



Norwegian University of
Science and Technology

Evaluating Algorithms for Nearest Neighbor Searches in Spatial Databases Using R-Trees

Jens Even Berg Blomsøy

Master of Science in Informatics

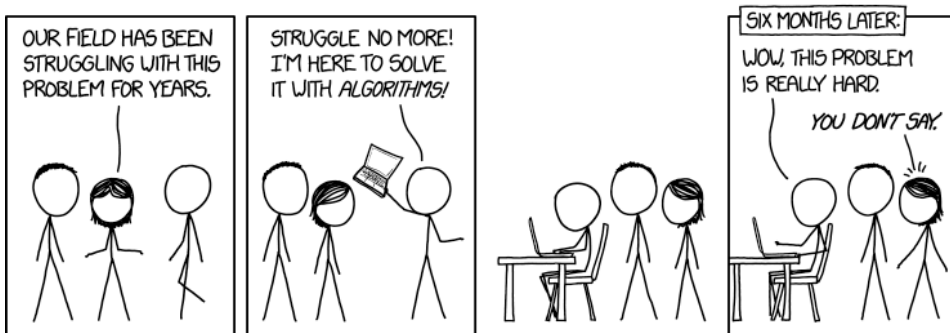
Submission date: June 2017

Supervisor: Svein Erik Bratsberg, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

This Master thesis is on the discussion and research of the topic Nearest Neighbors in Spatial Databases using R-trees. Carried out at The Norwegian University of Science and Technology as part of the study program Master of Science, Informatics: Databases and Search during the autumn 2016 and spring 2017. In cooperation with Oracle Norge, MySQL department. The feature of nearest neighbor searches has been a long awaited functionality in MySQL and is why this project was proposed. This thesis looks at several different data structures and algorithms for solving the nearest neighbor problem. The incremental nearest neighbor algorithm implemented for this thesis showed very good results in terms of speed and functionality.



[xkcd \(2017\)](#)

Trondheim, 2017-06-08

Supervisor I: Professor Svein Erik Bratsberg

Supervisor II: Norvald H. Ryeng

Sammendrag

Denne Masteroppgaven handler om evaluering og undersøkelser i forbindelse med Nærmeste Nabo Søk i Romlige Databaser ved bruk av R-Trær. Oppgaven var utført ved Norges teknisk-naturvitenskapelige universitet, institutt for datateknologi og informatikk som del i studieprogrammet Master i Informatikk: Databaser og Søk. Oppgaven ble utført høsten 2016 og våren 2017. Oppgaven har vært et samarbeid med Oracle Norge, MySQL avdeling. Funksjonaliteten for å utføre effektive nærmeste nabo søk har lenge vært en ønsket tillegg til i MySQL og er grunnen til at dette prosjektet ble foreslått. Denne oppgaven har tatt for seg flere typer datastrukturer og algoritmer for å utføre nærmeste nabo søk. Algoritmen inkrementerende nærmeste nabo som ble implementert under denne oppgaven i MySQL viste veldig gode resultater i form av hastighet og funksjonalitet.

Acknowledgment

I would like to thank Professor Svein Erik Bratsberg from the Department of Computer and Information Science and Norvald H. Ryeng from Oracle for all their help and support throughout this thesis. They have both guided the work with their expertise, which has been truly helpful. Norvald has put a lot of time in to ensure a great and efficient prototype for MySQL. I would also like to thank all the people working at Oracle Trondheim for creating a great environment for learning and enjoyment.

I would also like to thank Jimmy Yang for helping with InnoDB, and Tore Sivertsen Bergslid for helping with mathematics and everyday coding difficulties. To my motivator and light Stine Lilleberg, I am grateful.

Jens Even Berg Blomsøy

Contents

Abstract	i
Summary	ii
Acknowledgment	iii
1 Introduction	3
1.1 Motivation	3
1.2 Problem definition	4
1.3 Structure	4
2 Nearest Neighbor Problem	7
2.1 Nearest Neighbor Search	7
2.2 Where k-NN is applicable	9
2.2.1 Data mining	9
2.2.2 Clustering	10
2.2.3 Spatial data	11
2.3 Distance calculations	11
2.3.1 Hamming Distance	12
2.3.2 Manhattan Distance	13
2.3.3 Euclidean Distance	13
2.3.4 Great-circle distance	13
2.4 Dimensions	14

3	Similar works	17
3.1	Farthest Neighbor Pair	17
3.2	Closest Pair of Points Problem	17
3.3	Reverse k Nearest Neighbor	18
4	Space Partitioning data structures	21
4.1	Quadtree	21
4.2	k-d-B-Tree	23
4.3	LSD-Tree	25
4.4	B- and B ⁺ -Trees	27
4.5	R-trees	28
4.5.1	Splits	30
5	Evaluating and Classifying Algorithms	33
5.1	Linear search	33
5.2	Space Division algorithms	34
5.2.1	Incremental Nearest Neighbor search algorithm	34
5.2.2	Voronoi Diagram Algorithm in R-Tree	35
5.3	Locality Sensitive Hashing	38
5.4	Pivot Based Nearest Neighbor Algorithms	38
5.4.1	Approximating and Eliminating Search Algorithm	39
5.4.2	Linear Approximating and Eliminating Search Algorithm	41
5.4.3	Tree Linear Approximating and Eliminating Search Algorithm	41
5.5	The best fit for MySQL	43
6	Nearest Neighbor Searches in Other Database Management Systems	45
6.1	Oracle RDBMS - Oracle Spatial and Graph	45
6.1.1	SDO_NN	46
6.1.2	SDO_NN_DISTANCE	47

6.2	PostGIS extension to PostgreSQL	47
6.3	Microsoft SQL Server	49
6.4	MySQL	50
7	The most optimal algorithm found	53
7.1	The Incremental Nearest Neighbor Algorithm	53
7.1.1	Getting to The Leaf	54
7.1.2	The General Incremental Nearest Algorithm	55
7.2	Adapting to Rtrees	55
7.2.1	The Priority Queue	57
7.2.2	Extension to Furthest Neighbor	59
8	Implementation and MySQL Integration	61
8.1	InnoDB	62
8.1.1	The structure of the R-tree	62
8.1.2	Index header	62
8.1.3	Index Page	63
8.1.4	System records	64
8.2	The code	65
8.2.1	The Shortcut	65
8.2.2	The distance calculation	66
8.2.3	Limitations	68
9	Evaluating the Incremental Nearest Neighbor Algorithm in MySQL	69
9.1	Test plan and setup	70
9.1.1	Incremental Nearest Neighbor Algorithm	70
9.1.2	Table Scan	70
9.1.3	Hardware and Software	71
9.1.4	Load Emulation Client - mysqlslap	71

9.2	Querying	72
9.2.1	Random datasets	73
9.2.2	Optimal datasets	73
9.2.3	Worst case dataset	74
9.2.4	Best vs. Worst	74
9.2.5	Open Street Map	75
9.3	Summary	85
10	Conclusion and Future Work	87
10.1	Conclusions	87
10.2	Future work	89
10.2.1	MySQL Worklog	89
10.2.2	Other Spatial Data Types	89
10.2.3	Furthest Neighbor	90
A	Source Code	91
B	Python point generators for MySQL	93
B.1	Large random dataset	93
B.2	Best case dataset	94
B.3	Worst case dataset	95
C	Open Street Map	97
C.1	Introduction	97
C.1.1	Importing Open Street Map Data	97
C.1.2	Altering tables	100
	Bibliography	102

List of Figures

2.1	Distance query	8
2.2	Example of clustering kNN encoding. Notice the individual colored points, these have unique nearest neighbors Jacob Steeves (2015).	10
2.3	The vertices in of F_2^3 in a tree dimensional cube	12
2.4	The fastest route on Manhattan, is the Manhattan Distance	13
3.1	The closest pair Qef (2016)	18
3.2	R2NN and 2NN Tao et al. (2004)	19
4.1	A visual representation of a quadtrees internal structuring Johnson, Nick (2009)	22
4.2	A visual representation of the division of an indexed area of the quadtree Johnson, Nick (2009)	22
4.3	Structure of the k-d-B-tree Troyhildebr (2014).	24
4.4	Structure of the LSD-tree Gaede and Günther (1998).	26
4.5	A binary tree where you can see that the leaf nodes are at different levels Tmigler (2016)	27
4.6	A B-tree internal nodes can have more than two children Press (2008)	27
4.7	Structure of the R-tree Mendes et al. (2014).	29
5.1	Spatial Division and several NN Sourceforge (2016)	34

- 5.2 Voronoi Diagram Sharifzadeh and Shahabi (2010) 36
- 5.3 The Voronoi diagram within an R-Tree - VoR-Tree Sharifzadeh and Shahabi (2010) 37
- 5.4 Visualizing Pivots Jin Liang (2004). 39
- 5.5 Lower Bounds Tavian (2016). 40

- 6.1 The structure of the Nearest Neighbor Search in Microsoft SQL Server as show in their documentation Documentation (2017b). . . 49

- 8.1 Index header structure in InnoDB Jeremy (2013). 63
- 8.2 Index page structure in InnoDB Jeremy (2013). 64
- 8.3 System records in InnoDB Jeremy (2013). 65
- 8.4 Nearest neighbor hook 66
- 8.5 Point and minimum bounding rectangles. 67
- 8.6 The point to line segment calculation. 67
- 8.7 Push of matches 68

- 9.1 The mysqlslap function and query. 72
- 9.2 The New York area dataset visualization Mapzen (2017). 76
- 9.3 The 5 closest restaurants to Times Square. 78

List of Tables

9.1 Worst to Best Query Results	74
9.2 Restaurant Query Results	78
9.3 Places to get food Query Results	80
9.4 All amenities Query Results	81
9.5 All points within New York dataset Query Results	83

Chapter 1

Introduction

From a user point of view the nearest neighbor search in an app can be to find the closest restaurants, hotels or other amenities within an area. In spatial operations being able to do a nearest NNS simple and efficiently is important for many users. The applications for this search is vast and is the reason many of today's most used Database Management Systems (DBMS) has implemented this search specifically. MySQL is the last of the big to do so with the research from this master thesis.

In order to do an efficient NNS the DBMS need a spatial structure such as the B-Tree, K-D-B or R-Tree to search in. The next thing needed to perform a fast search is a dedicated algorithm, even though it is possible to find the nearest neighbor using some sort of distance scan and a later sorting. With this last approach the time complexity for most datasets becomes unbearable.

1.1 Motivation

MySQL as of today does not have a dedicated NNS algorithm, instead through full table scans, distance calculations and sorting, NNS is possible. MySQL wishes to extends its geographical information system features with an efficient

NNS to be on par with the other major DBMS of today. The NNS algorithm should fit into existing structures such as their implemented R-Tree and the way MySQL accesses indexes which are incremental. On several occasions the data is unknown, and finding the k-nearest neighbors may not be feasible. An example is if a user asked for the 10 nearest restaurants: most algorithms would require a bounding box to search within, now if this scope did not have 10 restaurants, the search would come up empty or the algorithm would need to increase the bounding box or reduce the number of wanted restaurant. A desirable feature would be to not require a bounding box scope for the search nor require knowing k in advance.

1.2 Problem definition

Research the existing NNS algorithms within different data structures to find a suitable version to implement as a prototype in MySQL. Use this implementation to test against existing solutions within MySQL. Research the performance of real world problems relating to NNS using data from real maps. While implementing the prototype it is important to leave options available for extensions and future work.

RQ I - What algorithms exists for performing NSS?

RQ II - Which algorithms is suitable to execute an efficient NNS within the existing R-tree of MySQL?

RQ III - Can an implemented NNS algorithm outperform existing solutions in MySQL?

1.3 Structure

The remaining parts of the thesis is structured as follows: Chapter 2 will define the nearest neighbor problem, where it is applicable and some of the different distance calculations.

Chapter 3 discusses similar works done within this field of algorithms.

Chapter 4 describes different types of Space partitioning data structures, such as the Quad-Tree, k-d-B-Tree, LSD-Tree and the closely related B and R trees.

Chapter 5 evaluates different types of NNS algorithms and goes into depth about the two most promising ones Tree Linear Approximation and Eliminating Search Algorithm and the Incremental Nearest Neighbor Search Algorithm.

Chapter 6 describes existing solutions within other DBMS. The chapter outlines the NNS in Oracle RDBMS, PostGIS, Microsoft SQL Server and the status of current solutions within MySQL.

Chapter 7 outlines in detail the most optimal algorithm found for MySQL. The Incremental Nearest Neighbor Algorithm (INNA) from standard top-down traversal of a data structure to the complete INNA with its belonging priority queue.

Chapter 8 outlines the inner workings of MySQL's storage engine InnoDB. Starting with the structure of the R-tree, Index header, Index pages and System records. The chapter discusses also the implementation of the prototype for this thesis.

Chapter 9 evaluates the INNA in MySQL. The chapter outlines the test plan, the querying process and different types of datasets used to see a range of results.

Chapter 10 gives the conclusion and future work for the INNA in MySQL.

Chapter 2

Nearest Neighbor Problem

The Nearest Neighbor (NN) rule is one of the classic pattern recognition problems. The first mentions of this problem is by Alhazen (Ibn al-Haytham, 965-1040 C.E.) one of the scientist of the Islamic golden age. The rule for NN classification can be seen in the not so well known parts of Alhazen's defining treatise on optics of the eye. He noted that the sight perceives an object, such as a man or horse and immediately seeks its counterpart in the imagination, thus connecting it to its nearest neighbor. He also noted that if there exists no similar form of the visible object in the imagination, the object will not be recognized [Pelillo \(2014\)](#).

The more formal version in which we think of the NN problem today can be seen in Nearest Neighbor Pattern Classification of 1967 [Cover and Hart \(2006\)](#). They state that the NN decision rule assigns to a point that is an unclassified sample which the classification of the nearest in a set of points previously classified.

2.1 Nearest Neighbor Search

The NN search in proximity searches is an optimization problem of finding the closest point to another point in a set. This can also be to find the most similar

object to another object as stated in Alhazen's interpretation of the functionality of our vision. As the points or objects gets further apart or dissimilar the larger the function values. In a simple real world problem, Donald Knuth named it the post-office problem. The problem was to assign each resident of a given area to their closest post office. A well known format of this problem is the k -NN search, where one struggles to find the k nearest points to another given point.

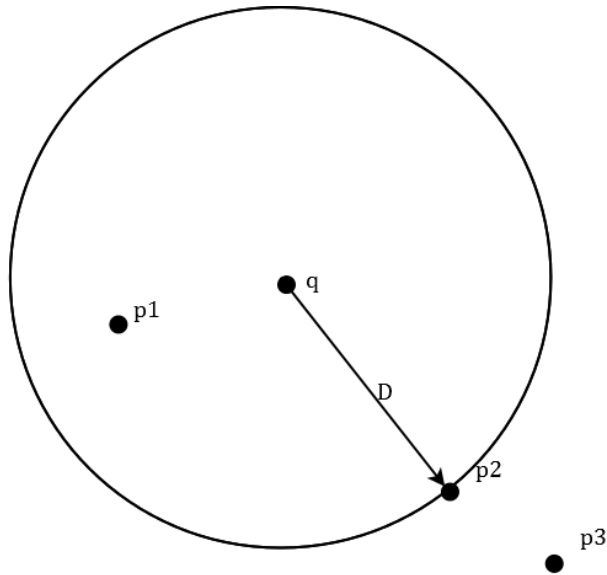


Figure 2.1: Distance query

Figure 2.1 illustrates the k -nearest neighbor (k -NN) query where k is set to 2. In this data set $p1$ is the nearest neighbor and $p2$ is the k -th NN, while $p3$ is not part of the k -NN. The distance from the query point q to $p2$ is marked by D , depending on the algorithm the distances could be returned with the result of the query. During the query processing the data structures are searched with the value of D in increasing order. In other queries one could only be interested in the closest objects regardless of the distance values, this would then be non-sorted.

2.2 Where k-NN is applicable

When you think of the NN problem in the terms of a set of points in n-dimensions. Now given a new point usually called query, the goal is to find the k-closest points to this query point. One can see that this applies to a lot of different real world problems.

2.2.1 Data mining

The NN algorithm is a useful technique within data mining that lets you use the data you already know the output value of to create new and unknown output values. This is both similar to regression and classification, but regression can only be used for numerical outputs and that differentiates it from the nearest neighbor immediately. With regards to classification it uses every instance to create a tree, and then traverses that to find the answer, this will become very time consuming for large data sets. Take eBay as an example, to make similar product information would require a very large tree structure; this would be an unreasonable model for this type of data. The nearest neighbor algorithm would solve all these problems and is a very good fit for problems of a similar sort [Abernethy \(2010\)](#).

Data mining has been in a large extent used to mine knowledge from medical databases. Such data has often several classifications and sometimes redundant and irrelevant attributes, this causes the classification to produce less accurate results. By combining so called genetic algorithms and the k-nearest neighbor algorithm Jabbar Akhil and his colleagues was able to produce better classifying results for heart disease [Jabbar et al. \(2015\)](#).

2.2.2 Clustering

Clustering is the process of grouping data or objects together such that everything contained in one group shares more similarity with each other than any other group. It is used in many fields such as pattern recognition, machine learning, information retrieval and data compression. To achieve clustering of data or objects one has many different algorithms for completing the task, the k-NN being one of them.

