

Seeing the results from the experiments it's, in the author's opinion, obvious that for Z-ordering to have the possibility of being a viable spatial index it needs an algorithm that reduces the range of Z-values that must be searched. This issue was briefly mentioned in section 3.7.2. Without such an algorithm the problem presented in section 4.4 manifests as it did in the experiments - leading to a spatial index with poor filtering accuracy and hence poor query timings.

Also, seeing the high query timings and knowing that the spatial functions are expensive one could be inclined to think that higher filtering accuracy is more important than fast filtering. In other words, a slower but more precise index will likely better than a fast but imprecise one.

The experiments were limited to window queries. Similar functionality could be implemented for other shapes than an axes-aligned rectangle; represent the query geometry as one range of Z-values. However, the problems discovered for this scheme would also be present for the other shapes as well.

7.2 Indexing time

The glimmer of light for the prototype is that it performed significantly better than the R-tree at time used for indexing. This was not conducted as an experiment, but the difference reported in section 5.1.3 is not to be mistaken with a difference of two orders of magnitude.

The mapping process used in the prototype probably even has room for improvements. At least for schemes that interleave the coordinates directly. An AWS blog post [29] mentions a scheme using floating point numbers which reinterprets the floating point numbers as sign-magnitude integers. The idea is that the relative order of the numbers are preserved. However, special handling is required in order to handle negative numbers.

7.3 Neighbor functions

The neighbor functions described in section 3.4 were useful during the thesis for exploring and testing various things, but were not used in the Z-ordering prototype that was tested. As mentioned it is one of the claimed benefits of the Z-order curve.

However, if it turns out that they provide little practical benefits for spatial indexing it might be more beneficial to trade finding neighbors fast with the Hilbert curve's better clustering. On that note, it would definitely be interesting to see how many rows the prototype index

would return if the Hilbert curve would have been used instead. According to the literature the Hilbert curve should be better, but seeing if the difference would make any practical difference is very desirable.

7.4 B+tree cost estimation

One of the benefits of using an SFC for spatial indexing is reuse of code. In the prototype's case this also includes the cost estimation of the B+tree. From the data presented in figure 23c it seems like the estimation is consistent w.r.t. the number of rows returned from the index, but is off by some constant.

Without deeper knowledge of the optimizer and cost estimation it is difficult to evaluate whether this is unusual or expected behavior, and thus if this can be fixed with a simple adjustment or would require a larger modification of B+tree's cost estimation.

8 Conclusion

Section 8.1 summarizes the findings and contributions of the thesis, while section 8.2 suggests how it can be extended through future work. Section 8.3 gives a brief overview of the most closely related work.

8.1 Contributions

The contributions of the thesis can be summarized by the goals stated in section 1.2.

8.1.1 Index performance

Through the experiments the Z-ordering prototype's performance is compared to MySQL's R-tree, for window queries on Point data. The key takeaway is that without an algorithm to speed up range search by pruning the search range the index exhibits inconsistent and mostly poor filtering accuracy.

Given this finding the implementation should be regarded as incomplete and comparing the timings of the two indexes is arguably not very meaningful. Hence, goal 1 is unfortunately not achieved. Nevertheless, it's in the author's opinion valuable to show that the large search ranges lead to poor filtering accuracy and a range search pruning algorithm should be a vital part of a Z-ordering as a spatial index.

The thesis also provides anecdotal evidence that the prototype's index creation time is significantly better than the R-tree's.

8.1.2 Explaining the implementation

Chapter 3 provides a detailed description of the Z-ordering implementation and hence achieves goal 2. This description can serve as inspiration or reference to others who can build upon the work.

8.1.3 Grid exploration

Probably the most unique contribution of the thesis, in the author's opinion, is the presentation of various aspects of the manifested grid resulting from the Z-order curve being applied to the Earth. The problems arising as a consequence that are discussed are to the author's knowledge not presented as thoroughly and clearly, if at all, in other resources.

The varying cell sizes and non-geodesic cell sides will be problematic for other space-filling curves as well. The problems do not stem from the particular way the grid cells are ordered.

8.2 Future work

First and foremost, rerunning the experiments with a search range pruning algorithm implemented could provide interesting results. This would be the natural next step to improve the knowledge base of Z-ordering as a spatial index. Other obvious candidates are the other two aforementioned missing features; geometry decomposition and multi-level grid. With an index that can index other query types such as the intersection query can also be tested.

It would also be fascinating to see how cost estimation affects the performance of Z-ordering and R-trees. If one method's estimation being superior would have a practical difference at all on query times. This is given that both implementations are well implemented and has a decently working cost estimation algorithm.

