University of Cape Town

Faculty of Engineering and the Built Environment
School of Architecture, Planning and Geomatics

Investigating the time efficiency of indexing vector point data via the Point Quadtree

Candidate:
Moloko M. Mokubedi
MKBMOL006

Supervisor:
Dr. Moreblessings Shoko

A research proposal submitted in partial fulfilment of the requirements for the degree of Bachelor of Science in Geomatics (Geoinformatics) at the University of Cape Town

November 18, 2022

Authorship Declaration

I, Moloko Mahlogonolo Mokubedi, declare that:

1. This research report and the work presented in it are my own.
2. I know that plagiarism is wrong. Plagiarism is using another's work and pretending that it is one's own.
3. These calculations/reports/plans/source codes are my own work.
4. I have not allowed and will not allow anyone to copy my work with the intention of passing it off as his or her own work.

Signature: Moloko M Mokubedi

Date: 18-11-2022

ETHICS APPLICATION FORM

Please Note:
Any person planning to undertake research in the Faculty of Engineering and the Built Environment (EBE) at the University of Cape Town is required to complete this form **before** collecting or analysing data. The objective of submitting this application *prior* to embarking on research is to ensure that the highest ethical standards in research, conducted under the auspices of the EBE Faculty, are met. Please ensure that you have read, and understood the **EBE Ethics in Research Handbook** (available from the UCT EBE, Research Ethics website) prior to completing this application form: http://www.ebe.uct.ac.za/ebe/research/ethics1

APPLICANT'S DETAILS		
Name of principal researcher, student or external applicant		Moloko Mahlogonolo Mokubedi
Department		Architecture, Planning and Geomatics (APG)
Preferred email address of applicant:		mkbmol006@myuct.ac.za
If Student	Your Degree: e.g., MSc, PhD, etc.	BSc (Geomatics)
	Credit Value of Research: e.g., 60/120/180/360 etc.	40
	Name of Supervisor (if supervised):	Dr. Moreblessings Shoko
If this is a researchcontract, indicate the source of funding/sponsorship		N/A
Project Title		A comparative analysis of Quadtree and R-tree spatial data structures

I hereby undertake to carry out my research in such a way that:

- there is no apparent legal objection to the nature or the method of research; and
- the research will not compromise staff or students or the other responsibilities of the University;
- the stated objective will be achieved, and the findings will have a high degree of validity;
- limitations and alternative interpretations will be considered;
- the findings could be subject to peer review and publicly available; and
- I will comply with the conventions of copyright and avoid any practice that would constitute plagiarism.

APPLICATION BY	Full name	Signature	Date
Principal Researcher/ Student/External applicant	Moloko Mahlogonolo Mokubedi	Moloko Mokubedi	12/04/2022
SUPPORTED BY	Full name	Signature	Date
Supervisor (where applicable)	Dr. Moreblessings Shoko	M.Shoko M.shoko (Apr 13, 2022 10:00 GMT+2)	Apr 13, 2022

APPROVED BY	Full name	Signature	Date
HOD (or delegated nominee) Final authority for all applicants who have answered NO to all questions in Section 1; and for all Undergraduate research (Including Honours).	Patroba Odera	[illegible]	20 April 2022
Chair: Faculty EIR Committee For applicants other than undergraduate students who have answered YES to any of the questions in Section 1.			

Abstract

The manner in which data is stored in non-spatial data structures makes using the spatial proximity trait of spatial data near-impossible due to the way the data is sorted and indexed in non-spatial data structures. The main aim of this report is to investigate the efficiency of the Point Quadtree spatial data structure in inserting spatial data and retrieving spatial data from a digital mapping system in order to deliver critical data to decision-making stakeholders on time. The objectives are to determine the insertion and range querying time complexities of the Point Quadtree indexer and to determine if a Point Quadtree indexer delivers spatial data more time efficiently than a Dynamic Array indexer. To meet the objectives of this project, an experiment which involves partitioning a significantly large dataset made up of vector points representing Town Survey Marks in the City of Cape Town, into increasingly growing sizes and comparing the runtimes of performing insertion and range query retrievals executed by the Point Quadtree and the Dynamic Array, where the Dynamic Array will act as a reference. The Point Quadtree of the project yielded an insertion Big-Oh average-case complexity of $O(logn)$, while the Dynamic Array yielded an insertion Big-Oh average-case complexity of $O(1)$. In terms of retrieval, the Point Quadtree yielded a retrieval Big-Oh average-case complexity of $O(n^2)$, while the Dynamic Array yielded a retrieval Big-Oh average-case complexity of $O(n)$. The Point Quadtree data structure is less time efficient than the basic non-spatial Dynamic Array indexer. The project did not find that in its computations, the Dynamic Array maintained the geospatial properties of the vector data that was experimented upon. Thus the Dynamic Array is only found to be more efficient than the Point Quadtree to time efficiency.