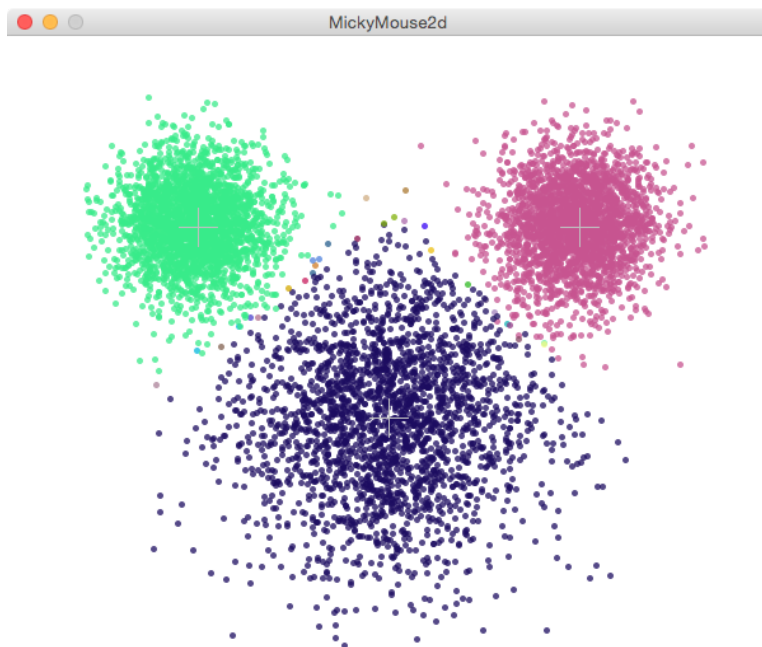


Figure 2.2: Example of clustering kNN encoding. Notice the individual colored points, these have unique nearest neighbors [Jacob Steeves \(2015\)](#).

The k-NN algorithm is among the most important non-parameter algorithms in the field of pattern recognition, it is also a supervised learning predictable classification algorithms. On the topic of clustering the k-NN algorithm has gained

much popularity and extensive development has been made. This in spite of its many faults, such as great calculation complexity and association functionality when we talk about clustering of words [Jiang et al. \(2012\)](#). Earlier mentioned rules, which is how the algorithm classifies the data, is derived from the training samples without any extra data. This in turn makes the algorithm depend only on the training sets and if the data changes it needs to be re-calculated, adding to the faults of the kNN algorithm. Clustering is a process of finding the NN for all objects so called all too all searches in addition to the grouping of the objects.

2.2.3 Spatial data

One of the most direct uses of the NN search algorithm is in spatial data. Searching in maps for the nearest restaurants, restrooms, hotels and so forth. The possibilities and use cases for NN searches in spatial data is of vast proportions. Furthermore the use of NN searches can be of great use in mobile social networks where users are in a particular distance from one another [Choi and Chung \(2015\)](#). Such searches are becoming very useful in different applications on the users phones, and to provide the user with a fastest possible search would be of importance.

2.3 Distance calculations

Which distance calculation one uses to do the NN query is based on the properties of the data. As there are several different distance calculations, to find the best fit for a dataset if of importance. If one does not know the dataset very well, trying out several distance calculations would be preferable. This is mostly when dealing in higher dimensional data, for spatial data it is easy to see the obvious answers [Brownlee \(2016\)](#).

2.3.1 Hamming Distance

The first distance mentioned here is the Hamming Distance, due to its prevalent nature in nearest neighbor topics. The Hamming distance $d(\underline{x}, \underline{y})$ of two vectors $\underline{x}, \underline{y} \in F^n$, where F is a field of finite proportions, is the number of coefficients differentiating them. Example:

$$F_2^4 d(xxyy, yyxy) = 3$$

$$F_3^5 d(xyzzz, yzzxz) = 3$$

The hamming distance between two of the vertices in this cube is the shortest path connecting these two vertices in any direction. The shortest of path of $d(xxx, xyy)$ is 2, through xyx . The Hamming distance in F_2^n can be understood as the least number of edges in a path connecting two vertices in this n-dimensional cube. For a NN this means that given a set $S \subset F^n$ and the vector $\underline{y} \in F^n$ then $\underline{x} \in S$ is a NN to \underline{y} if

$$d(\underline{x}, \underline{y}) = \min(d(\underline{z}, \underline{y}) | \underline{z} \in S) \text{ School of Mathematics (2007).}$$

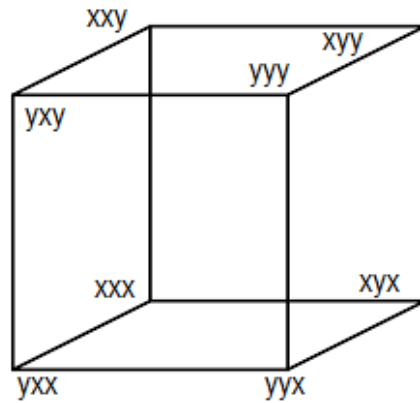


Figure 2.3: The vertices in of F_2^3 in a tree dimensional cube

2.3.2 Manhattan Distance

When the data is not of a uniform type such as age, gender, height and so forth, it is a good choice to use the Manhattan distance. The Manhattan distance between two points P M is defined as: $d(M, P) \equiv |M_x - P_x| + |M_y - P_y|$ [Wolfram \(2017b\)](#)

One can think of the Manhattan Distance as the route a taxi cab needs to drive to get to one point on Manhattan to another, navigating the grid of blocks in straight lines as shown in picture 2.4.

2.3.3 Euclidean Distance

The Euclidean distance is a good measure if the data is of a similar type, such as geometries [Brownlee \(2016\)](#).

If $u = (x_1, y_1)$ and $v = (x_2, y_2)$ are two points on the plane, their *Euclidean distance* is given by

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \quad (2.1)$$

2.3.4 Great-circle distance

A great circle is part of a sphere that has a diameter of the sphere. The sections that do not contain a diameter are referred to as small circles. The great circle transfers to a straight line in a gnomonic projection. The shortest distance between two points on a sphere are known as an orthodrome and is part of a

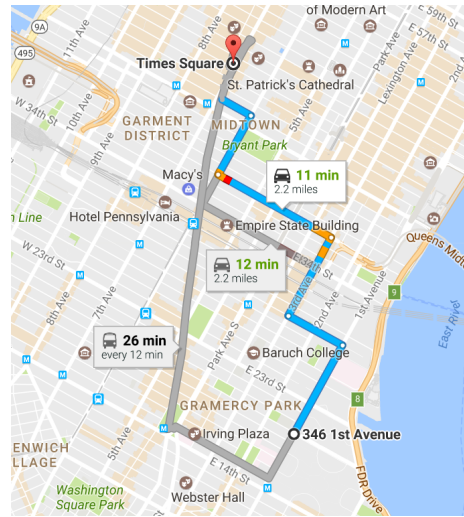


Figure 2.4: The fastest route on Manhattan, is the Manhattan Distance

great circle [Wolfram \(2017a\)](#).

When measuring data on the globe, such as the distance between two cities or the shortest flight path, it is a good choice to use the Great-circle Distance. When using this one would get the most precise measurement, and the dissimilarity between reality and distances measured in straight lines, increases with the distance measured. Using the great-circle distance to measure points within the same street is of a negligence difference from reality opposed to doing the distance calculation in the Euclidean distance. The great-circle distance calculation is a more costly distance measure than Manhattan and Euclidean distance as we will see.

$$Distance = 6372.8(2 \arcsin(\sqrt{\sin^2(\frac{rad(x_2 - x_1)}{2}) + \cos(rad(x_1)) * \cos(rad(x_2)) * \sin^2(\frac{rad(y_2 - y_1)}{2})})) \quad (2.2)$$

The distance between two points given the latitude as x and longitude as y.

2.4 Dimensions

The most naive and time consuming is to calculate the distance to all the other data points and sort them according to the lowest distance.

In the lower levels of dimensions one can use different types of spatial indexing.

However when the number of dimensions gets higher these spatial indexes performs even worse than the most naive of approaches [Arya et al. \(1998\)](#). Solving for the nearest neighbor problem in higher dimensions becomes increasingly difficult, this is in some degree due to the distance functions used can give vastly

different behaviors in low to higher dimension spaces. The term is "The Curse of Dimensionality" [Giannella \(2009\)](#). Solutions to the Curse of dimensionality is to use some sort of approximation techniques defined as Approximate Near Neighbors which will be further explained in [chapter 5](#).

Chapter 3

Similar works

This chapter looks into work similar to finding the nearest neighbor (NN) in metric space, touching on some lesser known and complex topics. The need to find something near another thing extends to so many applications and problems, and the field is vast.

3.1 Farthest Neighbor Pair

The farthest neighbor problem is as it sounds, the goal is to find a pair furthest away from the query point. Let V be a set of points n , find $\tilde{x}, \tilde{y} \in V$ that holds for:

$$d_p(\tilde{x}, \tilde{y}) = \max\{d_p(\tilde{u}, \tilde{v}) \mid \tilde{u}, \tilde{v} \in V\} \text{ Yao (1982)}$$

3.2 Closest Pair of Points Problem

The closest pair of points problem is given is a problem related to finding the 1 NN for all points in a data set, in figure 3.1 one can see 1 NN pair. Given n points in metric space, one needs to find the pair of points with the smallest distance from each other. The problem can be solved in $O(n \log n)$ time with the help of the recursive divide and conquer approach [Shamos and Hoey \(1975\)](#).

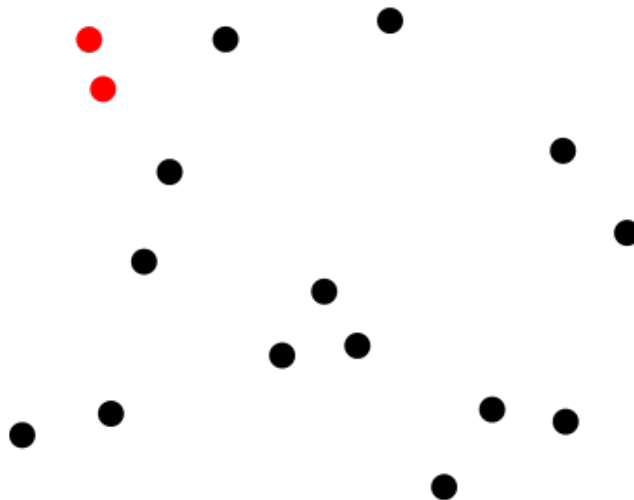


Figure 3.1: The closest pair [Qef \(2016\)](#)

3.3 Reverse k Nearest Neighbor

Given a point q in the multi-dimensional dataset P the reverse k nearest neighbor search (RkNNs) gives all data points $p \in P$ that have q as one of their k nearest neighbors (kNN). This is a fairly new problem but it has already gained much attention due to its importance in many applications, involving decision support, resource allocation, profile-based marketing and so on. In figure [3.2](#) one can see four points where all of the p points have a circle covering its two NN (2NN). The 2NN of $p_4(p_2, p_3)$ is in the circle with center at p_4 . This is denoted $p_4 \in R2NN(p_2)$ and $p_4 \in R2NN(p_3)$ [Tao et al. \(2004\)](#).

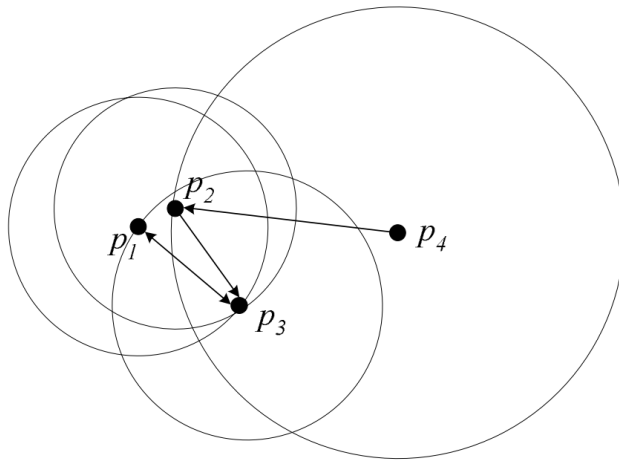


Figure 3.2: R2NN and 2NN [Tao et al. \(2004\)](#)

Chapter 4

Space Partitioning data structures

In this chapter we look at the different data structures that can be used to improve the nearest neighbor search. While there are even more data structures possible to optimize the nearest neighbor problem, we take a look at the most popular ones related to the nearest neighbor search. The main focus will lie on the different data structures of the R-tree variant.

4.1 Quadtree

One of the most widely known block decomposition data structures is the quadtree, which is usually used for two and three dimensional data. Although there are many different uses for quadtrees we focus here on the region quadtrees. The region quadtree is a specific example of interior-based representation for two- and three-dimensional regional data. The quadtree is a type of tree defined by a restricted-decomposition rule, as the environment containing the object is decomposed recursively into four or eight blocks of a rectangular fashion, blocks are within blocks until each block is occupied only by the object or is empty

Samet (2005a).

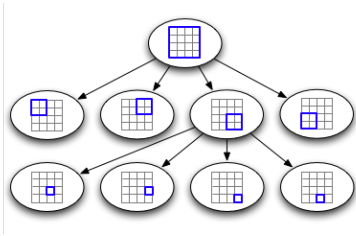


Figure 4.1: A visual representation of a quadtree's internal structuring Johnson, Nick (2009)

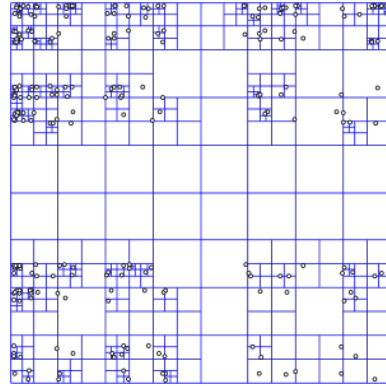


Figure 4.2: A visual representation of the division of an indexed area of the quadtree Johnson, Nick (2009)

In figure 4.1 one can see the node structure. The root node (top one) encapsulates the entire tree. The next level is the internal nodes they point to exactly four leaf nodes, in this case (there can be several internal node levels). The four leaf nodes is each a quadrant of the parent node. The leaf nodes contain an indexed object and no child nodes.

In figure 4.2 one can see the indexed area of a quadtree. Notice the space and time saving empty internal nodes in the middle.

Inserting into the quadtree is done by starting at the root and determining which quadrant the object belongs to, recurse that node and do the same again until you reach the leaf level. At this point add the object to this node's record of objects. If this exceeds the maximum number of objects allowed at the leaf level, split the node and move those objects respectfully.

The query process of the quadtree is simple as well, starting at the root one looks at each child node to see if it intersects the area being queried for. If this is the case it goes into that child. When it enters a leaf node the process is to examine

all entries to see if it intersect the query area and return it if so [Johnson, Nick \(2009\)](#).

4.2 k-d-B-Tree

The basic bucket methods within tree access structures uses a directory that is different from quadtrees and kd-trees access structures. The k-d-B tree is the simplest tree directory methods from a conceptual view but it has its drawbacks, the possible need to split several nodes when there is overflow within a node. The k-d-B-tree is a form of bucket variant of the k-d-tree as the partition lines does not need to intersect the data points, one need not to cycle through the keys and all the data is stored in the leaf nodes [Samet \(2005b\)](#).

The leaf nodes of the k-d-B-tree has a capacity of b (point pages) and the leaf node also called bucket for this special tree is split if the size exceeds b points. The non-leaf nodes are grouped into buckets of capacity c (region pages) and c corresponds to the number of pointers in each sub tree to the point pages or other internal nodes, region pages. C is the maximum number of regions the can be composed into a region page. The size of c does not need to correspond to the size of b and is usually larger because they represent the boundaries of regions [Samet \(2005b\)](#).

In figure 4.3 one can see the split of the root page into smaller region pages, these again contain the point pages also known as the leaf pages. The leaf pages contain the actual data points.

The resulting k-d-B-tree has several of the same features as the B-tree, but updates cannot always be achieved in a comparable way. This is especially true when the values of the new data are inserted or deleted and overflow or underflow happens, the bucket is either split or merged. If insertion of a point in a k-d-B-tree causes overflow in a point page, then a partition line and axis is

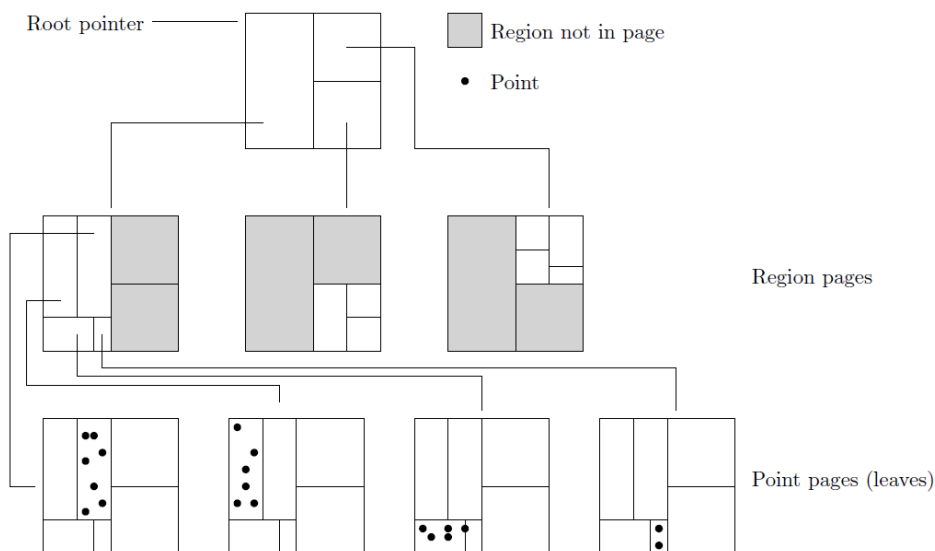


Figure 4.3: Structure of the k-d-B-tree [Troyhildebr \(2014\)](#).

decided and the bucket is split. This creates two additional point pages p_1 and p_2 and the data d is inserted into the correct new point page. After the insertion the data d is deleted from the region page r that used to hold it, now p_1 and p_2 is added to r . This adding may again cause the region page to overflow, needing another split. When this is the case one chooses a partition line l and axis in the region in consideration to r and then split r into two region pages, r_1 and r_2 . If the region page r has overflowed and r is not the root page, then r can be deleted from the parent page g . The region pages that one just created is inserted into g and the process of checking for overflow starts again. If r was the root page of the k-d-B-tree the process is a bit different, now a new region page is h is created to contain the region pages r_1 and r_2 as its objects and l as the partition line. One sees now that the k-d-B-tree grows at the root, this is similar to how the B^+ -tree grows [Samet \(2005b\)](#).