Given the extremely long time taken to create the R-tree index for the experiment data set, compared to the sum of Z-value computations and B+tree creation, it would be interesting to see experiments delving deeper into this theme. One index might be more suited for rapidly changing or ephemeral data sets. A side road on this path could look into interleaving floating point numbers to obtain the Z-value instead of first mapping them to an integer range as is done in the thesis' implementation.

Although space-filling curves have been extensively studied and the Hilbert curve is generally preferred it would be interesting to see the practical difference the better clustering will have on query timings. The same can also be said for the R*tree vs the R-tree.

8.3 Related work

The research closest to the thesis was done by Kothuri et al. [12] comparing quadtree and R-tree implementations in Oracle Spatial. They found that for the most part the R-tree outperformed the quadtree at query timings.

Fang et al. [18] gives an introduction to how spatial indexing was supported in Microsoft SQL Server 2008. They present an implementation using the Hilbert curve and a multi-level grid. Although being selectively detailed, they do present their geometry decomposition in a fairly thorough manner.

The documentation for the present (as of 2021) version of Microsoft SQL Server [18] provides a more DBMS user-oriented presentation of spatial indexing, but does include some interesting information on how the geodetic space is projected onto a plane.

Samet's textbook [27] and Gaede and Günther's survey paper [7] may serve as an entry to the closely related subject of multi-dimensional indexing.

The following sources are not academic texts, however, they provide valuable information about the subject:

- S2 Geometry [9]: A spatial index where the Hilbert curve is used to order cells along the six sides of the *unit cube* that is then projected onto a sphere. The most interesting part about this index is that the cell sides are geodesics.
- Uber H3 [3]: A hierarchical spatial index using hexagons to tile the space. This stands in contrast to the typical rectangular grids.

A Algorithms for Z-ordering

uint_fast32_t is a type definition used to select the fastest data type that can represent an unsigned integer of 32 bits based on the platform. It is used for interoperability with the Libmorton library.

```
// Calculate value of bit, and modify corresponding boundary
char get_bit(const double coordinate,
             double &lower_bound,
             double &upper_bound)
{
    char res;
    double middle = (lower_bound + upper_bound) / 2;

    if (coordinate < middle)
    {
        res = '0';
        upper_bound = middle;
    }
    else
    {
        res = '1';
        lower_bound = middle;
    }

    return res;
}


uint_fast32_t zvalue_from_coordinates(const double lon,
                                      const double lat)
{
    std::string zstring = "";
    double lat_lower = -90;
    double lat_upper = 90;
    double lon_lower = -180;
    double lon_upper = 180;

    for (int i = 0; i < 32; i += 2)
    {
        zstring.push_back(get_bit(lon, lon_lower, lon_upper));
        zstring.push_back(get_bit(lat, lat_lower, lat_upper));
    }

    // Interpret the string as an unsigned integer
    uint_fast32_t z = std::bitset<32>(zstring).to_ulong();
    return z;
}
```

Listing 1: Mapping from coordinates to Z-value (cell).

```
void cell_boundaries_from_zvalue(const uint_fast32_t zvalue,
                                 double &lon_lower,
                                 double &lon_upper,
                                 double &lat_lower,
                                 double &lat_upper)
{
    std::string zstring = std::bitset<32>(zvalue).to_string();
    lon_lower = -180;
    lon_upper = 180;
    lat_lower = -90;
    lat_upper = 90;

    for (int i = 0; i < zstring.size(); i += 2)
    {
        if (zstring[i] == '1')
        {
            lon_lower = (lon_lower + lon_upper) / 2;
        }
        else
        {
            lon_upper = (lon_lower + lon_upper) / 2;
        }

        if (zstring[i + 1] == '1')
        {
            lat_lower = (lat_lower + lat_upper) / 2;
        }
        else
        {
            lat_upper = (lat_lower + lat_upper) / 2;
        }
    }
}
```

Listing 2: Mapping from Z-value (cell) to coordinate boundaries.

```
uint_fast32_t neighbor_north(const uint_fast32_t &zvalue)
{
    // NB, adds to the opposite coordinate to accomodate the
    // different rotation of the grid (z vs n shaped curve).
    uint_fast16_t lon;
    uint_fast16_t lat;
    libmorton::morton2D_32_decode(zvalue, lon, lat);
    return libmorton::morton2D_32_encode(lon + 1, lat);
}
```

Listing 3: Finding the Z-value of northern neighbor cell.