Acknowledgements

Much appreciation goes to Dr Moreblessings Shoko for guiding this project and believing that it could be pulled off, despite the odds. Thank you to Daanyaal Ballim for helping with refreshing my knowledge of data structures. Thank you to Diane Steele for making accessing academic resources easier than it could have been. Thank you to Prof. Hussein Suleman, the work contributing to this report was largely inspired by your lectures on information retrieval techniques. Much appreciation to the South African Department of Agriculture, Land Reform, and Rural Development for funding this project. Thank you to Chidinma Ndu, who helped me to organize the project in a more academically presenTablemanner. To Dr Eric Thomas, you believed in me when even in the dark, your words carried me through the pain. Much appreciation for my parents and sister for persevering through this journey with me. Lastly, above all beings in the heavens and the earth, ALL CREDIT to Elohim Tzavaot in the heavens. You began this work and faithfully finished it, in Jesus Name, Amen.

Tehilah le'Elohim bashamayim, uveerets shalom veratsonn tov kelapey haanashiym.

•

Kganyo go Modimo magodimong, lefaseng go be khutso le mogau bathong ka moka.

•

Almajd lilah fi al'aeali , waealaa al'ard , walsalam walnawaya alhasanat nahw alnaasi.

•

Glory to God in the highest, and on earth, peace and goodwill towards all men.

Tableof Contents

List of Figures

List of Tables

Listings

1 CHAPTER ONE: INTRODUCTION

1.1 General Background

Since the advent of computer-based Geographical Information Systems (GIS) invented by Roger Tomlinson and IBM in the 1960s (Goodchild, 2018), geospatial software developers have been pushing the optimization limits of GIS operations. These include enhancing digital map visualizations, advancing the processing power of mainframe computers in relation to large-scale geospatial processes (Steenson, 2019) and finally, as will be the focus of this report, spatial data storage and retrieval in basic mapping systems.

Spatial data are unique and differentiated from other types of data by their ability to represent data elements in multi-dimensional spaces and most importantly their positional association with other spatial data elements distinguishes each spatial object and the processes that can be applied to it. These characteristics are considered during the design process of system architectures that deal with spatial storage and retrieval. The structural organization of spatial data in computer memory largely determines the efficiency with which this data can be accessed by any component of the computer system. The structural organization pattern referred to here is called a data structure (Goldman and Goldman, 2007). This report will be looking into a data structure that is specifically orientated towards spatial data management and how well it functions in a digital mapping system.

According to Kadochnikov, Shaparev, Tokarev, and Yakubailik (2019) mapping systems, geoinformation systems and geoportals are grouped into the same class of systems and are defined as the organisation of software tools, technology, information and computing support that focuses on the rapid development and delivery of applied geoinformation products and services. This entails that the tasks performed by such systems, involve the processing of large volumes of geospatial and related data and the provision of tools that will allow system users to interact with the system components via user interfaces (Kadochnikov et al., 2019). The uses of digital mapping systems are widespread, from business intelligence tracking to natural and urban disaster monitoring, weather and climate analysis and in most cases operational and performance management of urban activities.

Upon minor research, the spatial data structure found to be popularly used among geospatial data management systems is the Quadtree (Kothuri, Ravada, and Abugov, 2002). This spatial data structure, particularly the version of it that handles vector point data called the Point Quadtree, will be investigated and analysed to determine if a digital mapping system that has its spatial data insertion and retrieval tasks managed by it, will deliver critical data or alerts to decision-making

Figure 1: Geoinformation systems are ubiquitous and operational in most sects of society (Geography, 2021)

stakeholders in a timeous fashion.

The expectation held for a spatial data structure within this context will be to sort rapidly ingested spatial data from a streaming data source according to their geospatial positions in relation to each other, then apply an index to each data element, while placing them in a data store. This index serves as an address of where to find these data elements in the data store. The aim of the indexing pattern will be to reduce the number of spatial items to traverse through when any one or all of the data elements in the data store are required by a component of a geospatial system. Calls to retrieve and visualize data are frequently made by digital mapping systems, therefore, it only makes sense to design and develop them with this in mind so that they may be as efficient as possible given possible computational constraints on memory and processing cores (Guttman, 1984).