4.3 LSD-Tree

The local split decision tree (LSD) is dissimilar from the k-d-B-tree by choosing partition line that does not intersect any of the regions that makes up the overflowing region page, in doing so it avoids the need to do a recursive invocation of the splitting process. This is possible by partitioning each region page of the k-d-B-tree at the root and the space partition created by the overflowing region page [Samet \(2005c\)](#).

In similarity with the k-d-B-tree the LSD tree is best visualized by starting with a variant of a bucket generalized pseudo LSD tree. The main idea behind the LSD tree is the differentiating between the treatment of the non-leaf nodes and the leaf nodes that hold the data so called data buckets. The non-leaf nodes are contained in a directory page that has a finite capacity, this can be thought of as the region pages that were in the k-d-B-tree. This compiles into a variant of a B^+ -tree where the root, called the internal directory page, is contained in the main memory at all times. The internal directory page points to the other levels called external directory pages, they remain in external storage [Samet \(2005c\)](#).

The external directory page is a k^+ -d-tree in itself, it is implemented as an extended binary tree with a fixed height of h . The external directory page is made up of leaf nodes that point to either data buckets or other external directory pages, this would be the region pages in the k-dB-tree. By the definition of the LSD tree the paths from a leaf node in the internal directory to any data bucket is similar, while the path from different leaf nodes in the internal directory to any data bucket will at most be 1 step further or closer in the pathway [Samet \(2005c\)](#).

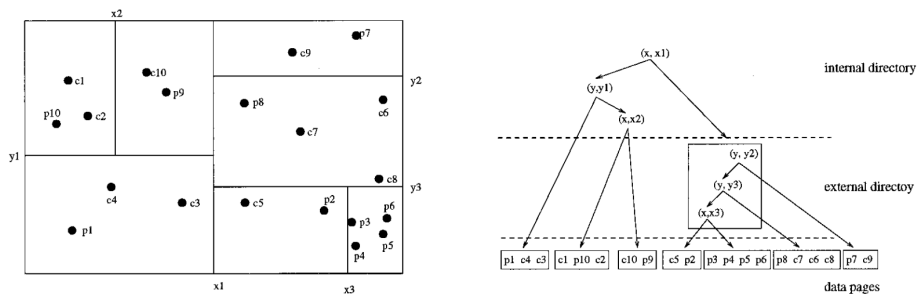


Figure 4.4: Structure of the LSD-tree [Gaede and Günther \(1998\)](#).

Pointers that connect the elements of external directory pages and data buckets form a collection of trees, the internal nodes of these are also external directory pages as shown in figure 4.4. The maximum fan-out of the internal nodes within these trees is 2^h . Moreover the path from the root to the leafs has the same length, meaning the number of external directory pages, this again means that the tree is balanced, although the individual internal nodes may be of a different degree. Last, the height of any two of these trees differs only by one [Samet \(2005c\)](#).

Insertion into the LSD-tree that causes the buckets to overflow creates a split of the node, the split is according to several special rules of the LSD-tree. One of these rules enforces that the number of data elements of the resulting data buckets is of the same size, further a rules ensures that space size of the two buckets is equal. The rules which is called data-dependent and distribution-dependent strategies, and they are based on a local view of the dataset. Splitting the buckets will create an additional bucket, and a corresponding additional internal node is inserted into the external directory page. The external directory pages are split if the adding of a new bucket causes the height of the tree to exceed maximum value h . When a split is made the directory page is split at the root of the internal tree of the node. If the split is propagated upwards in the tree it will cause a node to be added to the internal directory page.

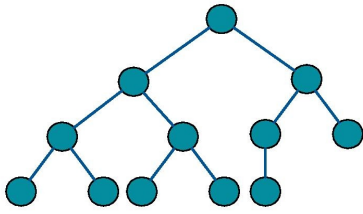


Figure 4.5: A binary tree where you can see that the leaf nodes are at different levels [Tmigler \(2016\)](#)

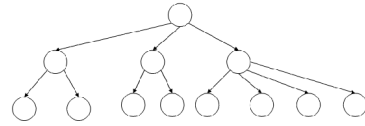


Figure 4.6: A B-tree internal nodes can have more than two children [Press \(2008\)](#)

4.4 B- and B⁺-Trees

The B-tree and the variants are the most widely used external searching techniques. When thinking of a binary tree page faults can be a problem, if one has a tree of 2 million records the amount of comparison to reach the leaf node is $\log_2 N$. At this number of records one would make 21 comparisons and have a possibility to cause 21 page faults, which makes the binary tree unsuitable for external searching. One tries to minimize these faults, by reducing the depth of the tree, if our 21 depth tree was reduced to 7, one would have 14 less possible page faults. By using a multiway tree instead of a binary one can also reduce the page fault considerably.

A multiway tree is based on the binary tree structure but at the leaf level it tests m ($m \geq 1$) key values instead of just one. From there it makes an $(m+1)$ -way branch but even this is flexible, the tree is guaranteed balanced in the way that all leaf nodes are at the same level. This known as the B-tree [Samet \(2005d\)](#).

Searching in B-trees is a very simple task. The process is similar to searching in a binary tree, but in addition one must perform a search within each non-leaf node with a key value k in a B-tree one searches for k among the k_i stored in the node. One will find k and exit or one will find the highest valued i such that $k_i < k$ and find the node pointed to by p_i , if one is at a leaf node the search was

unsuccessful. While searching in B-trees is simple, insertion is more complex. Searching for the record with the key value k , if the search is unsuccessful, this is the position to insert k . One tries to insert the record in the non-leaf node p with the pointer to t . If the node p has room then this is where one inserts the record. If node p is full, there is overflow, then one needs to split node p into two, the contents of node p is divided as equally as possible [Samet \(2005d\)](#).

Another version of the B-tree is the B^+ -Tree, it is very similar to the B-Tree but differs most the sense that all the keys reside in the leaves. The higher levels of the tree are organized as a regular B-tree and consists only of an index as a road-map to enable fast location of the index and key parts. The index nodes and the leaf nodes may have different formats and be of a different size. Usually leaf nodes are linked together left-to-right. Insertion and search operations are nearly identical to the B-tree. Improvement over the B-tree is that the B^+ -trees ability to leave non-key values in the index during deletion, and in the leaf is at least half full the tree does not need to be changed [Comer \(1979\)](#).

4.5 R-trees

The R-tree was a proposal by Antonin Guttman in 1984 focused on handling geometrical data, this including points, lines, areas, volumes and other higher dimensional spaces. There are dozens of different implementations of the R-tree, all aimed at solving different problems more efficiently. The R-tree is used within a wide variety of fields such as spatial, temporal, images and multimedia databases. In simple terms the R-tree is a hierarchical data structure based on the B^+ -tree, and is used for the dynamic organization of sets of d -dimensional geometries, by viewing them in their minimum bounding d -dimensional rectangles (MBR).

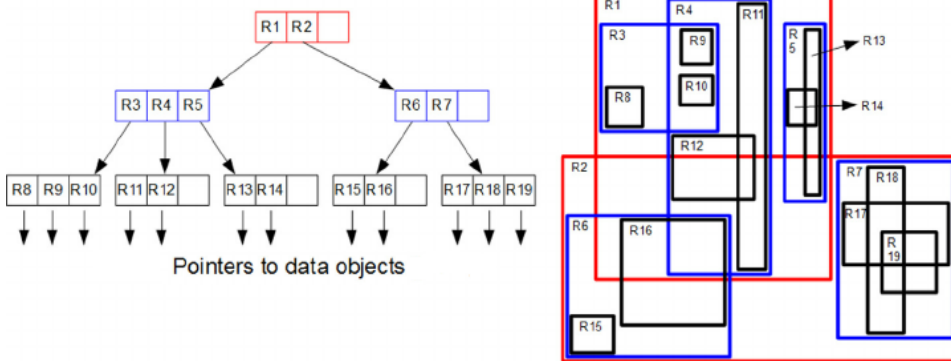


Figure 4.7: Structure of the R-tree [Mendes et al. \(2014\)](#).

The nodes within the R-trees is a MBR of the children of that node, the nodes are disk pages. At the leaf level there are pointers to the database object instead of the regular pointers to the child nodes. The MBR's that encapsulates the nodes may overlap with other nodes, how much and how often greatly depends on the type of split that is used on the tree as we will discuss in greater detail below [Manolopoulos et al. \(2005\)](#).

- M is the maximum number of entries in an R-tree node
- m is the minimum number of entries in an R-tree node
- oid signifies the object identifier
- mbr the MBR that spatially encapsulates the object
- p is a pointer to a child node

Given a R-tree of order (m, M) each leaf node (not the root node) can contain up to M entries, and the minimum allowed number of entries is $m \leq M/2$. The entries are on the form of (mbr, oid) . In regards to the internal nodes the maximum capacity is between $m \leq M/2$ and M . The internal nodes entries are on the form of (mbr, p) . The root node has a minimal requirement of 2 entries,

unless this is also a leaf node, in that case it may contain zero entries or one entry. At last all the leaf nodes are at the same level in the R-tree [Manolopoulos et al. \(2005\)](#).

By the definition of the R-tree it is a height-balanced tree and a generalization of the B⁺-tree structure for several dimensions. They are dynamic data structures, as the R-tree do not require reorganization to tackle insertions and deletions [Manolopoulos et al. \(2005\)](#).

4.5.1 Splits

The original R-tree had the goal of minimizing coverage, as an object t is inserted by starting at the root of the tree, it chooses the child whose bounding box needs to be expanded the least to encapsulate the new object. As the tree nodes needs to be as full as possible, it is unnecessary to check all possibilities of the split positions and grouping of objects, as this takes exponential time, $O(2^N)$ [Samet \(2005e\)](#).

One of the goals of the node splitting process is to distribute the objects to the nodes such that the possibility of the two nodes will be visited in subsequent searches is reduced. To achieve this the total area covered by bounding boxes of the resulting nodes is minimized, this is referred to as coverage. Another goal is to reduce the chance that both nodes will be examined in subsequent searches. To achieve this; the common area of both nodes are minimized, this is what one call overlap [Samet \(2005e\)](#).

There are several node splitting solutions and each try to take these goals into account. They differ in their execution-time complexity based on which one of these goals they try to achieve.

Optimal Split

A way to achieve the optimal split is to just try every possibility. The number of possible partitions is $2^M - 1$ as this becomes unproductive and incredibly time consuming for almost all values of M . In example for a page size of 1024 bytes M is 50. This exhaustive approach obtains an optimal node split in regards to coverage overlap and other functions [Guttman \(1984\)](#).

Even though the exhaustive algorithms takes $O(2^M)$ time, it is possible to construct it in such a way that the cost of time will be reduced drastically to $O(M^3)$ time for two dimensional data and $O(dM \log M + d^{2d-1})$ time for data of a higher dimension where d is the number of dimensions. This implementation proposed by Becker, Franciosa, Gschwind, Ohler, Thiemt and Widmayer has the benefit of ensuring that the splitting of the nodes satisfies a balancing criteria, this is also a requirement in most R-tree implementations [Becker et al. \(1992\)](#). By doing this the nodes are not overly fill, which would cause them to overflow once more.

Constraints such as comfortable spaced nodes are used in many R-tree implementations. The common thing is to have each node in the split contain half of the rectangles or at least 40% of the rectangles. The balancing criteria do of course add cost to the execution. By comparison an exhaustive algorithm without this balancing criteria runs on $O(M^2)$ time, and the one proposed by Becker and his colleagues runs on $O(M^3)$ time [Samet \(2005e\)](#).

Forced reinsertion

Even though throwing a lot of computational time at the problem and obtaining optimal node splits, the query performance is only raised by a small amount. This lead García, Lopez and Leutenegger to introduce another form of node splitting similar to what one calls *forced reinsertion*. This split is based on fitting

one of the two groups from a split node a into only one of a 's siblings, and not creating a new node for each split. One group is inserted into a 's siblings b , when there is overflow, one continues to insert into other unvisited siblings. Now, if one has visited all the siblings of a , the process creates a new node, this node contains the last overflown group of the siblings. This process will eventually terminate because there is one fewer sibling candidate at each step [García et al. \(1998\)](#). There is some difference between this approach and a regular forced reinsertion technique, this is reinsertion on individual entries at the root instead at the siblings. This strategy increased performance in querying by up to 120% compared with the Hilbert R-tree [García et al. \(1998\)](#).

Quadratic Split

The next type of split is what one calls seed-picking. The control structure consists of two stages, the *seed picking stage* and the *seed growing stage*. In the pickings stage, a pair of bounding boxes j and k is chosen as seeds for the two resulting nodes. The second stage is the redistribution of the remaining bounding boxes into the nodes that is j and k . While doing the redistributing the process tries to minimize the spatial growth covered by nodes j and k .

The quadratic split algorithm starts with finding the two bounding boxes that wastes the most space if they was in the same node. It does this by subtracting all of the areas of the two bounding boxes from the area of the covering bounding box. These two bounding boxes are now placed in different nodes, j and k . For every other bounding box i , d_{i_j} and d_{i_k} is calculated, this is the increase in the area of the covering bounding box of the nodes j and k when i is added. Given the bounding box r $|d_{r_j} - d_{r_k}|$ as a maximum is found and r is added to the node with the smallest increased area. This process is executed for the remaining bounding boxes [Samet \(2005e\)](#).

Chapter 5

Evaluating and Classifying Algorithms

This chapter looks at the nearest neighbor search (NNS) algorithm landscape that exists. While there are too many different algorithms to mention for NNS here, we take a look at some of the more interesting and promising algorithms, starting with the basic linear search to establish a base line.

5.1 Linear search

Linear search is the simplest and most naive approach to the nearest neighbor (NN) problem. It finds the NN by computing the distance from the query point to every other point in the database, keeping track of the best candidate. This has a running time of $O(dN)$ where N is the number of points and d is the number of dimensions. While the linear search is a bad choice for most applications, using it for searches in high-dimensional spaces can on average be faster than other space partitioning algorithms [Weber et al. \(1998\)](#).

5.2 Space Division algorithms

Within the space partitioning data structures such as the k-d-b-Tree here in figure 5.1 and the others discussed in chapter 4 there are several highly optimized algorithms for doing the NNS. This section touches on several algorithms researched for their functionality and time complexity.

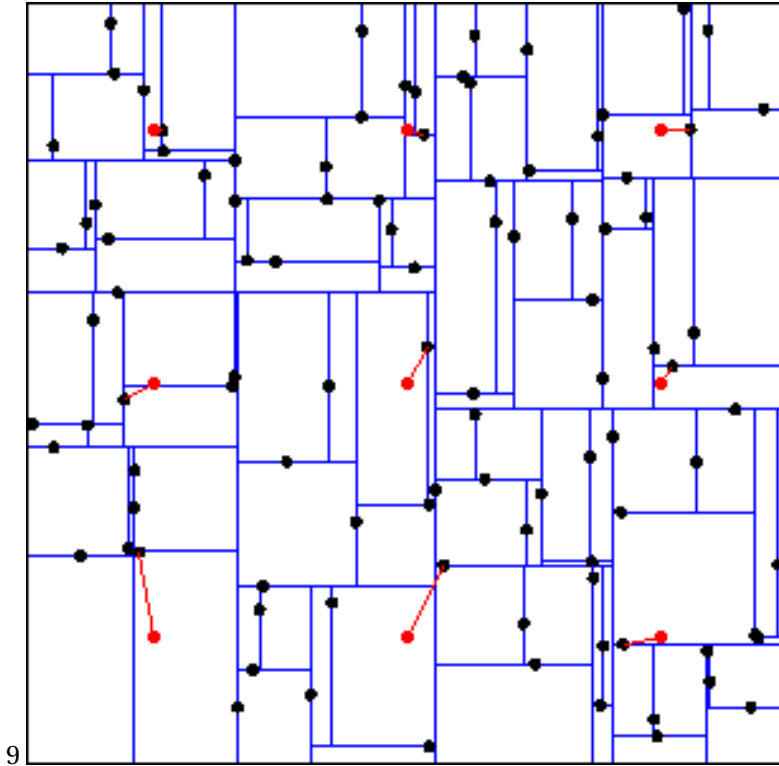


Figure 5.1: Spatial Division and several NN [Sourceforge \(2016\)](#)

5.2.1 Incremental Nearest Neighbor search algorithm

An incremental nearest neighbor algorithm can provide objects in order of increasing distances without defining a number of nearest neighbors in advance. This gives a great advantage in searches where the number of desired neighbors is unknown.