B Cell measurements

The data set analyzed in section 4.1.1 consisted of a column of cells stretching from the Equator to the North Pole. The column was created by taking a cell just above the Equator, containing point (0.0000001 0.0000001), and repeatedly using the north neighbor function (see section 3.4). For each cell the corner points where calculated using the mapping function (see section 3.3) and used to construct a Polygon of the cell.

```
create table cellsizes(
    z int unsigned not null,
    cell geometry not null srid 4326,
    ll geometry not null srid 4326,
    lr geometry not null srid 4326,
    ur geometry not null srid 4326,
    ul geometry not null srid 4326,
    primary key(z));

alter table cellsizes add column west_edge double generated
   always as (st_distance(ll, ul)) stored;
alter table cellsizes add column south_edge double generated
   always as (st_distance(ll, lr)) stored;
alter table cellsizes add column east_edge double generated
   always as (st_distance(lr, ur)) stored;
alter table cellsizes add column north_edge double generated
   always as (st_distance(ul, ur)) stored;
```

Listing 4: Creating the table.

```
INSERT INTO cellsizes VALUES
    (3221225472, ST_GEOMFROMTEXT("POLYGON((0.0000000 0.0000000,
   0.0054932 0.0000000, 0.0054932 0.0027466, 0.0000000 0.0027466,
   0.0000000 0.0000000))", 4326, "axis-order=long-lat"),
   ST_GEOMFROMTEXT("POINT(0.0000000 0.0000000)", 4326, "axis-order
   =long-lat"), ST_GEOMFROMTEXT("POINT(0.0054932 0.0000000)",
   4326, "axis-order=long-lat"), ST_GEOMFROMTEXT("POINT(0.0054932
   0.0027466)", 4326, "axis-order=long-lat"), ST_GEOMFROMTEXT("
   POINT(0.0000000 0.0027466)", 4326, "axis-order=long-lat")),
    .
    .
    (3579139413, ST_GEOMFROMTEXT("POLYGON((0.0000000 89.9972534,
   0.0054932 89.9972534, 0.0054932 90.0000000, 0.0000000
   90.0000000, 0.0000000 89.9972534))", 4326, "axis-order=long-lat
   "), ST_GEOMFROMTEXT("POINT(0.0000000 89.9972534)", 4326, "axis-
   order=long-lat"), ST_GEOMFROMTEXT("POINT(0.0054932 89.9972534)"
   , 4326, "axis-order=long-lat"), ST_GEOMFROMTEXT("POINT
   (0.0054932 90.0000000)", 4326, "axis-order=long-lat"),
   ST_GEOMFROMTEXT("POINT(0.0000000 90.0000000)", 4326, "axis-
   order=long-lat"));
```

Listing 5: Populating the table.

```
mysql> set @a = -1;
Query OK, 0 rows affected (0,00 sec)

mysql> select z,
              round(st_area(cell), 7) as area,
              round(west_edge, 7) as west_edge,
              round(south_edge, 7) as south_edge
              from cellsizes where (@a := @a + 1) % 1000 = 0;

  +------------+----------------+-------------+-------------+
  | z          | area           | west_edge   | south_edge  |
  +------------+----------------+-------------+-------------+
  | 3221225472 |  185714.645872 | 303.6998689 | 611.5002281 |
  | 3221574720 | 185503.6520455 | 303.7069376 |  610.802471 |
  | 3222622464 | 184877.2225869 | 303.7170063 | 608.7106834 |
  | 3225765184 | 183852.2558519 | 303.7630558 | 605.2294177 |
  | 3226813440 | 182393.0313169 | 303.8115899 |  600.366158 |
  | 3238346816 |  180522.059211 | 303.8621576 | 594.1314661 |
  | 3239384320 | 178285.0409984 | 303.9473834 | 586.5388147 |
  | 3242529088 | 175587.1039667 | 304.0333956 | 577.6047166 |
  | 3243577344 | 172567.2244997 | 304.1304676 | 567.3485776 |
  | 3288663104 | 169083.3457849 | 304.2377079 | 555.7927881 |
  | 3289710848 | 165227.4147767 | 304.3541315 |  542.962645 |
  | 3292812608 | 161044.4721459 | 304.4786691 | 528.8863011 |
  | 3293860864 | 156439.0364372 | 304.6101769 | 513.5946833 |
  | 3305394240 | 151456.1248959 | 304.7474469 | 497.1216694 |
  | 3306439936 | 146187.6653161 | 304.8892185 | 479.5037202 |
  | 3309584704 | 140632.0006969 | 305.0341894 | 460.7801046 |
  | 3310632960 | 134506.1681767 | 305.1810281 | 440.9926253 |
  | 3489928256 | 128277.6214269 |  305.328386 | 420.1856829 |
  | 3490976000 | 121663.6698551 | 305.4637877 | 398.4061593 |
  | 3494118720 | 114757.0770714 | 305.6192532 | 375.7032889 |
  | 3495166976 |  107650.797829 | 305.7600909 | 352.1286076 |
  | 3506700352 |  100250.324934 |  305.884992 | 327.7358515 |
  | 3507574016 |  92649.2277756 | 306.0261187 | 302.5807748 |
  | 3510718784 |  84659.5274208 | 306.1488654 | 276.7211107 |
  | 3511767040 |  76657.0051655 | 306.2520913 | 250.2164147 |
  | 3556852800 |  68454.3739259 | 306.3681977 | 223.1278374 |
  | 3557900544 |  59958.2374835 | 306.4627688 | 195.5182209 |
  | 3561035072 |  51263.4698159 | 306.5460866 | 167.4516262 |
  | 3562083328 |  42747.1190349 | 306.6173859 | 138.9933894 |
  | 3573616704 |  33846.6715987 | 306.6760117 | 110.2098169 |
  | 3574662400 |   24939.533376 | 306.7214257 |  81.1681591 |
  | 3577807168 |   15933.388059 | 306.7532106 |  51.9365164 |
  | 3578855424 |   6923.9911125 | 306.7710747 |  22.5829816 |
  +------------+----------------+-------------+-------------+

33 rows in set, 1 warning (0,65 sec)
```