1.2 Statement of the Problem

Before spatial data can be queried from a data store, it has to be inserted into the components that make up that data structure. In non-spatial data structures, this is usually done by splitting the data into fields that are all encapsulated by a single record, thereby reducing the data to a single point, usually a row in a 2D Tableor an element in an array. This one-dimensional data element has no spatial relation to any of the other points/records. This becomes a problem for spatial data which are mainly queried based on their spatial properties and the spatial properties of the data records that are in proximity to the queried data (Samet, 1989). In such a case, there will be a need to retrieve the encapsulated spatial attributes in the fields of other records first, then only begin the query. This additional step will unnecessarily increase processing time, which is the antithesis of efficient data handling and may prove to be fatal in the context of a critical real-time mapping system such as one used in emergency services. Furthermore, the manner in which data is stored in non-spatial data structures makes using the spatial proximity trait of spatial data near-impossible due to the way the data is sorted and indexed in non-spatial data structures. In the case of spatial data, the sorting and indexing technique should be based on the spatial properties of the data. Such techniques are known as spatial indexing methods (Samet, 1989). The spatial indexing capabilities of the Point Quadtree data structure in supporting digital map systems will be the focus of this report. These capabilities will be assessed based on algorithm complexity metrics, namely the insertion and querying time complexity of a Point Quadtree implementation.

1.3 Purpose of the study

The main aim of this report, is to investigate the efficiency of the Point Quadtree spatial data structure in inserting spatial data and retrieving spatial data from a digital mapping system in order to deliver critical data to decision-making stakeholders on time.

1.4 Research Questions

1. What is the querying computational time complexity of Point Quadtree indexing?
2. What is the insertion computational time complexity of Point Quadtree indexing?
3. Does a Point Quadtree indexed data structure deliver spatial data more or less efficiently than non-spatial data structures?

1.5 Objectives of the study

- To determine the querying time complexity of Point Quadtree indexed spatial data.
- To determine the insertion time complexity of Point Quadtree indexed spatial data.

- To determine if a Point Quadtree indexer delivers spatial data more efficiently than a Dynamic Array indexer.

1.6 Scope and Limitations

In the world of data management (spatial and non-spatial), there is often confusion between databases and data structures and this leads to the erroneous interchangeable use of these phrases. It is important, that this confusion is cleared here for the case of this report. **?** defines a database as a collection of data that is stored, updated, and retrieved at any time from the same memory location. On the other hand, **?** defines a data structure as the manner in which the actual storing, updating and searching of the data, in the most computationally efficient way is implemented. Making this distinction will streamline the contents of this research report in a manner that does not cross into fields that are not aligned with the main issues addressed, which relate more specifically to data structures.

2 CHAPTER TWO: LITERATURE REVIEW

2.1 Introduction

In this chapter, scholarly work that has contributed to the knowledge base of spatial data, Quadtree data structures, data indexing and retrieval will be reviewed with the intention to gain an understanding of the history, developments, advances, challenges, opportunities, restrictions and future trends of the elements that make up point quadtrees.

2.2 Point-based spatial data

2.2.1 Spatial Data

To understand any attribute of an information system, it is imperative to understand the data that feeds into that system. For the purposes of this report, spatial data is of the most interest. van den Brink, Barnaghi, Tandy, Atemezing, Atkinson, Cochrane, Fathy, García Castro, Haller, and Harth (2019) refer to spatial data as any data that has a location component, further elaborating that this location component allows certain operations such as proximity and containment functions to execute within the spatial domain. Furthermore, the location component must have its reference to a place on Earth or within some other defined space such as a built environment structure. The distinctions between spatial and so-called non-spatial data are that spatial data are generally multi-dimensional and auto-correlated whilst non-spatial data are generally one-dimensional and independent of each other Fazal (2008, p. 100). These distinctions are vital as they are guiding points on how to handle the processing and storage issues associated with each class of data respectively.