The main principles of the algorithm are as follows. A query object q and a file X organized by a structure T . The hierarchy is made up of the elements e_t which has several different types, $t = 0, \dots, t_{max}$. Every element represents a subset of X with e_0 of type 0 which is a single object in X . The element e_t can give rise to several child elements of type 0 to $t - 1$, in this way the search task for e_t is composed of smaller sub-tasks. The elements of type t have a related distance function $d_t(q, e_t)$ that calculates the distance from a query object q [Hjaltason and Samet \(1999\)](#).

Starting the algorithm initializes the queue of waiting requests at the root of the search structure, this is referred to as a priority-queue. The main loop e_t which is closest to q is taken off the queue and if it is an object it is reported as the next nearest neighbor. Else the children of e_t is inserted into the priority queue [Hjaltason and Samet \(1999\)](#).

In the experiment results of Distance Browsing in Spatial Databases, it was shown that the number of node accesses is $O(k + \sqrt{k} + \log N)$. k is the number of neighbors and N is the size of the data set. They also proved that, at each step in the execution, the incremental nearest neighbor algorithm was optimal with respect to the spatial data structure. This again means that a minimum of nodes were visited in order to report each object. By adapting the algorithm to the R-tree structure only the minimum of objects is visited. These are the objects with minimum bounding rectangles that lie within a distance of $d(q, o_k)$ of q , where o_k is the k^{th} neighbor and q is the query object [Hjaltason and Samet \(1999\)](#).

5.2.2 Voronoi Diagram Algorithm in R-Tree

This algorithm is based on the Voronoi diagram, which is a partitioning of a plane into regions by measuring distance to points in a specific subset of the

plane such as shown in figure 5.2. R-trees are excellent at getting to the "nearest neighborhood" of the search query, by traversing its tree structure fast. While they do this first phase well, the second phase of selecting from this subset is slow, and leads to several unnecessary investigation of nodes [Sharifzadeh and Shahabi \(2010\)](#).

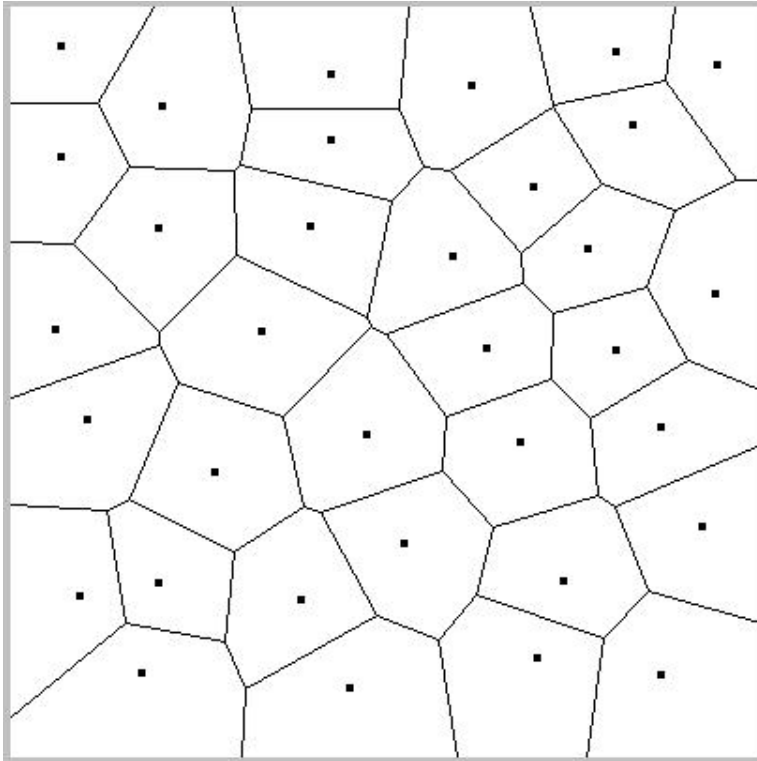


Figure 5.2: Voronoi Diagram [Sharifzadeh and Shahabi \(2010\)](#)

Several studies show that the Voronoi diagrams are extremely efficient in exploring an NN subset, but because they lack an efficient access method, the time to get to this subset is slow. The VoR-Tree incorporates Voronoi diagrams into the R-tree as shown in figure 5.3, taking the best from both and combines them into one powerful data structure to use in NNS. The rough rectangle structure of the R-tree lets us get in proximity of the query result in logarithmic time, and

the Voronoi diagram gets us to the result by the finer polygons of the Voronoi diagram [Sharifzadeh and Shahabi \(2010\)](#).

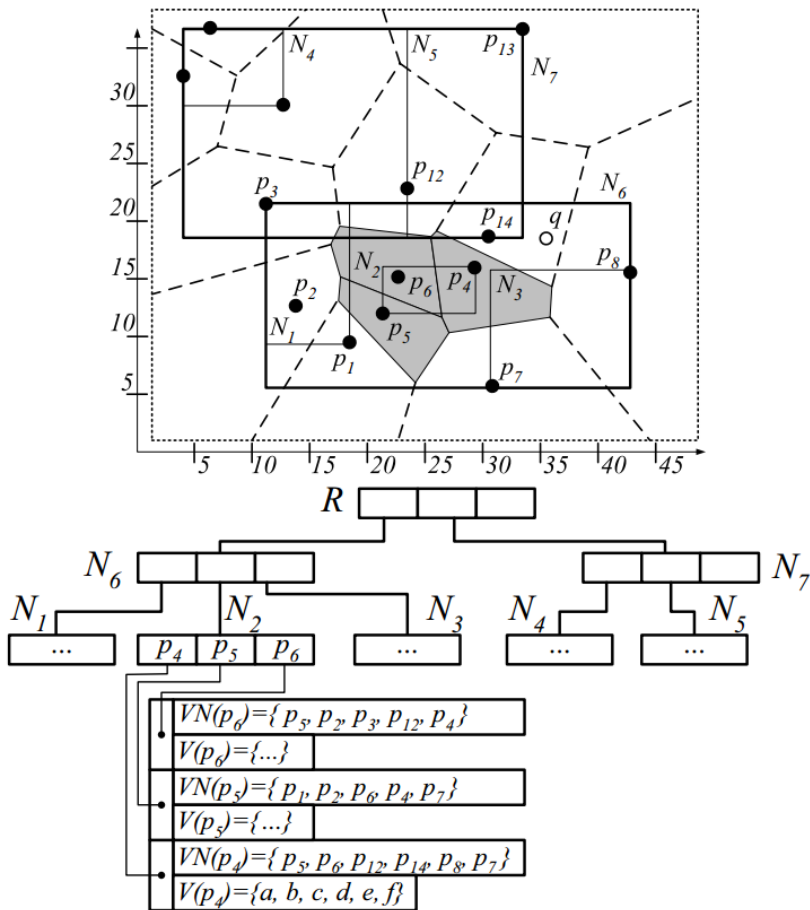


Figure 5.3: The Voronoi diagram within an R-Tree - VoR-Tree [Sharifzadeh and Shahabi \(2010\)](#)

Given a query q , kNN query gives the k closest points to q in the data set P . The complexity for I/O is in terms of Voronoi records and R-tree nodes retrieved by the algorithm. In this way the I/O complexity of kNN in VoR-Tree is $O(\phi(|P|) + k)$ where $\phi(|P|)$ is the complexity for localizing the first NN of q [Sharifzadeh and Shahabi \(2010\)](#).

5.3 Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) is a type of algorithm for solving either the approximate NN or the exact NN in high dimensional spaces. By grouping points in space into 'buckets' based on some distance measurements on the points one finds the nearest neighbors. The algorithm is divided into two main features: the width, denoted k and the number of hash tables noted L . An LSH family F is defined for metric space by $M = (M, d)$ with a threshold $R > 0$ and an approximation factor $c > 1$. F is a family of functions $h : M \rightarrow S$ for mapped elements from the metric space into a bucket $s \in S$ [Andoni and Indyk \(2008\)](#).

The first task is to obtain a random hash function g by concatenating k randomly chosen hash functions from a family F . In the preprocessing step all n points are hashed from the set S into each of the L hash tables, using $O(n)$ memory with standard hash functions. Now given a query point q , the algorithm starts to iterate over the hash functions L of g . For all g 's considered data points are retrieved that are hashed into the same bucket as q . The process stops when a point with distance cR from q is located [Andoni and Indyk \(2008\)](#).

The time complexity of the LSH algorithm for NNS is divided into the preprocessing time and the search time. Preprocessing takes $O(nLkt)$, t is the time for a function $h \in F$ to be evaluated on an input of point p . The space used is $O(nL)$ in addition to the space needed for storing the data points. After all this the query time is $O(L(kt + dnP_k^2))$ [Andoni and Indyk \(2008\)](#).

5.4 Pivot Based Nearest Neighbor Algorithms

These next algorithms with basis in the Approximation and Eliminating Search Algorithm (AESAs) uses pivot based methods. This means that it selects some points as pivots from the dataset and classifies all the other elements according

to their distance from the pivots such as shown in figure 5.4.

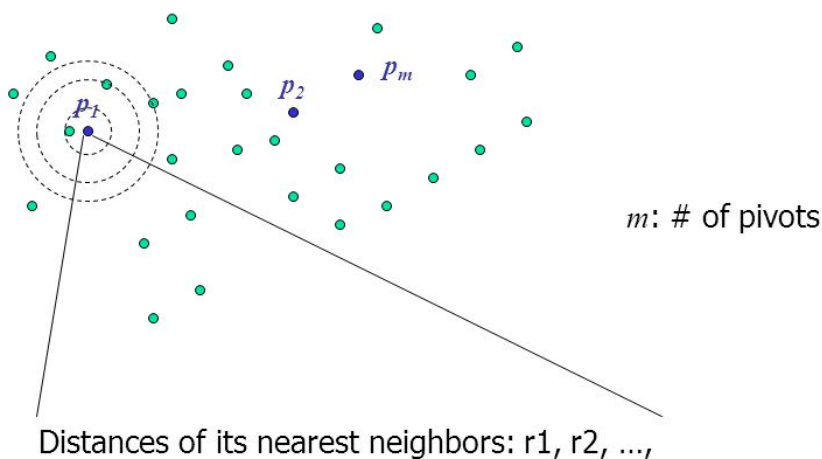


Figure 5.4: Visualizing Pivots [Jin Liang \(2004\)](#).

5.4.1 Approximating and Eliminating Search Algorithm

The AESA needs to preprocess a set of candidate points. When this is done the time complexity to answer nearest neighbor queries is $O(n)$. It reduces the number of computations per query to $O(1)$ on average. The essence is to pre-compute distances between all candidate pairs, this is where $O(n^2)$ comes from. The distances between all pairs of the n database objects, computed during the initial phase of the AESA procedure is held in an $n \times n$ matrix. The pre-computed distances are used to find better and better lower bounds from the target to each of the candidates as shown in figure 5.5 [Tavian \(2016\)](#).

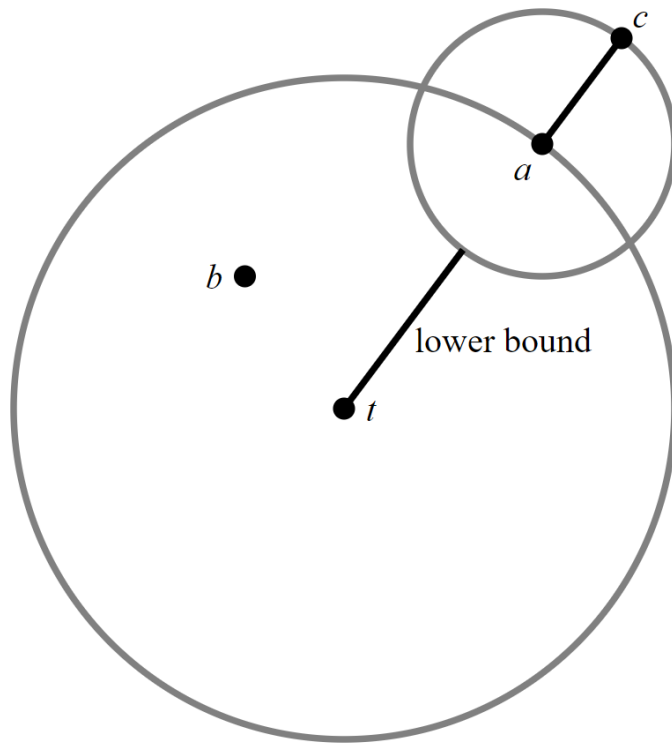


Figure 5.5: Lower Bounds [Tavian \(2016\)](#).

In figure 5.5 one can see the target noted t , b is the best match found so far, a is the current candidate, and c is another candidate waiting to be visited. Now by calculating $d(t, a)$ and utilizing the pre-computed value of $d(a, c)$, c is eliminated without having to compute $d(t, c)$ [Tavian \(2016\)](#).

If the lower bound is larger than the distance to the nearest candidate, c cannot be the nearest neighbor. AESA utilizes the algorithm design pattern of best-first branch and bound, utilizing the lower bound to discard candidates. The lower bound is also used as a heuristic to select the next current candidate [Tavian \(2016\)](#). A range query $R(q, r)$ picks an object p at random and uses it as a pivot. The distance from the query q to the candidate p is calculated, this is used in the pruning process. To prune an object it has to satisfy the statement

$|d(q, p) - d(p, o)| > r$, this means that the lower bound is greater than the query radius. This process is repeated until the set of non-discarded objects is as small as it needs to be. The objects that are left are compared directly with q . This means that the $d(q, o)$ distances are evaluated and the objects holding for $d(q, o) \leq r$ are returned. It has a query time of $O(1)$, quadratic space complexity $O(n^2)$ and quadratic construction complexity. This makes AESA only applicable to smaller data sets [Dohnal \(2004\)](#).

5.4.2 Linear Approximating and Eliminating Search Algorithm

The Linear Approximating and Eliminating Search Algorithm (LAESA) is an improved version of AESA, it uses a linear array of distances and solves the quadratic space drawback. This version stores distances from objects to a fixed number of pivots m . The matrix is now an $n \times m$ matrix, and not the $n(n-1)/2$ entries stored by AESA. This again has a tradeoff, the new problem that arises is choosing the appropriate pivots. Pivots are selected to be as far away from each other as possible. The search process is almost exactly the same as in AESA but some objects will not be pivots, because of this we cannot select the next pivot in the non-discarded objects up until now, since we may have eliminated some of the pivots already. In the first step the search algorithm eliminates objects using the pivots. Second all objects still left are directly compared with the query object q . The space complexity and construction time in regards to LAESA is $O(mn)$ and the search complexity is $m + O(1)$ [Dohnal \(2004\)](#).

5.4.3 Tree Linear Approximating and Eliminating Search Algorithm

The Tree Linear Approximating and Eliminating Search Algorithm (TLAESA) is one of the fastest algorithms for NNS in metric space. This algorithm greatly reduces the number of distance computations needed by using a branch and bound approach. By traversing the search tree by depth first, the distance matrix

is only referred to when a node is visited, this is to skip unnecessary traversal of the tree. As for the actual distance, it is only computed at the leaf node [Tokoro et al. \(2006\)](#). The time needed is linear for the LAESA and sub-linear for the TLAESA as a function of the number of prototypes. However, the LAESA was faster than TLAESA for not very large sets of prototypes or for expensive distance calculations.

The number of computations for a search is independent of the database size, the average time complexity grows only sub-linearly with the size of the database [Mícó and Oncina \(1998\)](#). Even though TLAESA performs on this level there has not been a lot of attention on the index build complexity. The index build complexity grows at $O(n \log n)$ with the database size. Because of this TLAESA is not suitable for interactive systems, where the index needs to be built each time new data is inserted into the database.

The TLAESA has two kinds of data structures that it uses, one distance matrix mentioned earlier, like the ones in AESA and LAESA and a search tree. The distance matrix holds the distances from objects to the pivots. The search tree holds all the objects in a hierarchy. The search algorithm traverses the tree and uses the distance matrix to skip parts of the branches [Tokoro et al. \(2006\)](#). The search algorithm has the basic steps: A query object q and the distance between q and the pivot are computed. The result is stored in an array, the closest object to q in the set of pivot is selected as the nearest neighbor candidate [Tokoro et al. \(2006\)](#).

The tree built by TLAESA is a Generalized Hyperplane Tree-like structure using the same m pivots as in LAESA. The CPU time is between $O(\log(n))$ and $O(mn)$, where n is the number of database sets and m is a fixed number of pivots, the search time is the same as LAESA, $m + O(1)$ [Dohnal \(2004\)](#).

5.5 The best fit for MySQL

Even though TLAESA is a highly performing algorithm it seems to be too far away from existing structures found in MySQL. The integration is surely possible, but the pros and cons swing in the incremental nearest neighbor algorithms direction. As of now MySQL has a type of R-tree for storing the spatial objects. The general incremental algorithm can at that point be used virtually unchanged, in regards to R-tree. If though the spatial objects are stored separate from the R-tree, the difference will be the use of the bounding rectangles stored in the leaf nodes.

Both of these algorithms are sub-linear in search time and space complexity. In contrast to the TLAESA algorithm the incremental nearest search algorithm does not need to specify k (k -nearest neighbors) beforehand. This can be useful in some situations where $m > k$, if not the k -nearest algorithm would have to be re-invoked for m neighbors.