Listing 6: Measurements of every 1000th row.

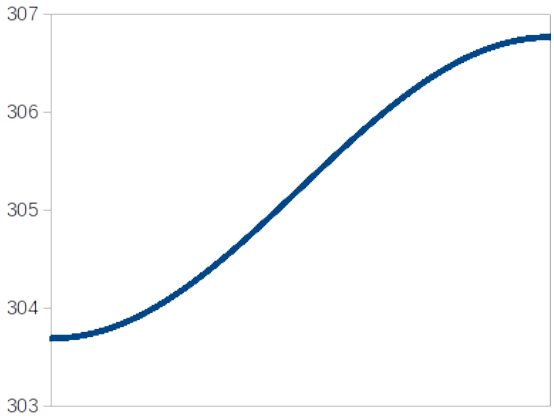


Figure 27: Length of cell west edge with adjusted Y-axis (m).

Narrowing the Y-axis range of figure 12b reveals a sigmoid curve shaped relationship between cells and their latitude, shown in figure 27. Cells are ordered along the X-axis according to their location from the Equator (left) to the North Pole (right).

Bibliography

[1] Jeroen Baert. *Libmorton: C++ Morton Encoding/Decoding Library*. https://github.com/Forceflow/libmorton. 2018.

[2] Norbert Beckmann et al. "The R*-tree: an efficient and robust access method for points and rectangles". In: *INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA*. ACM, 1990, pp. 322–331.

[3] Isaac Brodsky. *H3: Uber's Hexagonal Hierarchical Spatial Index*. URL: https://eng.uber.com/h3/ (visited on 06/28/2021).

[4] cppreference. *Shuffle*. URL: https://en.cppreference.com/w/cpp/algorithm/random_shuffle (visited on 06/26/2021).

[5] Clyde da Cruz. *OpenStreetMap WKT Playground*. URL: https://clydedacruz.github.io/openstreetmap-wkt-playground/ (visited on 04/10/2021).

[6] Yi Fang et al. "Spatial indexing in Microsoft SQL Server 2008". In: Jan. 2008, pp. 1207–1216. DOI: 10.1145/1376616.1376737.

[7] Volker Gaede and Oliver Günther. "Multidimensional Access Methods". In: *ACM Computing Surveys* (July 1999). DOI: 10.1145/280277.280279.

[8] Geofabrik. *Geofabrik download server*. URL: https://download.geofabrik.de/europe/norway.html (visited on 06/11/2021).

[9] S2 Geometry. *S2 Cells*. URL: https://s2geometry.io/devguide/s2cell_hierarchy (visited on 06/08/2021).

[10] Antonin Guttman. "R Trees: A Dynamic Index Structure for Spatial Searching". In: vol. 14. Jan. 1984, pp. 47–57. DOI: 10.1145/971697.602266.

[11] Open Geospatial Consortium Inc. *OpenGIS Implementation Standard for Geographic information - Simple feature access*. Ed. by John R. Herring. 1.2.1. 2011.