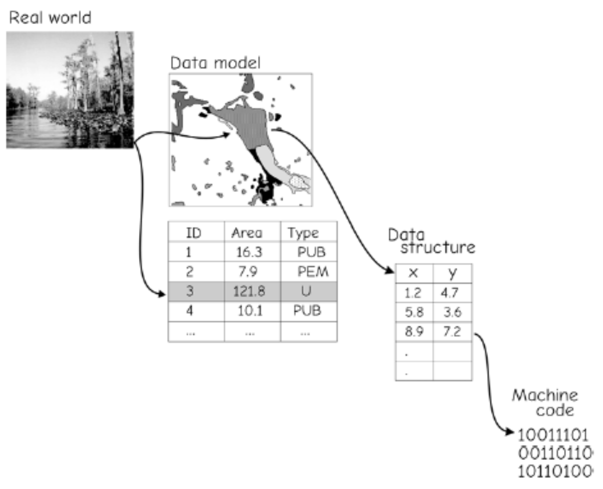
Due to spatial data's multidimensional properties, this data has the flexibility of being represented in a variety of ways, where each form of representation affects the specific tasks that could be performed on the data Samet (1989, p. 43). Sorting is one of the more common and important tasks that are considered when there is a need to process and then ultimately represent the data via visualization information systems.

A major concern with spatial data, unlike with non-spatial data is that it is not obvious which algorithms should be used to sort it. Therefore the distinctions stated above justify a need to apply a separate and more spatial-orientated data-handling approach for understanding for indexing spatial data.

2.2.2 Spatial Data Models

In the world of data, conceptual paradigms are expressed through data models. Forlizzi, Güting, Nardelli, and Schneider (2000) state that a data model is given as a collection of data types and operations which can be plugged into computer data stores to obtain a complete representation of the data and a query language. In this report, the focus will be on spatial data models. Spatial data models are defined as spatial objects in a spatial database and the relationships between each of them, additionally, spatial data models are specialized to provide a means of representing and manipulating spatially referenced information (Bolstad, 2016, p. 30).

To understand the relevance of spatial data models, the concept of computational abstraction must be explored. Computational abstraction is a series of processes that turns measurable real-world phenomena into machine versions of the same phenomena Wise (2016). Some texts go as far as to refer to computational abstraction as 'data modelling' (Worboys and Duckham, 2004) as they are so critical to the formation of data models. As seen in Figure 2, the computational abstraction processes involve turning real-world entities with spatial data into data models, then turning those models into data structures such as the Point Quadtree (which is the focus of this report) then turning those into machine-readable code that can be accessed and manipulated by the machine in order to execute operations on the data. The real-world entities that move along this abstraction pipeline, are identified as objects Fazal (2008, p. 148) and in the data model stage, these objects are described in terms of their data type, geometric elements, attributes, relations, and qualities.



ID	Area	Type
1	16.3	PUB
2	7.9	PEM
3	121.8	U
4	10.1	PUB
...	...	...

x	y
1.2	4.7
5.8	3.6
8.9	7.2
.	
.	

```
10011101
00110110
10110100
```

Figure 2: The process of abstracting real-world items into machine-readable and executablecode. (Bolstad, 2016, p. 30)

Brisaboa, Bernardo, Gutiérrez, Luaces, and Paramá (2017) and Bolstad (2016) point out that over the years, digital spatial models have been split into 2 camps: vector data model (also referred to as spatial objects) and raster data models (also referred to as spatial fields). Vector data models discretely represent objects as points, lines, and polygons, whilst raster data models represent objects as a set of grid cells for a given region of interest (Bolstad, 2016, p. 30). The vector data model is of most interest to this project as it gives context to the spatial data that Point Quadtrees perform on and is covered in more detail in section 2.2.3.

In general, when geoinformation system developers make decisions on whether to use raster or vector data models, a number of factors are taken into account, these factors are stipulated in Table1 what is taken into consideration is the frequency of operations performed (Bolstad, 2016, p. 30). The best data model for a given application depends on the most common operations, the experiences and views of the GIS users, the form of available data, and the influence of the data model on data quality.

Factor	Vector Model	Raster Model
data structure	usually complex	usually simple
storage requirements	smaller for most data sets	larger for most data sets without compression
coordinate conversion	simple	may be slow due to data volumes, and require resampling
analysis	preferred for network analyses, many other spatial operations more complex	easy for continuous data, simple for many layer combinations
spatial precision	limited only by positional measurements	floor set by cell size (spatial resolution)
accessibility	often complex	easy to modify or program, due to simple data structure
display and output	maplike, with continuous curves, poor for images	good for images, but discrete features may show “stairstep” edges

Table 1: Factors to take into account when choosing between raster and vector data models (Bolstad, 2016, p. 59)

Other spatial data models exist but are less commonly used as they are too specialized for specific use-cases such as triangulated irregular networks (TIN) which are specialized for terrain analysis, object data models which are specialized for business intelligence (BI) use cases and 3D data models which are specialized for built environment information systems such as building information modelling (BIM)(Bolstad, 2016, p. 63).