The incremental nearest algorithm can be expanded to handle other geometries if needed in the future. The time complexity for this algorithm was $O(k + \sqrt{k} + \log N)$ compared to the time complexity of TLAESA, $O(\log(n)) + m + O(1)$, both are sub linear.

Lastly, by using the incremental nearest neighbor algorithm there is a possibility for an easy extension to perform furthest neighbor searches.

Chapter 6

Nearest Neighbor Searches in Other Database Management Systems

Here we take a look at how other DBMS finds the nearest neighbors using spatial operators.

6.1 Oracle RDBMS - Oracle Spatial and Graph

The spatial feature nearest neighbor has been part of the Oracle environment for some time. They have both the feature for returning the nearest neighbor and to return the distance of the nearest neighbor in two different spatial operators. Oracle DBMS has the operators `SDO_NN()` to find the nearest objects and `SDO_NN_DISTANCE()` to find the distance of and object returned by `SDO_NN()` [Oracle \(2017\)](#).

6.1.1 SDO_NN

The function SDO_NN works as in the example below. The SDO_BATCH_SIZE keyword specifies the number of rows that will be evaluated at a time, this can only be used with R-tree indexes. This is when the SDO_NN function needs to be evaluated several times to ensure the SDO_NN function returns the desired number of results that corresponds to the WHERE clause.

In this example the SDO_BATCH_SIZE keyword is used as the bakeries table has several types of bakeries and one asks for the nearest Dutch bakery to your apartment. It needs to be in the form SDO_NN(geometry1, geometry2, param[number]) [Oracle \(2017\)](#).

```
SELECT b.name FROM bakeries b WHERE
    SDO_NN(b.geometry, :my_apartment,
        'sdo_batch_size=10') = 'TRUE'
    AND b.cuisine = 'Dutch' AND ROWNUM <=2;
```

The ROWNUM<=2 clause is to limit the number of results to 2 WHERE cuisine is Dutch. It is worth noting that if the SDO_BATCH_SIZE is not specified as the query is structured, and if SDO_NUM_RES=2 is specified but not ROW_NUM<=2, only the nearest bakery is considered even though the cuisine is not Dutch. When this happens and the first two nearest bakeries are not of the cuisine "dutch" the query will return no rows.

Knowing your data is key to better performance, setting the number of candidates rows to a number likely to satisfy the WHERE clause is a good practice. Using our example of finding the nearest Dutch bakery, if 5 percent of the bakeries closest to the apartment are Dutch and we are looking for the two closest ones, a value of 40 would be best. If this was an extremely Dutch bakery area and 20 percent of

the bakeries were Dutch a value of 10 would give the best performance. If the percentage of data searched for is small one would use a bigger batch size, and if the percentage is large one would use a smaller batch size [Oracle \(2017\)](#).

If you do not know much about the data, it is better to specify `SDO_BATCH_SIZE=0`, this will cause the system to calculate the best batch size for your result. This again will cause some overhead, and may not produce the most optimal batch size [Oracle \(2017\)](#).

6.1.2 SDO_NN_DISTANCE

The function `SDO_NN_DISTANCE` works as shown in the example below:

```
SELECT /*+ INDEX(c cola_spatial_idx) */
       c.mkt_id, c.name, SDO_NN_DISTANCE(1) dist
FROM   cola_markets c
WHERE  SDO_NN(c.shape, sdo_geometry(2001, NULL,
                                     sdo_point_type(10,7,NULL), NULL, NULL),
            'sdo_num_res=2', 1) = 'TRUE' ORDER BY dist;
```

The format for this method is `SDO_NN_DISTANCE(number)`, where the number must match the last parameter in the call to the `SDO_NN` operator [Oracle \(2017\)](#).

6.2 PostGIS extension to PostgreSQL

PostGIS is a geographical support feature to the object relational database PostgreSQL. The spatial indexing in PostGIS is done with an R-Tree-over-GiST scheme [PostGIS \(2015\)](#). By using Generalized Search Trees (GiST) both B+-trees and R-trees can be implemented without much work as extensions, resulting in one code base for indexing multiple dissimilar applications [Hellerstein et al. \(1995\)](#). One reason PostGIS switched from regular R-tree to this R-Tree-over-GiST is because of the R-tree indexes in PostgreSQL. The indexes could not handle objects

larger than 8K in size while GiST indexes are able to. This is due to the "lossy" feature of substituting bounding boxes for the object itself [PostGIS \(2015\)](#).

To perform nearest neighbor searches PostGIS has the operator ST-Distance. The technique PostGIS uses is suitable for even very large tables with variable densities and has a high performance. The system works by evaluating distances between bounding boxes inside the PostGIS R-Tree index. Since the index is built by using the bounding boxes of the geometries, distances separating geometries which are not points will not be exact. The distance will be the length between the bounding boxes of the geometries [Boundless \(2016\)](#).

The index-based syntax of the kNN query has a special "index-based distance operator" at the ORDER BY clause as shown bellow:

- distance between box centers < - >
- distance between box edges < # >

Here is an example of how a kNN query would look in PostGis [Geographic Information Systems \(2017\)](#).

In this first query we know our data somewhat, and realize that there is no nearest neighbor beyond 300 units. This will improve our query a lot.

```
SELECT g1.gid As gref_gid, g1.description
As gref_description, g2.gid As gnn_gid,
       g2.description As gnn_description
FROM tableA As g1, tableA As g2
WHERE g1.gid = 1 and g1.gid <> g2.gid
AND ST_DWithin(g1.the_geom, g2.the_geom, 300)
ORDER BY ST_Distance(g1.the_geom,g2.the_geom)
LIMIT 10
```

Most of the time we do not know such intricate details of our data, and we will need to perform a search on the entire dataset, or we may end up not finding the nearest neighbors.

6.3 Microsoft SQL Server

There are several methods for writing a nearest neighbor query. But for the search to use a spatial index the query must be on the form shown in the documentation [8.7](#).

Syntax

```

VB
SELECT TOP ( number )
  [ WITH TIES ]
  [ * | expression ]
  [ , ... ]
FROM spatial_table_reference, ...
  [ WITH
    (
      [ INDEX ( index_ref ) ]
      [ , SPATIAL_WINDOW_MAX_CELLS = <value> ]
      [ ,... ]
    )
  ]
WHERE
  column_ref.STDistance ( @spatial_object )
  {
    [ IS NOT NULL ] | [ < const ] | [ > const ]
    | [ <= const ] | [ >= const ] | [ <> const ] ]
  }
  [ AND { other_predicate } ]
}
ORDER BY column_ref.STDistance ( @spatial_object ) [ ,...n ]
[ ; ]

```

Figure 6.1: The structure of the Nearest Neighbor Search in Microsoft SQL Server as show in their documentation [Documentation \(2017b\)](#).

The TOP and ORDER BY clauses are used to execute the Nearest Neighbor query in spatial data. The ORDER BY has the STDistance() function to calculate the distances, while the TOP keyword sets the number of wanted returned results [Documentation \(2017b\)](#).

- There must be a spatial index on the column and STDistance() can only use that column in its WHERE and ORDER BY clauses.

- The WHERE clause needs to have a STDistance() function.
- The ORDER BY clause needs to use the STDistance() function.
- The sorting order must be ASC for the first STDistance() expression.
- Return of NULL by STDistance() must be filtered out.

6.4 MySQL

In MySQL there is currently no special case for doing nearest neighbor searches, which is a real competitive disadvantage over the other DBMS's. But for now to do the nearest neighbors search in MySQL is a matter of selecting a point to search from and sorting the table with a distance from that point made by a distance calculation. This method will be used to compare the improvement against the new nearest neighbor algorithm.

```
mysql> SET @first = Point(1,1);
mysql> SET @second = Point(2,2);
mysql> SELECT ST_Distance(@first, @second);
+-----+
| ST_Distance(@first, @second) |
+-----+
|           1.4142135623730951 |
+-----+
```

The ST_DISTANCE function returns the distance between the two values provided in Euclidean distance.

```
SET @g1 = POINT(-739851300, 407588960);
SELECT ST_ASTEXT(geom) FROM new_york
ORDER BY ST_DISTANCE(geom, @g1)
LIMIT 1;
```

Here one can see the `ST_DISTANCE` function used to sort (`ORDER BY`) the table by the distance from the set point `g1`. This is a table scan of the table, something that takes a lot of time and gets worse and worse as the table grows. Even with a small `LIMIT` like 1, the process needs to scan the whole table.

Chapter 7

The most optimal algorithm found

The most optimal algorithm found for MySQL is the Incremental Nearest Neighbor Algorithm (INNA) proposed by Gisli R. Hjaltason and Hanan Samet in their paper Distance Browsing in Spatial Databases. The reason for choosing this algorithm comes down to certain features desirable specially by implementing this algorithm. Mainly the option to not specify the k in the k -nearest neighbors beforehand. This chapter explains in detail how the algorithm works and its strength and weaknesses.

7.1 The Incremental Nearest Neighbor Algorithm

The INNA has a predicted node access of $O(k + \sqrt{k} + \log N)$, where k is the number of neighbors and N is the size of the data within the search. Further the INNA is optimal when it comes to the spatial data structure that is used, this mean that a minimum number of nodes is visited, this is important as disk I/O is one of the bottlenecks of the execution [Hjaltason and Samet \(1999\)](#). As with InnoDB's R-tree the leaf nodes stores only the minimum bounding rectangles (MBR) of

the spatial object and a pointer to the objects itself. The improvement over the regular k-nearest algorithm was noted to be over 50%. This is because of how few times the INNA needs to calculate the distance to the actual data objects, instead it prunes the data by only calculating the objects MBR [Hjaltason and Samet \(1999\)](#).

7.1.1 Getting to The Leaf

The pseudo code below [Hjaltason and Samet \(1999\)](#) is an example of how a regular top-down traversal of the tree looks to find the leaf node containing the search object. The traversal is initiated with the root node of the spatial index as the second argument.

Algorithm 1 Find Leaf Node

```

1: function FINDLEAF(QueryObject, Node)
2:   if QueryObject is in node Node then
3:     if Node is a leaf node then
4:       Report leaf node Node
5:     else
6:       for each Child of node Node do
7:         FINDLEAF(QueryObject, Child)
8:       end for
9:     end if
10:  end if
11: end function

```

The first task is to extend the algorithm to locate the object closest to the query object. Once a leaf node containing the *QueryObject* has been located in line 3, one starts by examining the objects contained in that node. The closest object could be in another node though. To find this node it may require unwinding the recursion to the top and descending again but this time deeper in the tree. To resolve this, the recursion stack is replaced by a priority queue. As leaf nodes are processed nodes and objects are also put on the queue, sorted by the distance from the query object [Hjaltason and Samet \(1999\)](#). Within the queue a node is

examined once it is at the first position on the queue, at this time all other nodes and objects have been examined and calculated the distance for.

7.1.2 The General Incremental Nearest Algorithm

This 7.1.2 is the incremental nearest neighbor algorithm [Hjaltason and Samet \(1999\)](#). In line 2-3 the queue is initialized. In line 10 the next closest object is found. At this point it would be possible for another routine to take control, re-summing the algorithm at a later time to get the next closest object, or terminating it if no more objects are needed. For some types of spatial indexes a object can span several nodes, meaning that the algorithm must check if the object has already been reported, it does so in line 13. In line 7-8 the duplicates of objects are removed from the queue, because many priority queue implementations has inefficient detection for duplicates among the queue elements. [Hjaltason and Samet \(1999\)](#).

7.2 Adapting to Rtrees

This section the general incremental algorithm is adapted to R-trees, by using some of the unique properties of R-Trees. This is the version that was implemented and tested in MySQL during this master thesis. When the spatial objects are stored external to the R-tree, in the sense that the leaf nodes contain only the MBR of objects, the algorithm performs even better. The reason for this is the usage of the MBR as pruning devices, this means the disk I/O is considerably reduced [Hjaltason and Samet \(1999\)](#). MySQL stores only the MBR of objects in the leaf node, so one is able to take advantage of this feature. One of the perks of the R-tree is that each object is only stored once, meaning looking for duplicates is unnecessary. One can then also remove the secondary ordering of the priority queue of the general INNA.

The input to the INNA is the query object q and an R-Tree R , in MySQL one would

Algorithm 2 Incremental Nearest Neighbor Algorithm

```

1: function INCNEAREST(QueryObject, SpatialIndex)
2:   Queue = NEWPRIORITYQUEUE()
3:   ENQUEUE(Queue, SpatialIndex.RootNode, 0)
4:   while not ISEMPY(Queue) do
5:     Element = DEQUEUE(Queue)
6:     if Element is a spatial object then
7:       while Element = FIRST(Queue) do
8:         DELETEDFIRST(Queue)
9:       end while
10:      Report Element
11:    else if Element is a leaf node then
12:      for each Object in leaf node Element do
13:        if DIST(QueryObject, Object)  $\geq$  DIST(QueryObject, Element)
14:          then
15:            ENQUEUE(Queue, Object, DIST(QueryObject, Object))
16:          end if
17:        end for
18:      else ▷ /* Element is a nonleaf node */
19:        for each Child node of node Element in SpatialIndex do
20:          ENQUEUE(Queue, Child, DIST(QueryObject, Child))
21:        end for
22:      end if
23:    end while
24: end function

```

pass the root node and from there get access to the whole tree. To use the fact that the leaf only stores the MBR a third type of queue element is introduced, the object bounding rectangle (OBR). The distance to an OBR is never larger than the distance of the object inside. When inserting the elements into the queue the distance is measured from the query object to the OBR, only when this OBR is at the top of the queue is the real distance calculated. Now if this object is closer to the query object than the next element in the priority queue it is the next nearest neighbor. If this is not the next nearest neighbor the object is inserted into the queue with the real distance from the query object [Hjaltason and Samet \(1999\)](#).

Looking at the INNA adapted to R-trees [7.2 Hjaltason and Samet \(1999\)](#). The priority queue is initialized in lines 2-3 and in line 10 the next nearest object is found. In line 8 the object p is queued with the real distance to the query object since this distance is larger than other elements on the queue. In another case where there were no other elements on the queue with a lower value, this object would have been the next nearest neighbor. In line 14 an OBR is queued, this has a pointer to the original object, this is where the regular INNA had a check for duplicates [Hjaltason and Samet \(1999\)](#).

7.2.1 The Priority Queue

The performance of the incremental nearest neighbor algorithm depends largely on two things, disk IO and the size of the priority queue. When the size of the queue gets larger, the cost of each operation gets higher. Finally if the queue gets too big to fit in the memory, and its content must be stored on disk, the costs of each operation increases even more [Hjaltason and Samet \(1999\)](#).

Worst case for the incremental nearest neighbor algorithm in regards to the queue size is when all leaf nodes are within distance to from the query object

Algorithm 3 Incrementing Nearest Neighbor Algorithm with R-trees

```

1: function INCNEAREST(QueryObject, R-tree)
2:   Queue = NEWPRIORITYQUEUE()
3:   ENQUEUE(Queue, R-tree.RootNode, 0)
4:   while not ISEMPY(Queue) do
5:     Element = DEQUEUE(Queue)
6:     if Element is an object or its bounding rectangle then
7:       if Element is the bounding rectangle of Object and not
       ISEMPY(Queue) and DIST(QueryObject, Object) > FIRST(Queue).Key then
8:         ENQUEUE(Queue, Object, DIST(QueryObject, Object))
9:       else
10:        Report Element (or if bounding rectangle, the associated object)
        as the next nearest object
11:      end if
12:      else if Element is a leaf node then
13:        for each entry (Object, Rect) in leaf node Element do
14:          ENQUEUE(Queue, [Object], DIST(QueryObject, Rect))
15:        end for
16:      else ▷ /* Element is a nonleaf node */
17:        for each entry (Node, Rect) in node Element do
18:          ENQUEUE(Queue, Node, DIST(QueryObject, Rect))
19:        end for
20:      end if
21:    end while
22: end function

```

and all data objects are further away. When this is the case, the leaf nodes must be processed by the incremental algorithm and all data objects, must be placed in the priority queue before the nearest neighbor can be found [Hjaltason and Samet \(1999\)](#).

7.2.2 Extension to Furthest Neighbor

With small modifications the INNA can be used to find the farthest object from the query point. This is by using the priority queue to sort the elements in decreasing order of their distance. This however is not enough because objects or nodes contained in a node n is usually further from the query object than its bounding rectangle. In this case the condition that elements are dequeued in decreasing order of distance is broken. By using the upper bound as the key for node n on distance from the query object q to an object in the sub-tree at n this can be fixed [Hjaltason and Samet \(1999\)](#).

Chapter 8

Implementation and MySQL Integration

This chapter talks about the implementation the best approach for the nearest neighbor problem in MySQL. The incremental nearest neighbor algorithm (INNA) was chosen based on several factors, including one of the fastest algorithms proposed but mainly the option to not predetermine either the scope of the dataset or the k -nearest neighbors beforehand. The INNA can be asked for any given number of neighbors in any given amount of data. If the number of neighbors asked for is too great, the algorithm will return just what it got, while others may have terminated and returned nothing. In addition one does not have to set a scope for the data, meaning the INNA will search incrementally until it finds the desired amount of neighbors, while the more common result is that one would have to re-specify and increase the search area within the data.

The nearest neighbor algorithm made here will be part of a future MySQL release, pending several stages of testing within the MySQL framework and some other depending work-logs not yet implemented. Some crucial parts of the implemen-

tation will be shown here while the complete implementation can be viewed in the appendices.