[12] Ravi Kanth V Kothuri, Siva Ravada, and Daniel Abugov. "Quadtree and R-Tree Indexes in Oracle Spatial: A Comparison Using GIS Data". In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. SIGMOD '02. Madison, Wisconsin: Association for Computing Machinery, 2002, pp. 546–557. ISBN: 1581134975. DOI: 10.1145/564691.564755. URL: https://doi.org/10.1145/564691.564755.

[13] MySQL 8.0 Reference Manual. *EXPLAIN ANALYZE*. URL: https://dev.mysql.com/doc/refman/8.0/en/explain.html (visited on 06/26/2021).

[14] MySQL 8.0 Reference Manual. *Spatial Reference System Support*. URL: https://dev.mysql.com/doc/refman/8.0/en/spatial-reference-systems.html (visited on 05/23/2021).

[15] MySQL 8.0 Reference Manual. *ST-AREA*. URL: https://dev.mysql.com/doc/refman/8.0/en/gis-polygon-property-functions.html#function_st-area (visited on 05/21/2021).

[16] MySQL 8.0 Reference Manual. *ST-DISTANCE*. URL: https://dev.mysql.com/doc/refman/8.0/en/spatial-relation-functions-object-shapes.html#function_st-distance (visited on 05/21/2021).

[17] MySQL 8.0 Reference Manual. *ST-LENGTH*. URL: https://dev.mysql.com/doc/refman/8.0/en/gis-linestring-property-functions.html#function_st-length (visited on 05/26/2021).

[18] Microsoft. *Microsoft SQL Docs - Spatial Indexes Overview*. URL: https://docs.microsoft.com/en-us/sql/relational-databases/spatial/spatial-indexes-overview?view=sql-server-ver15 (visited on 06/08/2021).

[19] Mohamed F. Mokbel and Walid G. Aref. "Space-Filling Curves". In: *Encyclopedia of GIS*. Ed. by Shashi Shekhar and Hui Xiong. Boston, MA: Springer US, 2008, pp. 1068–1072. ISBN: 978-0-387-35973-1. DOI: 10.1007/978-0-387-35973-1_1233. URL: https://doi.org/10.1007/978-0-387-35973-1_1233.

[20] OpenStreetMap. *Nodes*. URL: https://wiki.openstreetmap.org/wiki/Node (visited on 06/23/2021).

[21] OpenStreetMap. *OpenStreetMap*. URL: https://www.openstreetmap.org/about (visited on 06/11/2021).

[22] OpenStreetMap. *PBF Format*. URL: https://wiki.openstreetmap.org/wiki/PBF_Format (visited on 06/11/2021).

[23] Osmcode. *Osmium Library*. URL: https://osmcode.org/libosmium/ (visited on 06/11/2021).

[24] Frank Ramsak et al. "Integrating the UB-Tree into a Database System Kernel". In: *VLDB*. 2000.

[25] P. Rigaux et al. *Spatial Databases: With Application to GIS*. Series in Data Management Systems. Elsevier Science, 2002. ISBN: 9781558605886. URL: https://books.google.no/books?id=o8LfhpFOnPwC.

[26] Norvald Ryeng. Personal communication. June 24, 2021.

[27] H. Samet and J.G.) *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann series in computer graphics and geometric modeling. Elsevier Science, 2006. ISBN: 9780123694461. URL: https://books.google.no/books?id=KrQdmLjTSaQC.

[28] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. "The R+-Tree: A Dynamic Index For Multi-Dimensional Objects". In: 1987, pp. 507–518.

[29] Zack Slayton. *Z-order indexing for multifaceted queries in Amazon DynamoDB: Part 2*. URL: https://aws.amazon.com/blogs/database/z-order-indexing-for-multifaceted-queries-in-amazon-dynamodb-part-2/ (visited on 06/28/2021).

[30] Gaute Håhjem Solemdal. Unpublished University Project Report. Dec. 9, 2020.

[31] H. Tropf and H. Herzog. "Multimensional Range Search in Dynamically Balanced Trees." In: *Angewandte Informatik* 23 (Feb. 1981), pp. 71–77.

[32] PJM van Oosterom. "Spatial access methods". In: *Geographical information systems: principles, techniques, applications and management*. Ed. by PA Longley et al. John Wiley & Sons, 2005, pp. 385–400. ISBN: 0-471-32182-6.

[33] Wikipedia. *Dimensionally Extended 9-Intersection Model*. URL: https://en.wikipedia.org/wiki/DE-9IM (visited on 06/03/2021).

[34] Wikipedia. *Longitude*. URL: https://en.wikipedia.org/wiki/Longitude (visited on 05/21/2021).