2.2.3 Vector Data Model

As mentioned earlier, the vector data model aims to present spatial features via 3 basic graphical elements: points, lines and polygons in a user-interface-based information system such as a GIS. The Vector data model has an edge over previously mentioned data models because they have their downfalls in comparison to the vector data model particularly when it comes to transferring, manipulating, storing and visualizing the graphical aspects of the same data. For instance, according to (Fazal, 2008, p. 112), Object data models are not easily abstracted directly into spatial databases or spatial data files because some real-world entities may be comprised of several other more complex objects. (Fazal, 2008, p. 112) also points out that although the raster model is appealing due to the simplicity of organizing data into grid-like patterns, the model often finds itself failing to deliver high levels of detail.

The vector method of representing geographic features is based on the concept that these features can be identified as discrete entities or objects (Goodchild, 1992) that can be uniquely located using two or three numbers in a coordinate system, such as in Universal Transverse Mercator (UTM) coordinate system. This means that the locational element of vector data is intrinsically simple to handle as it can simply be denoted by a pair or triplet of coordinates in the data store (Wise, 2016). The slightly more complex part of handling the data is in telling apart the data elements from each other. This is the main role of indexing data structures and attributes data. Handling digital forms of spatial data using the vector model in an information system involves using vector files as well. How vector data files are structured to handle the indexing of spatial will be discussed in section 2.2.6. In terms of computational efficiency vector data files have an upside when it comes to storage. Worboys and Duckham (2004, p. 17) points out that this is because vector files only carry data of interest. On the flip side the computing times of vector data-based operations such as overlays, tend to be excessive and early error-prone, and expensive.

2.2.4 Vector Point data

Spatial point data as a sub-variant of vector data is the main data type that will be focused on in the investigation that leads to this report. From here on vector-based points will be referred to simply as points. According to Fazal (2008, p. 112) a point is "the simplest graphical representation of an object that may be indicated on maps or displayed on screens by using symbols." Points may have attribute data attached to them which basically describe an object's spatial and non-spatial features (Fazal, 2008, p. 148). Samet (1989) introduces the concept of the *Multidimensional point.* To understand the multidimensional point, the data store in which it is stored needs to be understood. Multidimensional points are stored in databases termed *files* which are a collection of records where each record is an instance of a point. The most critical part of a file is that each record contains attributes or keys that correspond with it (Samet, 1989, p. 43). Samet (1989) further allows this view to be abstracted as such: "a file contains N records with k keys apiece". This will allow us to talk about the k-dimensions of each point record N. The multidimensional point allows systems to operate on the point in multiple ways without having to rely on additional sub-system components, for instance as is the aim of this report, executing a query for a particular multidimensional point record in a file may require finding a key of that record without having the system having to call a sub-system function that would run any calculation.

Previous literature has supported using $k = 2$ dimensional cases to demonstrate the effectiveness of using multidimensional points in spatial information processes and systems, where the X and Y coordinates of the point will be the values of these keys respectively (Xiao, 2015). This is because it is relatively easy to generalize the performance of the system to an arbitrary number of dimensions once the spatial dimensions of the multidimensional point have been helped to determine a pattern. (Samet, 1989).

2.2.5 Sources of Vector Point Data

Before any file of spatial point records can be used by any system, the point records have to be collected from the real-world objects that supply the data they represent. The process of collecting this data is called data capturing (Worboys and Duckham, 2004, p. 19). In the modern era, data capturing is usually an automated or semi-automated process where the digital data feeds directly to an information system, undergoes processing, and thereafter gets stored within a data store or database (Worboys and Duckham, 2004, p. 19). Since the interest of this investigative report is in vector point data, the two primary sources of vector data will be explored, these are ground surveys and Global Navigation Satellite Systems (GNSS).