8.1 InnoDB

InnoDB is a general-purpose default storage engine of MySQL. In this section we look at the structure of some of the internal working mechanism of InnoDB where the code is built into.

8.1.1 The structure of the R-tree

The split used in InnoDB's R-tree SPATIAL indexes is a version of Guttman's quadratic split. In InnoDB's spatial index the objects Minimum Bounding Rectangle (MBR) is the only thing included in the index. The actual data is not stored in the R-tree. This makes the index entry size very small and compact, even with large and complex data of geometries. The index is used as a first screening process to pick out the possible candidates for the search criteria in the form of bounding boxes. From here the data stored in the primary index is sent to the server for a detailed screening [Jimmy \(2014\)](#).

8.1.2 Index header

For each index page [8.2](#) there is a header with a fixed-width and it has the following structure as show in figure [8.1](#). The most interesting part of the header for this implementation and in regards to R-trees is the Page Level. The page level is the level of this page in the index. The leaf pages are at level 0, in regards to the R-tree, this would be the bounding box of the object e.g. geometries. The levels increment upwards from there, such that the root of the R-tree will have the highest page level [Jeremy \(2013\)](#). In between the leaf and the root are the internal pages with levels from 1 to root.

INDEX Header

38	Number of Directory Slots (2)
40	Heap Top Position (2)
42	Number of Heap Records / Format Flag (2)
44	First Garbage Record Offset (2)
46	Garbage Space (2)
48	Last Insert Position (2)
50	Page Direction (2)
52	Number of Inserts in Page Direction (2)
54	Number of Records (2)
56	Maximum Transaction ID (8)
64	Page Level (2)
66	Index ID (4)
74	

Figure 8.1: Index header structure in InnoDB [Jeremy \(2013\)](#).

8.1.3 Index Page

Everything is an index in InnoDB, and that has some implications on the physical structure. The records on a index page are sorted in their binary sequence, this means that it is not necessarily related to their MBR [MySQL-Developer \(2017\)](#). All tables has a primary key, even if one does not specify one InnoDB will make one for one, this is the first non-NULL unique key. The non-PRIMARY key fields (row data) are stored in the PRIMARY KEY index structure. The secondary keys are kept in an identical index structure and keyed on the KEY fields [Jeremy \(2013\)](#). In figure 8.2 User record is the actual data stored. All of these records has a variable sized header and the actual column data. The header has the next-record pointer which also stores the offset to the next record, making a linked list [Jeremy \(2013\)](#).

INDEX Overview

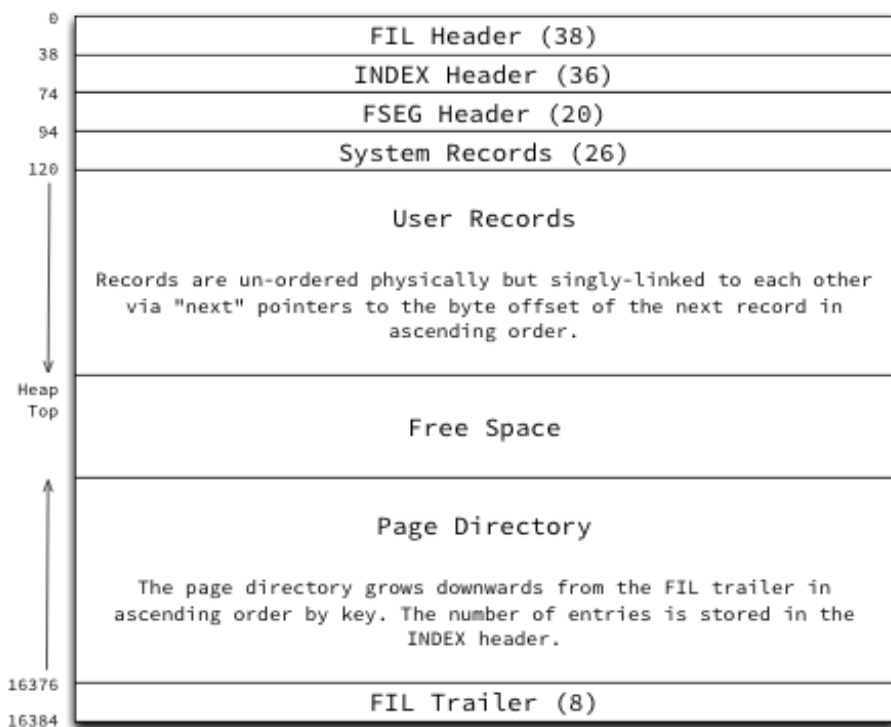


Figure 8.2: Index page structure in InnoDB [Jeremy \(2013\)](#).

8.1.4 System records

Each index page in InnoDB contains two system records, infimum and supermum. These are at fixed locations at the offset 99 and 112 within a page. The index system records have the following structure shown in figure 8.3 [Jeremy \(2013\)](#). The **infimum record** has a value lower than any other key in the page and the **supermum** has a key higher than all the other keys in a page [Jeremy \(2013\)](#). They are very useful when traversing the tree, as one can stop the traversing when at a infimum or a supermum, and in such a way know one has come to the end

of the index page.

INDEX System Records

94	Info Flags (4 bits)
	Number of Records Owned (4 bits)
95	Order (13 bits)
	Record Type (3 bits)
97	Next Record Offset (2)
99	"infimum\0" (8)
107	Info Flags (4 bits)
	Number of Records Owned (4 bits)
108	Order (13 bits)
	Record Type (3 bits)
110	Next Record Offset (2)
112	"supremum" (8)
120	

Figure 8.3: System records in InnoDB [Jeremy \(2013\)](#).

8.2 The code

In this section we take a look at some of the important parts within the code implemented in MySQL.

8.2.1 The Shortcut

While this task was a part of the internal work log system, other non-implemented tasks is necessary to have a finished product. The entry query to this nearest neighbor method is a hard-coded internal hook as show in figure 8.4 for query purposes. For testing and verification of the code along the way the query was structured as show bellow, where the ST_Within() function is a dummy insert of sorts.

```

SELECT ST_ASTEXT(geom) FROM new_york
USE INDEX(geom_SPATIAL)
WHERE ST_Within(geom,ST_POLYGONFROMTEXT(
'POLYGON((
-532 -532,-532 532,532 532 -532,-532 -532))'))
LIMIT 24000;

```

```

8957 case HA_READ_MBR_WITHIN:
8958     return(PAGE_CUR_NEAREST_NEIGHBOR); //PAGE_CUR_WITHIN

```

Figure 8.4: Nearest neighbor hook

8.2.2 The distance calculation

The Calculation done for the minimum bounding rectangles are done such that it measure the distance from a point to a line. The reason for doing a point to a line and not just a point to a point (4 corners) is shown in figure 8.5. Here one can see two minimum bounding rectangles (mbr) labeled A and B and a point p1. There are three distances d1, d2 and d3. As the algorithms tries to find the next nearest mbr if it had used a point to point based calculation the next nearest mbr would have been B at 3.16, and the next nearest mbr A at 3.46. When in reality the nearest mbr is A with the distance of 3. The method in figure 8.6 is called for each of the four lines of the mbr.

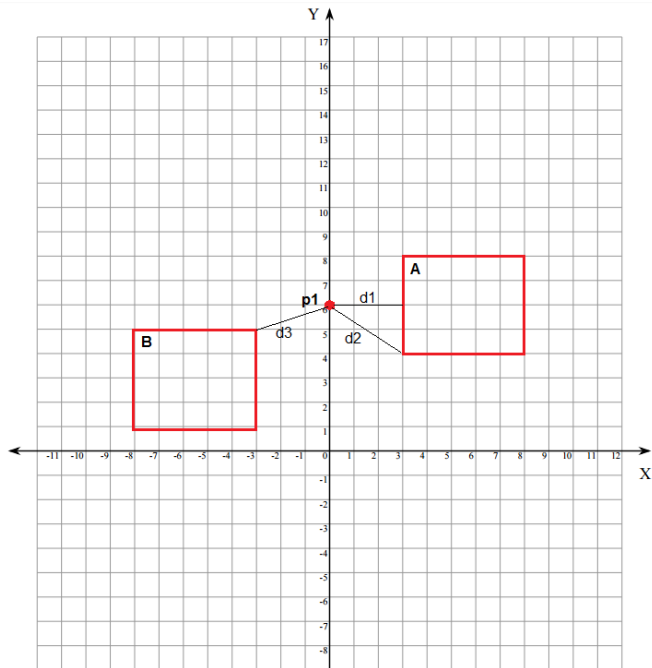


Figure 8.5: Point and minimum bounding rectangles.

```

double
linesegment_distance(point_xy* query, double x1, double x2, double y1, double y2)
{
    // Return minimum distance between line segment and point p
    double xp = query->x;
    double yp = query->y;

    // This will avoid a square root
    double l2 = (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1);

    // t is how far along the line that the projection falls
    double t = ((xp - x2)*(x1 - x2) + (yp - y2)*(y1 - y2)) / l2;
    t = std::max(0.0, std::min(1.0, t));

    // The projection of point xp-yp onto the line is the point on the line closest to xp-yp
    double distance = (xp - (x2 + t*(x1 - x2)))*(xp - (x2 + t*(x1 - x2)))
        + (yp - (y2 + t*(y1 - y2)))*(yp - (y2 + t*(y1 - y2)));
    return distance;
}

```

Figure 8.6: The point to line segment calculation.

8.2.3 Limitations

One of the major drawbacks of the existing code is how matches are retrieved.

```

if (matched_page == page_get_page_no(page_align(get<0>(prielement).data.rect)) ||
    rtr_info->matches->block.page.id.page_no() == NULL && (rtr_info->matches->used + 100) < UNIV_PAGE_SIZE){
    rtr_leaf_push_match_rec(get<0>(prielement).data.rect, rtr_info,
        offsets, page_is_comp(page_align(get<0>(prielement).data.rect)), get<1>(prielement));
}
else
{
    //Push this popped element back into the queue
    rtr_info->priority_q.push(prielement);
    //Input a save this rec that was not pushed back into the queue.
    goto exit;
}

```

Figure 8.7: Push of matches

In in the code snippet 8.7 one can see how matches are handled. If the current match page is the same page as the current nearest element found, then we can push this next nearest neighbor to the list of matches. However if the next nearest neighbor found belongs on another page, the neighbor is pushed back into the priority queue and the algorithm exits. Now the matches are dumped from the list of matches and into another list outside of InnoDB. When this is done the algorithm continues at the top of the priority queue. This going in and out of the algorithm takes the most time, and slows the execution of the algorithm down by far, this is the bottleneck of the implementation. This will be further proven with querying of a worst case dataset for the algorithm 9.2.3.

Chapter 9

Evaluating the Incremental Nearest Neighbor Algorithm in MySQL

This chapter will test the Incremental Nearest Neighbor Algorithm (INNA) implemented inside MySQL. Results from the new algorithm will be shown here and compared to the existing Table Scan method still in use in MySQL.

The goals for this chapter are to see if the new INNA can beat the existing method of doing nearest neighbor searches with different datasets. There will be comparisons against different types of datasets from worst cases to best cases and data extracted from the real world. At the end there will be a summary of the findings.

9.1 Test plan and setup

9.1.1 Incremental Nearest Neighbor Algorithm

Since there is no ST function for the INNA in MySQL yet a hook into the system was created as described in chapter 8. The resulting query for the INNA is on the form:

```
SELECT ST_ASTEXT(geom) FROM food_places
USE INDEX(geom_SPATIAL)
WHERE ST_WITHIN(geom,ST_POLYGONFROMTEXT(
'POLYGON((-532 -532,-532 532,532 532,532 -532,-532 -532))'))
LIMIT 1000;
```

9.1.2 Table Scan

The table scan is just as the name suggests, the tables are scanned entirely and calculated the distance on. At last the results is sorted, this whole process takes a lot of time. The distance calculation `ST_distance()` was used to perform the table scans sorting.

The form of the query is of:

```
SET @g1 = POINT(-739851300, 407588960);

SELECT ST_ASTEXT(geom),
ST_DISTANCE(geom, @g1)
AS distance
FROM food_places
ORDER BY distance
LIMIT 100;
```

9.1.3 Hardware and Software

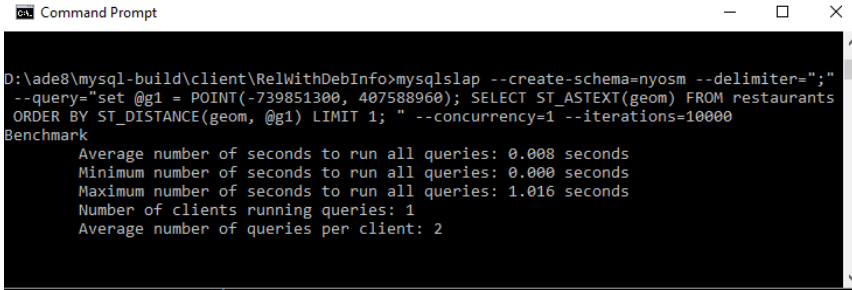
Within this thesis the testing has been done on MySQL version 8.0.0 with a release build. This is however not the latest stable version of MySQL, due to the fact that a lot of R-tree functionalities has been moved and changed by the MySQL team during the development of this code. The Nearest Neighbor patch with work log number 9440 is has been applied to this version of MySQL without any other source code changes.

The tests are done running Windows 10 Pro with the additional specification for the hardware:

- Processor: Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz 3.41GHz
- Installed memory(RAM): 32,0 GB (31,9 GB usable)
- System type: 64-bit Operating System, x64-based processor
- MySQL 64 bit Release Build

9.1.4 Load Emulation Client - mysqlslap

The load emulation client mysqlslap is a diagnostics program used to emulate the client load for a server running MySQL. It reports the timing in a precise manner and it is possible to configure the load such as needed [Documentation \(2017a\)](#). The configurations used in these queries differs slightly on the number of runs for each query, the smaller datasets are given a higher set of iterations to get a decent understanding of the time consumption. In the queries used within these tests mysqlslap has been given several query options as shown in picture [9.1](#).



```

D:\ade8\mysql-build\client\RelWithDebInfo>mysqlslap --create-schema=nyosm --delimiter=";"
--query="set @g1 = POINT(-739851300, 407588960); SELECT ST_ASTEXT(geom) FROM restaurants
ORDER BY ST_DISTANCE(geom, @g1) LIMIT 1; " --concurrency=1 --iterations=10000
Benchmark
Average number of seconds to run all queries: 0.008 seconds
Minimum number of seconds to run all queries: 0.000 seconds
Maximum number of seconds to run all queries: 1.016 seconds
Number of clients running queries: 1
Average number of queries per client: 2

```

Figure 9.1: The mysqlslap function and query.

The average number of seconds to run the queries is what will be used in the result tables. The reason Maximum number of seconds is higher is because the server loads the data into memory and the succeeding queries are read from there. This is the reason for having more iterations to get a better average on the smaller datasets where the execution time is very fast.

- **--create-schema** Schema to run the tests
- **--delimiter** Delimiter to use in the SQL statements
- **--query** The query statement to use for retrieving data
- **--concurrency** Is the number of clients to use while simulating the SELECT statement
- **--iterations** Is the number of times to run the queries

9.2 Querying

In this section we look at different spatial datasets and see how they compare. To check if the algorithm produces the correct result the results was written to a text file and then another type of distance search was used, the common Table scan of MySQL. The result files were compared in the simple text editor program Notepad++. The results one can see here is not made from the queries which also created a text file, as that would slow down the query speed.