Ground surveys are the traditional form of capturing spatial data for mapping purposes. The basic principle of capturing point data via a ground survey is that the 3-dimensional location of a point on the Earth's surface is deduced by measuring angles and distances from or to a point that has its 3-dimensional points known (Fazal, 2008). Core to the ground surveying data capturing process is the coordinate system in which the point is defined, as this will affect all subsequent processes conducted on the point(s) recorded. The coordinate system is usually captured as metadata during the data-capturing process and is configured into the file according to the geographic data format that the spatial data file follows (see section 2.2.6). The advantage of capturing point data via ground survey measurements is that the geospatial location of each recorded point is relative to the other recorded points. This relativity is referred to as *spatial auto-correlation* (Fazal, 2008, p. 122). Spatial auto-correlation is formally defined by Getis (2010) as *"the relationship between nearby spatial units, as seen on maps, where each unit is coded with a realization of a single variable."* and more famously described by Tobler (1970) as the idea that *"Everything is related to everything else, but near things are more related than distant things"*. In the context of multidimensional point data operations in a geospatial information system, it is imperative that there is an awareness of the level of spatial auto-correlation that the data maintains as this affects any assumption made on the spatial stationarity and spatial heterogeneity of the mapped-out put of the system (Getis, 2010, p. 258). Spatial stationarity and spatial heterogeneity are the spatial analysis concepts that describe the distribution parameters of the point records such as the spatial mean and spatial variance (Getis, 2010, p. 258). These are important as the effect of the spatial distribution of the data captured can trickle down to the data store operations level of the geospatial system and affect computation factors such as operating system thrashing and data decomposition recursion (see 2.4). Contemporary surveying tools are sensor-based and include equipment such as total stations which can measure angles and distances to an accuracy of 1 millimetre, calculate point positions according to a pre-set coordinate system and directly add point records to a file that can be transferred electronically (Fazal, 2008, p. 122). Some sophisticated total stations have the ability to allow the user to add and link attribute data (keys) to vector points recorded in the field.

The second primary way of capturing vector point data is via the GNSS. GNSSs are networks of space-borne satellites, monitoring control stations, and user-based signal receivers primarily used to capture spatial data of the Earth (Kumar and Moore, 2002). The 3 GNSS segments described above are constituents of the US-based and managed Global Positioning System (GPS) that has gained a *de facto* GNSS services provider reputation globally (Bolstad, 2016, p. 204). This is especially due to the development of the differential GPS procedure, the removal of selective availability, and the availability of low-cost, low-powered receivers, which means that members of the general public can access spatial point data at accuracies of 10 m and less (Kumar and Moore, 2002). The GPS data-capturing process works on the geometric principle of trilateration, where a GPS receiver calculates its three-dimensional coordinates (where the coordinates are usually based on the widely accepted WGS84 coordinate system that denotes position via latitude, longitude, and altitude). The user segment of GPS is of most interest to GISs and geospatial information systems alike. This is because these receivers have become valuable sources of big data for numerous Information and Communications Technology (ICT) tools such as transportation systems, operational dashboards, weather monitoring systems and many more (Croce, Musolino, Rindone, and Vitetta, 2019). The problem connected to big data is that they are scattered and indistinct, therefore, a great effort is necessary to filter, integrate and convert them into easier forms of data for machines to process (Croce et al., 2019). GPS receivers can be attached to different types of devices which makes it a powerful and near-ubiquitous vector point data-capturing tool due to it being dynamic, near real-time, and relatively more accessible than surveying, it is a more appealing data-capturing method to feed into time-based mapping systems.

Other vector data capture tools that deserve honourable mention and are considered secondary vector data capture techniques involve digitizing vector objects *legacy data* sources, which are basically already produced paper maps (Worboys and Duckham, 2004, p. 19). Popular secondary vector data capture methods are manual digitizing (by hand), vectorization, scanning and photogrammetry (Fazal, 2008, p. 122). Most of these techniques are unreliable for the high-quality data capturing of real-time systems as they are either difficult and complex or time-consuming and laborious leading to exorbitant costs for the data capturing project (Fazal, 2008, p. 122).

2.2.6 Geographic Data formats

With various ways to digitally capture vector point data, there comes the challenge of dealing with the various ways the file that carries the data is encoded and structured within it. For instance point data captured on a total station may be encoded in Unicode and stored in a raw field data format file such as LandXML or may be encoded in ASCII characters and stored in a pre-processed XYZ coordinate data format file such as the Leica GEO Serial Interface (GSI) (Costa, Gaignon, and Bianconi, 2020) where the encoding scheme and data format are dictated by the total station type, total station brand and the user. The same goes for GPS data where the collected points could be encoded in well-known binary (WKB) units and stored in the GPS Exchange Format (GPX) (Ma and Wang, 2014). Why would geospatial information system developers and stakeholders be concerned about the encoding and file format of the data that passes through their systems? System compatibility and cohesion are the reason why (Longley, Goodchild, Maguire, and Rhind, 2005). For an information system to maintain the correctness of its output, its components need to handle each input file in a cohesive and synchronized manner. This implies that each system component expects the data to be structured in a particular way before the processing of it commences. For instance, when performing a delete operation from a tabular file format such as a comma-separated values (CSV) file, the system function that is responsible for executing the task is expecting to find all *k* keys of that record in the same row, separated exclusively by commas as the data format entails. An exception to this expectation may cause the system to fail in fully accomplishing its use case. In the geospatial data context, this also means that the reference system in which the points may have been collected may need to be transformed, overlaid, merged, corrected or compared to the map projection and/or datum as the one the system will output the final result in before the objective processing begins (Longley et al., 2005).

There are many different geospatial data formats that have been developed and used for the exchange of geospatial data across different geospatial information systems or components of the same geospatial information system. There is no single universal format because, as stated before, different systems and sub-systems handle spatial data differently (Chen, Sabri, Rajabifard, and Agunbiade, 2018). Although it is not feasible to design a universal geospatial data format, there are more than 25 international organizations that are involved in the standardization of geographic data formats and exchanges. The International Standards Organization (ISO) has rigorously worked on standardizing geographic and spatial data communication at a global level via the technical committees TC 211 and 287 which coordinates extensively with the Open Geospatial Consortium (OGC) to remain up to date with advances in geospatial technologies and maintain the relevance of the standards.(van den Brink et al., 2019).

2.2.7 GeoJSON Data format

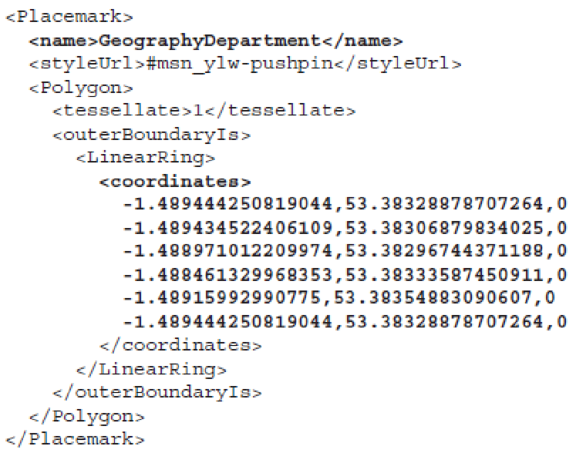
GeoJSON is a format for encoding of geospatial data that is based on JavaScript Object Notation (JSON). Although GeoJSON is a relatively new geographic data format, it is not an entirely novel invention as most of its features are derived from pre-existing open geographic information system standards set by the OGC (Butler, Daly, Doyle, Gillies, Hagen, Schaub, and Wilde, 2016). What necessarily specializes GeoJSON from other standardized spatial]data formats is that it is streamlined to fit into modern web system mechanisms.

A GeoJSON file at its core is a nested key-value data structure with 3 essential key-value dictionaries that dictate the data responsible for the representation of a single or multiple spatial object abstractions (Butler et al., 2016). These keys are:

- The Geometry dictionary which represents points, curves, and surfaces in coordinate space
- A Features dictionary which represents a single spatially bounded entity
- A FeatureCollection which is a list of Feature dictionaries

The benefits of the GeoJSON file is that it is structured exactly as a JSON file, thus it is inter-operable with web-based and non-web-based sub-systems that are programmed to deal with data in JSON format. It is for this reason that GeoJSON has taken over as a primary spatial geometry format used in the geospatial and Web communities ahead of older and more robust spatial data formats such as the Geography Markup Language (GML), Keyhole Markup Language (KML) and GeoSPARQL (van den Brink et al., 2019).