9.2.1 Random datasets

Several random data sets were created with the [B.1](#) Python program, during the development of the algorithm to compare it against the table scan method of MySQL. Two of them will be used in this testing, 1mill and 12mill. The 12mill dataset used here has 11 483 747 points and was created by the `random.randint()` function in Python. The 12mill values ranges from -999999999 to 999999999 to make it look like the coordinate systems used in the later tested Open Street Map datasets. The 1mill dataset has 999200 points and was created with the same Python function as the 12mill dataset and has values ranging from -99999 to 99999. This dataset will be used to compare it to the Optimal and Worst case datasets.

```
+-----+-----+
| Table          | Create Table          |
+-----+-----+
| 12mill_random | CREATE TABLE '12mill_random' ( |
| 'g' geometry NOT NULL,          |
| SPATIAL KEY 'g' ('g')          |
| ) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+-----+-----+
```

9.2.2 Optimal datasets

This data set was created to see how much of a difference the optimal data would affect the query time. This data set is structured such that the next closest point is most likely in the same internal node, one step up from the previous leaf node. In regards to the MySQL pages, the next nearest neighbor is on the same page. This means that the incremental nearest neighbor algorithm (INNA) dumps a full page of matches at a time. The dataset was created by the [B.2](#) Python program and has 999200 points. The points are on the form:

```
POINT(0, 0), POINT(1, 1), POINT(2, 2),
POINT(3, 3), ...POINT(999199, 999199)
```

9.2.3 Worst case dataset

This worst case dataset will show how much of a difference this constraint of only being able to dump matches from one page at a time matters. To do this, we set the query point to POINT(0, 0) and make it such that for nearly each match, the algorithm has to exit, dump the matches, and come back again. This is very time consuming and is one of the limitations of the R-tree in MySQL mentioned in 8.2.3. This dataset was created by the Python program B.3 and is on the form:

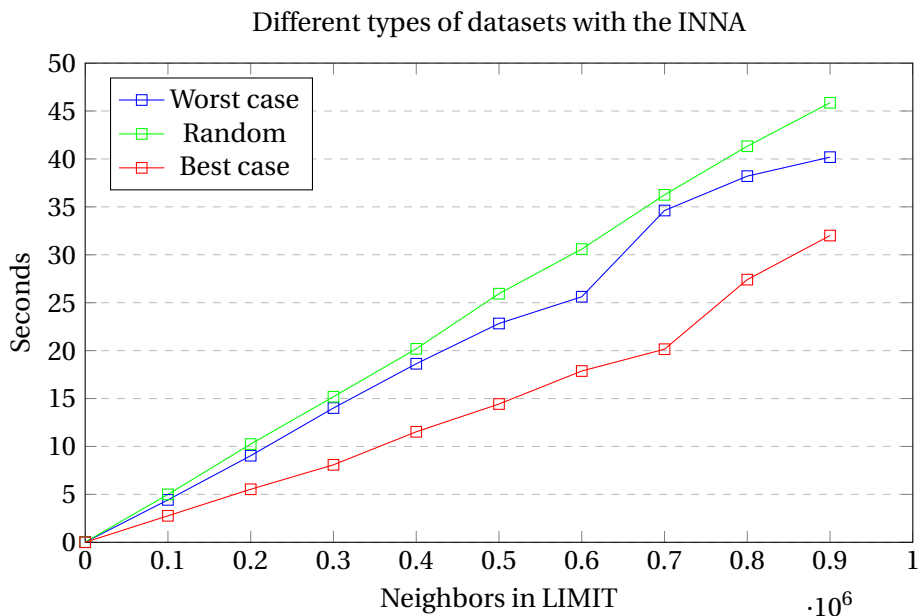
```
POINT(0, 0), POINT(-1, -1), POINT(2, 2),
POINT(-3, -3), ...POINT(-999199, -999199)
```

9.2.4 Best vs. Worst

We look at three different types of datasets to see how much of a difference the data can make. We compare the three tables `worst_case`, `1mill` and `best_case` against each other with the INNA.

Neighbors	Worst case	Random	Best case
1	0.014	0.018	0.003
100 000	4.426	4.998	2.765
200 000	9.037	10.248	5.535
300 000	14.010	15.189	8.085
400 000	18.629	20.189	11.531
500 000	22.829	25.940	14.429
600 000	25.620	30.595	17.871
700 000	34.618	36.251	20.148
800 000	38.203	41.324	27.406
900 000	40.186	45.856	32.007

Table 9.1: Worst to Best Query Results



As one can see what was supposed to be a worst case dataset was in fact not. The reason for this may be that the data was not enough spread to cause the INNA to exit enough times, in other words the matches were not spread enough in the page structure to cause maximum page swaps. One can think if the data had been small clusters around 400 in each, in every direction, this would have produced a worst case. The random data set had bigger spread in its data, and may be the reason it performed worse. The best case dataset performed significantly better as planned, dumping a full page of points for each time the INNA exited.

9.2.5 Open Street Map

To give a real life representation of how the INNA would work a Open Street Map dataset over New York was acquired at Mapzen [Mapzen \(2017\)](#). Since these are points over a real city, the data is more clustered. The smaller tables extracted from this data set is over restaurants and places to get food. The next largest table is of points of amenities throughout the city. Finally we search the whole dataset

to compare it to the random dataset in the previous section, although these datasets have. We search for the closest restaurants, food places and amenities to Times Square, Manhattan, NY, USA with the coordinates latitude 40.758896, and the longitude -73.985130, hard-coded into the code as:

```
point_xy geom;
geom.x= -739851300;
geom.y= 407588960;
```

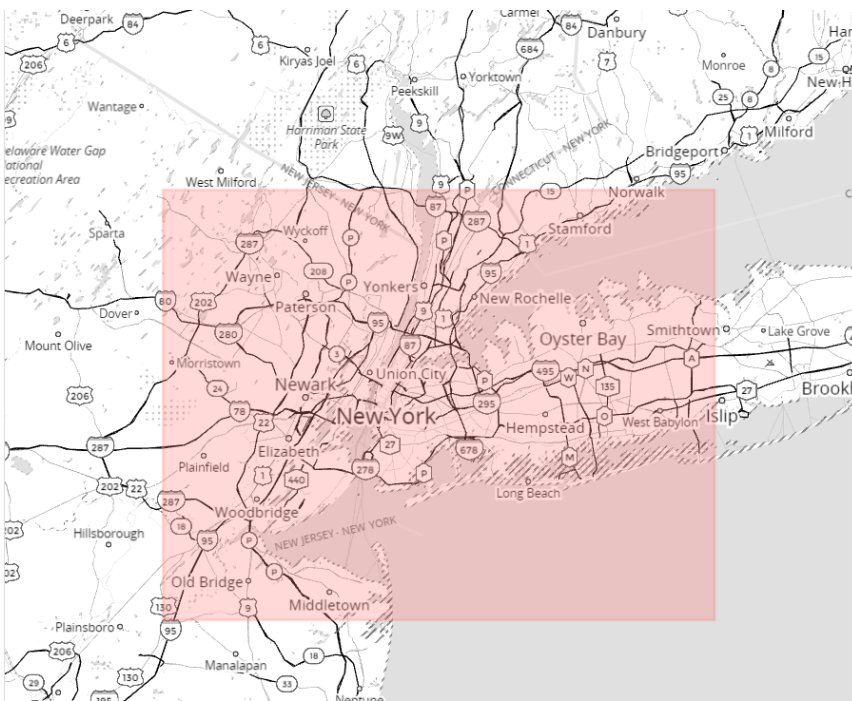


Figure 9.2: The New York area dataset visualization [Mapzen \(2017\)](#).

Then in the complete dataset extracted over New York, we search for the closest points to the Statue of Liberty National Monument, New York, USA with the coordinates latitude 40.689247, and the longitude is -74.044502, hard-coded into the code as:


```
point_xy geom;
    geom.x= -740445020;
    geom.y=  406892470;
```

Restaurants

This is the restaurants dataset and results.

```
+-----+
| SHOW CREATE TABLE restaurants |
+-----+
| restaurants | CREATE TABLE 'restaurants' ( |
| 'node_id' bigint(20) NOT NULL, |
| 'geom' geometry NOT NULL, |
| 'tag' text, |
| PRIMARY KEY ('node_id'), |
| SPATIAL KEY 'geom_SPATIAL' ('geom') |
| ) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+-----+
```

```
SELECT COUNT(*) FROM restaurants;
```

```
+-----+
| count(*) |
+-----+
|      3009 |
+-----+
```

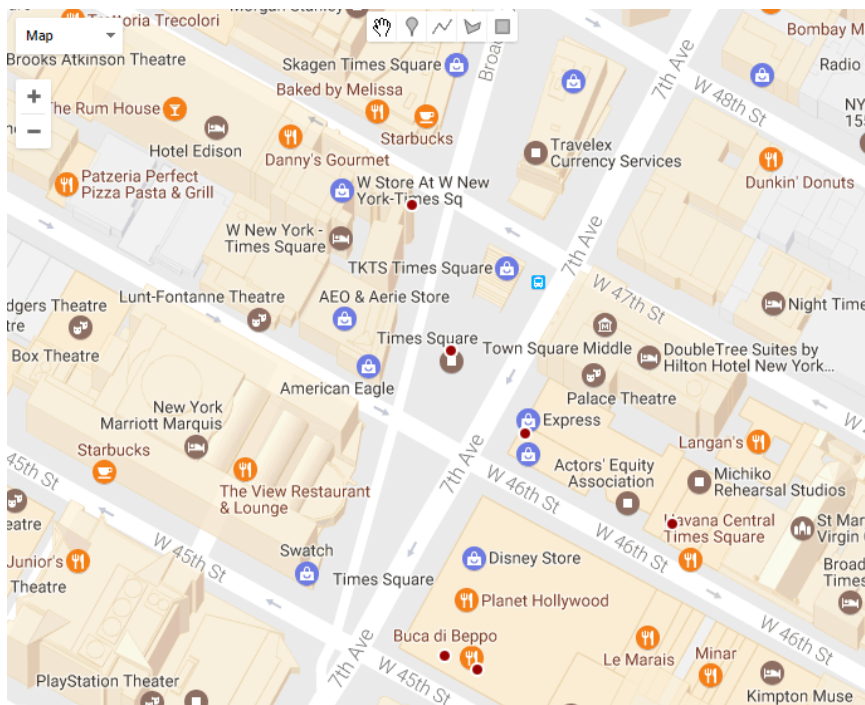


Figure 9.3: The 5 closest restaurants to Times Square.

Neighbors	Table Scan (seconds)	INNA (seconds)	% of Table Scan
1	0.008	0.002	25
301	0.009	0.005	56
601	0.010	0.007	70
901	0.011	0.009	82
1201	0.012	0.011	91
1501	0.013	0.013	100
1801	0.014	0.016	114
2101	0.015	0.017	113
2401	0.016	0.020	125
2701	0.017	0.022	129
3001	0.018	0.024	133

Table 9.2: Restaurant Query Results

The results are very close in this small dataset. The `mysqlslap` was run with 10 000 iterations for each of these queries to get the most accurate average. One can see that the INNA performs better when the number of desired neighbors is a small as possible.

Places to get food

This dataset encapsulates anywhere you could buy some food or drinks within the boundaries of the dataset.

```
+-----+
| SHOW CREATE TABLE food_places |
+-----+
| food_places | CREATE TABLE 'food_places' ( |
| 'node_id' bigint(20) NOT NULL, |
| 'geom' geometry NOT NULL, |
| 'tag' text, |
| PRIMARY KEY ('node_id'), |
| SPATIAL KEY 'geom_SPATIAL' ('geom' |
| ) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+-----+

SELECT COUNT(*) FROM food_places;

+-----+
| COUNT(*) |
+-----+
|      5593 |
+-----+
```

Neighbors	Table Scan (seconds)	INNA (seconds)	% of Table Scan
1	0.011	0.002	18
601	0.014	0.008	57
1201	0.016	0.012	75
1801	0.018	0.017	94
2401	0.020	0.022	109
3001	0.022	0.026	118
3601	0.024	0.029	120
4201	0.026	0.032	123
4801	0.029	0.037	128
5401	0.030	0.042	140

Table 9.3: Places to get food Query Results

All amenities in this area of New York

To get a somewhat lager dataset of points of interest, these are all the amenities contained within the boundaries of the dataset.

```
+-----+
| SHOW CREATE TABLE amenities |
+-----+
| amenities | CREATE TABLE 'amenities' ( |
| 'node_id' bigint(20) NOT NULL, |
| 'geom' geometry NOT NULL, |
| 'tag' text, |
| PRIMARY KEY ('node_id'), |
| SPATIAL KEY 'geom_SPATIAL' ('geom') |
|) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+-----+
```

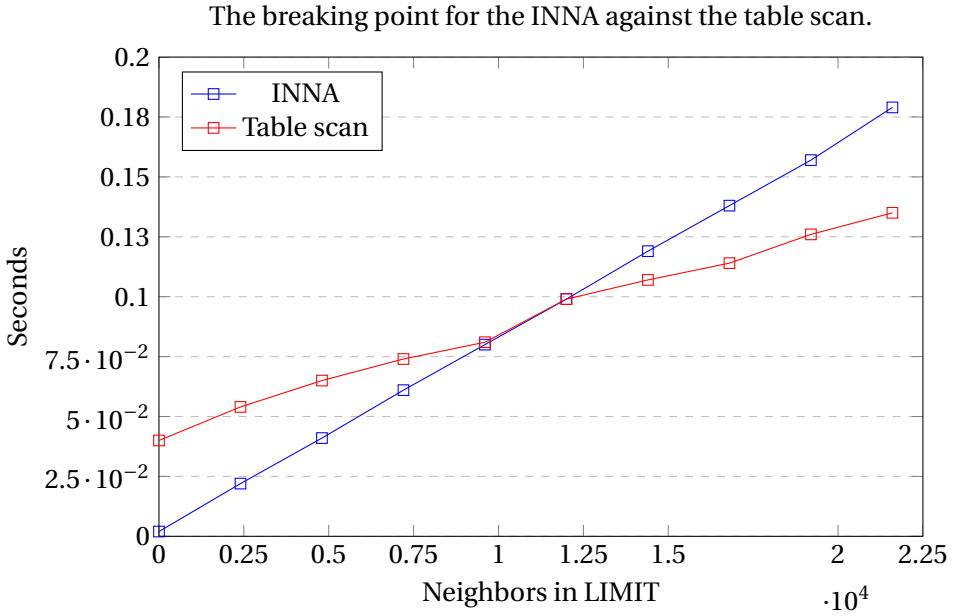
```
SELECT COUNT(*) FROM amenities;
```

```
+-----+
| COUNT(*) |
+-----+
|    23861 |
+-----+
```

The two data sets `food_places` and `amenities` further proves that the smaller the number of desired nearest neighbors the better the INNA performs against the table scan method of MySQL.

Neighbors	Table Scan (seconds)	INNA (seconds)	% of Table Scan
1	0.040	0.002	5
2401	0.054	0.022	40
4801	0.065	0.041	63
7201	0.074	0.061	82
9601	0.081	0.080	99
12001	0.099	0.099	100
14401	0.107	0.119	111
16801	0.114	0.138	121
19201	0.126	0.157	125
21601	0.135	0.179	133

Table 9.4: All amenities Query Results



All points contained within this dataset

The biggest and baddest table of Open Street Map data points, this is where we clearly can see where the INNA algorithm has its best performance.

```

+-----+
| SHOW CREATE TABLE new_york |
+-----+
| new_york | CREATE TABLE 'new_york' ( |
| 'geom' geometry NOT NULL, |
| SPATIAL KEY 'geom_SPATIAL' ('geom') |
| ) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+-----+

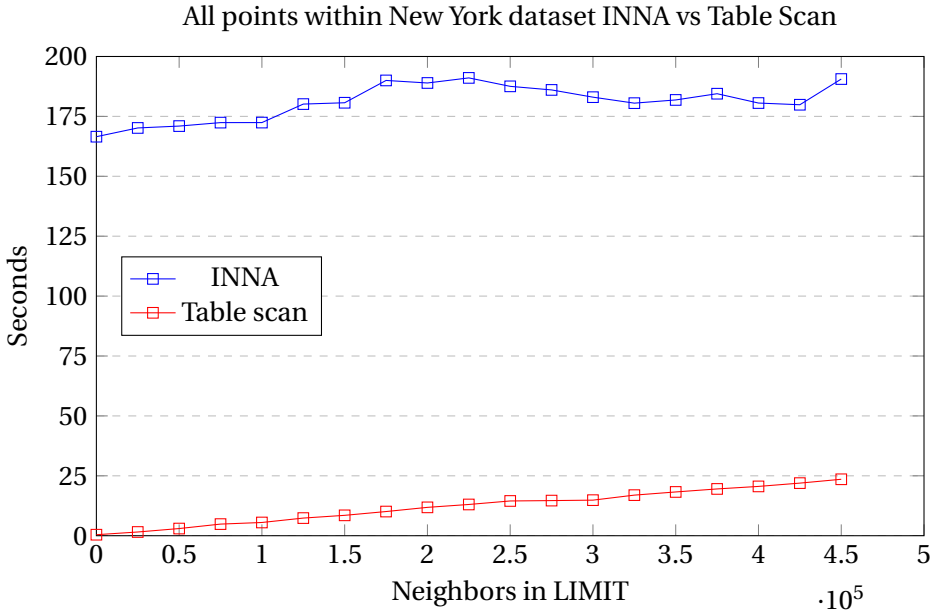
```

```
SELECT COUNT(*) FROM new_york;
```

```
+-----+
| COUNT(*) |
+-----+
| 11483747 |
+-----+
```

Neighbors	Table Scan	INNA (seconds)	% of Table Scan
1	166.527	0.421	0.25
25001	170.175	1.554	0.91
50001	170.957	2.933	1.71
75001	172.402	4.851	2.81
100001	172.418	5.523	3.20
125001	180.136	7.363	4.08
150001	180.691	8.508	4.70
175001	190.004	10.089	5.30
200001	188.941	11.836	6.26
225001	191.062	13.039	6.82
250001	187.539	14.504	7.73
275001	186.043	14.660	7.87
300001	183.023	14.847	8.11
325001	180.527	16.941	9.38
350001	181.863	18.265	10.04
375001	184.464	19.523	10.58
400001	180.570	20.578	11.39
425001	179.859	21.953	12.20
450001	190.586	23.558	12.36

Table 9.5: All points within New York dataset Query Results



The most interesting thing about this result from a table of nearly 12 million points is the profound improvement on returning just a couple of thousand nearest neighbors. At under a second you have the nearest neighbor and at 1.554 seconds you have the 25001 nearest neighbors. In common applications the need to return more than this is seldom. In contrast trying to get the single nearest neighbor with the table scan takes 2 minutes and 46 seconds, due to the fact that it needs to scan the whole table and then sort the data. This is also why the table scan increases minimally. Due to the fact that the table scan queries took so long, there was not enough time to get a perfect average with mysqlslap within a reasonable time frame. A straight line from the first value to the last would have been more accurate for the table scan result.

9.3 Summary

The implemented incremental nearest neighbor algorithm (INNA) has proven very efficient against small and large datasets. With the notion that the INNA performs better and better compared to the existing solution of table scan when the datasets gets considerably large (12 million points). In common use for this query, being under a thousand nearest neighbors, the algorithm performed to satisfaction.

Chapter 10

Conclusion and Future Work

This chapter is in two parts, the first contains the conclusion to the research done for the nearest neighbor problem (NNP) in spatial databases using R-Trees. The chapter describes the findings from the testing of the Incremental Nearest Neighbor Algorithm (INNA) and what kinds of improvements were made in comparison to existing solutions within MySQL. In the second section we look at what future work that can be done with this implementation.

10.1 Conclusions

Looking at the nearest neighbor search (NNS) it has been long standing topic in the field of pattern recognition and computational geometry. Going back all the way to Ibn al-Haytham with his observations of the eye. This thesis has looked at some of the different data structures commonly used with NNS and focused on the topic of NNS in R-Trees. The most common ones being k-d-b-Trees, B-Trees and R-Trees, while researching the R-Tree thoroughly it has proven to be a very efficient data structure to perform spatial queries within. Even though, as mentioned in section [5.2.2](#) there can be improvements done to the last stages of the R-Tree search structure.

There has been proposed several different algorithms for solving the NNP in the least amount of time, memory and preferably both. The most basic algorithm is the linear search and should be avoided for almost all purposes, being mentioned in this thesis only as a basis for understanding how a NNS can be implemented. Within space partitioning algorithms there seems to be a large supply, and this thesis has only described but a few of them. The ones described showed a lot of promise in regards to space and time complexity. The INNA is one of them, and is the algorithm chosen to build a prototype based on in MySQL. The other algorithms considered for implementation were the pivot based, and the most promising in regards to time complexity was the Tree Linear Approximating and Eliminating Search Algorithm (TLAESA). TLAESA falls short in regards to build complexity, making it undesirable in interactive systems.

The INNA was chosen due to its many properties overlapping with the functions desired in MySQL. The existing structures within MySQL fits perfectly with the INNA, since the objects are not stored in the R-tree, INNA can perform even better. Furthermore the INNA does not need to specify a desired number of neighbors beforehand and it is possible to extend this algorithm to perform a furthest neighbor search with simple modifications.

In chapter 9 several types of datasets were tested to see how well the INNA performed in regards to the data size and against existing solutions within MySQL. The algorithm was also tested on data from the real world, where clusters of points are more likely to occur. One of the prominent results was the ability to perform very fast even in large datasets.

The algorithm was able to find the nearest neighbor in less than half a second for the smallest and the biggest of our datasets. This indicates that the algorithm scales well. Further implementation and testing is necessary to prove this. In comparison to the existing solution the INNA was almost better in every way. It

is when the desired number of neighbors returned becomes very high that the INNA starts to perform less efficient.

10.2 Future work

In this section we first look at the work that needs to be done in order to fully implement the NNS with the INNA in MySQL. Second we give examples of features wanted as extension to the INNA and the NNS from the MySQL team.

10.2.1 MySQL Worklog

This thesis has been a part of an internal worklog in the MySQL system, and is not yet done. There are still the query specifications to design and how the user will ask InnoDB to perform the query. This implementation has been built on a stable release build in a version that is not up to date, there has been several changes to the internal landscape that is InnoDB from this thesis started to the end. The patch will need to be shuffled around to match the current version of the InnoDB code and especially the files containing the supporting R-Tree functions.

10.2.2 Other Spatial Data Types

There is a wish for being able to extend the NNS to include other spatial data types than just points. MySQL as of now supports 8 different spatial data types linestrings and polygons would be the natural next step to add support for. Being able to locate the nearest park (polygon) or the nearest tourist trail (linestring) would be welcome. One could simply do this by using the minimum bounding rectangle of that geometry and return the approximate distance. The current distance calculation implemented in this thesis does a calculation of the distance to the nearest line segment. This means that the effort needed to extend it to calculate distances to polygons or linestrings is minimal, if one can represent each of these geometries as several line segments.

10.2.3 Furthest Neighbor

As mentioned in section [7.2.2](#) there is an easy extension for doing a furthest neighbor search. Even with just small modifications this could be a great feature in MySQL. To implement this one would either need to make it such that one chooses a max or a min keyword, or perhaps a better choice would be to make the nearest function default and make the user specify if one should instead look for the furthest neighbor instead. How the query will look at the end will be up to the Optimizer team in MySQL.

Appendix A

Source Code

The source code for the incremental nearest neighbor algorithm in MySQL, described in chapter 8 and tested in chapter 9, can be found at the link below:

<https://github.com/jensevenb/nearest-neighbor>

Appendix B

Python point generators for MySQL

These are the python scripts made to generate different datasets of points.

B.1 Large random dataset

```
import random
def my_func():
    x = random.randint(-999999999, 999999999)
    y = random.randint(-999999999, 999999999)
    point = ''
    point +=str(x)
    point +=', '
    point +=str(y)
    return point

def write_file(data):
    data = my_func()
    file_name = r'D:\12mill_random.txt'
    for x in xrange(1, 17200):
        with open(file_name, 'a') as x_file:

            x_file.write('INSERT INTO 12mill_random (g) values '.format())
            for x in xrange(1, 700):
                data = my_func()
                with open(file_name, 'a') as x_file:
                    x_file.write('(POINT({})), '.format(data))

            with open(file_name, 'a') as x_file:
                data = my_func()
                x_file.write('(POINT({} '.format(data))
                x_file.write(');\n\n'.format(data))

def run():
    data = my_func()
```

```

write_file(data)

for _ in range(1):
    run()

```

B.2 Best case dataset

```

import random
def my_func():
    x = random.randint(0, 200)
    y = random.randint(0, 200)
    point = ''
    point +=str(x)
    point +=', '
    point +=str(y)
    return point

def write_file(data):
    data = my_func()
    xu = -2
    file_name = r'D:\best_case.txt'
    for x in xrange(1, 1250):
        xu+=1
        with open(file_name, 'a') as x_file:
            x_file.write('INSERT INTO best_case (g) values '.format(x))

            for x in xrange(1, 800):
                xu+=1
                data = my_func()
                with open(file_name, 'a') as x_file:
                    x_file.write('(POINT({} '.format(xu))
                    x_file.write(', {}), '.format(xu))

                with open(file_name, 'a') as x_file:
                    data = my_func()
                    x_file.write('(POINT({} '.format(xu))
                    x_file.write(', {}));\n\n'.format(xu))

def run():
    data = my_func()
    write_file(data)

for _ in range(1):
    run()

```

B.3 Worst case dataset

```

import random
def my_func():
    x = random.randint(0, 200)
    y = random.randint(0, 200)
    point = ''
    point +=str(x)
    point +=', '
    point +=str(y)
    return point

def write_file(data):
    data = my_func()
    xu = -2
    x1 = -2
    file_name = r'D:\worst_case.txt'
    for x in xrange(1, 1250):
        xu=x1*( (-1)**x1 )
        x1+=1
        with open(file_name, 'a') as x_file:
            x_file.write('INSERT INTO worst_case (g) values '.format())

            for x in xrange(1, 800):
                xu=x1*( (-1)**x1 )
                x1+=1
                data = my_func()
                with open(file_name, 'a') as x_file:
                    x_file.write('(POINT({} '.format(xu))
                    x_file.write(', {}), '.format(xu))

                with open(file_name, 'a') as x_file:
                    data = my_func()
                    x_file.write('(POINT({} '.format(xu))
                    x_file.write(', {}));\n\n'.format(xu))

def run():
    data = my_func()
    write_file(data)
for _ in range(1):
    run()

```


Appendix C

Open Street Map

C.1 Introduction

This appendix gives instructions on how to acquire spatial data from Open Street Map (OSM). The data imported is a segment of the entire world mapped in OSM, the segment being the New York City area.

C.1.1 Importing Open Street Map Data

The OSM file was downloaded in May of 2017 at this location:

https://mapzen.com/data/metro-extracts/metro/new-york_new-york/

Linux

It is possible to use a customized version of the OSM MySQL data import script found at this location:

<https://www.dropbox.com/s/l17vj3wf9y13tee/osmdb-scripts.tar.gz>

This will in addition create a geometry column from the “longitude,latitude” coordinate pairs. It will also create a new spatial index for InnoDB.

To load the New York City area data into MySQL use these commands:

```
mysql -e "create database nyosm"
bunzip2 new-york.osm.bz2
./bulkDB.pl new-york.osm nyosm
```

When imported the data should look like this.

```
+-----+
| SHOW CREATE TABLE nodes |
+-----+
|Create Table: CREATE TABLE 'nodes' ( |
| 'id' bigint(20) DEFAULT NULL, |
| 'geom' geometry NOT NULL, |
| 'user' varchar(50) DEFAULT NULL, |
| 'version' int(11) DEFAULT NULL, |
| 'timestamp' varchar(20) DEFAULT NULL, |
| 'uid' int(11) DEFAULT NULL, |
| 'changeset' int(11) DEFAULT NULL, |
| UNIQUE KEY 'i_nodeids' ('id'), |
| SPATIAL KEY 'i_geomidx' ('geom') |
|) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+-----+
```

Windows

To do the same form Windows the customized scripts did not work. Therefore the process is somewhat longer.

A regular OSM script can be found here: http://blog.openalfa.es/uploads/osm_createdb.zip

```
source d:/osm_createdb.sql;
```

The least painful way to do import the data is to download and configure osmo-

sis:

Detailed step by step instructions can be found here:

[wiki.openstreetmap.org/wiki/Osmosis/Quick_Install_\(Windows\)](http://wiki.openstreetmap.org/wiki/Osmosis/Quick_Install_(Windows))

After osmosis is configured:

```
mysql> CREATE DATABASE OSM;
```

```
mysql> CREATE USER 'newuser'@'localhost'
IDENTIFIED BY 'password';
```

```
osmosis --read-xml D:\newyork.osm --write-apidb dbType=mysql
host=localhost database=osm
user=newuser password=password
validateSchemaVersion=no
```

When imported the tables should look like this:

```
+-----+
| SHOW CREATE TABLE nodes |
+-----+
|Create Table: CREATE TABLE 'nodes' (
| 'node_id' bigint(20) NOT NULL,
| 'latitude' int(11) NOT NULL,
| 'longitude' int(11) NOT NULL,
| 'changeset_id' bigint(20) NOT NULL,
| 'visible' tinyint(1) NOT NULL,
| 'timestamp' timestamp NOT NULL DEFAULT
| CURRENT_TIMESTAMP ON UPDATE
| CURRENT_TIMESTAMP,
| 'tile' bigint(20) NOT NULL,
|
```

```
| 'version' bigint(20) NOT NULL, |
|) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+-----+

```

Set primary keys and create a geometry column from the “longitude,latitude” coordinate pairs.

```
mysql> ALTER TABLE nodes ADD geom Point;
mysql> UPDATE nodes SET geom = POINT(longitude, latitude);

mysql> ALTER TABLE nodes ADD PRIMARY KEY (node_id);
mysql> ALTER TABLE nodetags ADD PRIMARY KEY (node_id);

```

Use the ALTER TABLE to remove unnecessary columns

```
mysql> ALTER TABLE nodes
        DROP latitude, DROP longitude;

```

C.1.2 Altering tables

Now both Linux version and Windows version should look alike. To avoid redundant and time consuming sub-queries, we can make use of the full-text indexing feature in InnoDB. To utilize this we load all of the nodetags into a single column for each of the nodes like shown below.

```
mysql> ALTER TABLE nodes
ADD COLUMN tags TEXT,
ADD FULLTEXT index(tags);

mysql> UPDATE nodes
SET tag=(SELECT group_concat( concat(k, "=", v) SEPARATOR ',')

```



```
FROM node_tags
WHERE node_tags.node_id=nodes.node_id
GROUP BY nodes.node_id);
```


Bibliography

- Abernethy, M. (2010). Nearest neighbor and server-side library.
- Andoni, A. and Indyk, P. (2008). Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122.
- Arya, S., Mount, D. M., Netanyahu, N. S., Silverman, R., and Wu, A. Y. (1998). An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923.
- Becker, B., Franciosa, P. G., Gschwind, S., Ohler, T., Thiemt, G., and Widmayer, P. (1992). Enclosing many boxes by an optimal pair of boxes. In *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science, STACS '92*, pages 475–486, London, UK, UK. Springer-Verlag.
- Boundless (2016). Introduction to postgis. [Online; accessed April 23, 2017].
- Brownlee, J. (2016). K-nearest neighbors for machine learning.
- Choi, D.-W. and Chung, C.-W. (2015). Nearest neighborhood search in spatial databases. *2015 IEEE 31st International Conference on Data Engineering (ICDE)*, 00:699–710.
- Comer, D. (1979). Ubiquitous b-tree.

- Cover, T. and Hart, P. (2006). Nearest neighbor pattern classification. *IEEE Trans. Inf. Theor.*, 13(1):21–27.
- Documentation, M. (2017a). 4.5.9 mysqlslap — load emulation client.
- Documentation, M. (2017b). Query spatial data for nearest neighbor.
- Dohnal, V. (2004). *Indexing Structures for Searching in Metric Spaces*. PhD thesis, Masaryk University: Faculty of Informatics.
- Gaede, V. and Günther, O. (1998). Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231.
- García, Y. J., Lopez, M. A., and Leutenegger, S. T. (1998). On optimal node splitting for r-trees. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 334–344, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
- Geographic Information Systems, B. (2017). Solving the nearest neighbor problem in postgis.
- Giannella, C. (2009). New instability results for high-dimensional nearest neighbor search. *Inf. Process. Lett.*, 109(19):1109–1113.
- Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57.
- Hellerstein, J. M., Naughton, J. F., and Pfeffer, A. (1995). Generalized search trees for database systems. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 562–573. Morgan Kaufmann Publishers Inc.
- Hjaltason, G. R. and Samet, H. (1999). Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318.

- Jabbar, M. A., Deekshatulu, B. L., and Chandra, P. (2015). Classification of heart disease using K- nearest neighbor and genetic algorithm. *CoRR*, abs/1508.02061.
- Jacob Steeves (2015). Clustering knn encodings. [Online; accessed April 18, 2017].
- Jeremy, C. (2013). The physical structure of innodb index pages.
- Jiang, S., Pang, G., Wu, M., and Kuang, L. (2012). An improved k-nearest-neighbor algorithm for text categorization. *Expert Syst. Appl.*, 39(1):1503–1509.
- Jimmy, Y. (2014). Innodb spatial indexes in 5.7.4 lab release.
- Jin Liang, Koudas Nick, L. C. (2004). Nnh: Improving performance of nearest-neighbor searches using histograms.
- Johnson, Nick (2009). Damn cool algorithms: Spatial indexing with quadtrees and hilbert curves. [Online; accessed April 22, 2017].
- Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A. N., and Theodoridis, Y. (2005). *R-Trees: Theory and Applications*. Springer Publishing Company, Incorporated.
- Mapzen (2017). New york.
- Mendes, C. A. T., Gattass, M., and Lopes, H. (2014). Fgng: A fast multi-dimensional growing neural gas implementation. *Neurocomput.*, 128:328–340.
- Micó, L. and Oncina, J. (1998). Comparison of fast nearest neighbour classifiers for handwritten character recognition. *Pattern Recogn. Lett.*, 19(3-4):351–356.
- MySQL-Developer (2017). Wl6968: Innodb gis: R-tree index support.
- Oracle (2017). Spatial developer's guide.
- Pelillo, M. (2014). Alhazen and the nearest neighbor rule. *Pattern Recognition Letters*, 38:34–37.

PostGIS (2015). Postgis 1.3.6 manual. [Online; accessed April 22, 2017].

Press, C. U. (2008). Search structures for dictionaries.

Qef (2016). File:closest pair of points.svg.

Samet, H. (2005a). *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Samet, H. (2005b). *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Samet, H. (2005c). *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Samet, H. (2005d). *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Samet, H. (2005e). *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

School of Mathematics, U. o. M. (2007). Math32031: Coding theory • part 2: Hamming distance.

Shamos, M. I. and Hoey, D. (1975). Closest-point problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science, SFCS '75*, pages 151–162, Washington, DC, USA. IEEE Computer Society.

Sharifzadeh, M. and Shahabi, C. (2010). Vor-tree: R-trees with voronoi diagrams

- for efficient processing of spatial nearest neighbor queries. *Proc. VLDB Endow.*, 3(1-2):1231–1242.
- Sourceforge (2016). Nearest neighbor search with kd-tree demo.
- Tao, Y., Papadias, D., and Lian, X. (2004). Reverse knn search in arbitrary dimensionality. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30, VLDB '04*, pages 744–755. VLDB Endowment.
- Tavian, B. (2016). The approximating and eliminating search algorithm.
- Tmigler (2016). A complete binary tree that is not full.
- Tokoro, K., Yamaguchi, K., and Masuda, S. (2006). Improvements of tlaesa nearest neighbour search algorithm and extension to approximation search. In *Proceedings of the 29th Australasian Computer Science Conference - Volume 48, ACSC '06*, pages 77–83, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.
- Troyhildebr (2014). k-d-b-tree structure. [Online; accessed April 29, 2017].
- Weber, R., Schek, H.-J., and Blott, S. (1998). A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases, VLDB '98*, pages 194–205, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
- Wolfram, M. (2017a). Great circle.
- Wolfram, M. (2017b). Taxicab metric.
- xkcd (2017). Here to help.
- Yao, A. C.-C. (1982). On constructing minimum spanning trees in k-dimensional spaces and related problems*. *SIAM J. COMPUT.*, 11(4):780–787.