Features in GeoJSON files are essentially the records of the vector objects that have been collected during data capturing. Each feature contains a geometry component. A collection of features in the same GeoJSON file is represented as a FeatureCollection which is an array of features (Butler et al., 2016). This structure is very similar to that of a KML as seen in Figure 3 where the KML Placemark tag performs the function of the GeoJSON FeatureCollection key, the KML Polygon tag performs the function of the Geometry key.



```
<Placemark>
  <name>GeographyDepartment</name>
  <styleUrl>#msn_ylw-pushpin</styleUrl>
  <Polygon>
    <tessellate>1</tessellate>
    <outerBoundaryIs>
      <LinearRing>
        <coordinates>
          -1.489444250819044,53.38328878707264,0
          -1.489434522406109,53.38306879834025,0
          -1.488971012209974,53.38296744371188,0
          -1.488461329968353,53.38333587450911,0
          -1.48915992990775,53.38354883090607,0
          -1.489444250819044,53.38328878707264,0
        </coordinates>
      </LinearRing>
    </outerBoundaryIs>
  </Polygon>
</Placemark>
```

Figure 3: Keyhole Markup Language (KML) data format (Wise, 2016, p. 19)

The geometry component of a record GeoJSON can be explicitly configured according to one of the seven concrete geometry types defined in the OpenGIS Simple Features Implementation Specification for SQL [SFSQL] which are:

1. 0-dimensional Point
2. MultiPoint (*Multidimentional Point*)
3. 1-dimensional curve LineString
4. MultiLineString
5. 2-dimensional surface Polygon
6. Multi-Polygon

7. heterogeneous GeometryCollection

Once the type of geometry has been set, the coordinates key can be given a value or set of coordinate values as a list construct, for the record in question. The GeoJSON Code Listing 1 illustrates this file structural pattern. If a GeoJSON record feature has an empty array or no value in its coordinates key, the GeoJSON processors MAY interpret the feature as a null object (Butler et al., 2016). This is critical for geospatial system developers to be aware of since the position of a feature is the fundamental geometry construct of the feature, where the position is an array of numbers where the first two elements are the WGS84 longitude and latitude coordinates, precisely in that order and using decimal numbers(Wise, 2016, p. 20). It is possible to extend GeoJSON files so that they may hold attributes and any additional indexing data, this ability is credited to GeoJSON's inheritance of JSON features (Butler et al., 2016).

Listing 1: GeoJSON Example (Butler et al. 2016)

```
{
    "type": "FeatureCollection",
    "features": [{
        "type": "Feature",
        "geometry": {
            "type": "Point",
            "coordinates": [102.0, 0.5]
            },
        "properties": {
            "prop0": "value0"
        }
    }
    }]
}
```

2.3 Point Data Access in Computer systems

The elements involved in any data-accessing process on a computing device are mainly the central processing unit (CPU) which is made up of caches and registers, the memory unit, and the long-term data storage unit which is made up of hard disk or solid-state drives (Wise, 2016, p. 213).

These 3 components are linked to each other in a computing architecture schema that facilitates the flow of data via an *input* → *processing* → *output* data accessing process. Figure 4 on the next page illustrates this schema diagrammatically.

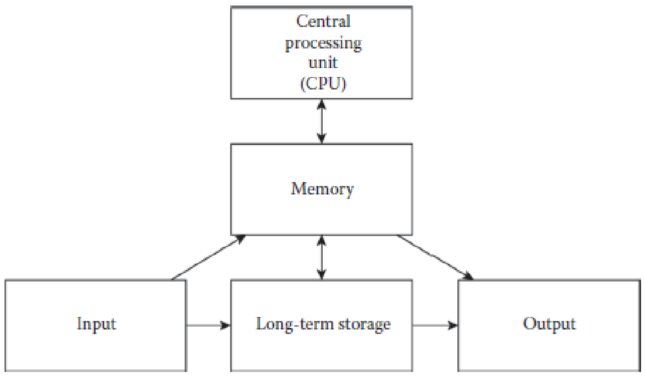


Figure 4: Schematic diagram showing the main elements of a computer. (Wise, 2016, p. 102)

The majority of the data processing step takes place in the CPU where the instructions of an algorithm are executed via basic control flow logic, addition & subtraction of numbers, and the signalling of input/output to the various sub-components of the computer (Patterson and Hennessy, 2016, p. 19). The memory unit is the component where the programs that carry the algorithms and data are kept when they are required by the user to be run (Patterson and Hennessy, 2016, p. 19).

As shown in Figure 4, the flow of data follows that the data the CPU needs is passed from long-term storage (which is usually on a magnetic disk) to memory and from there to the CPU, with the results of the computation being passed back to memory then, to the long-term storage as to be presented as output (Wise, 2016, p. 213). One can pick out that the efficiency of the whole computing process can roughly be abstracted to how efficiently data is moved from the elements, with the main objective of getting the data from memory to the CPU and back as fast as possible.

Due to the respective primary roles of the CPU and memory stipulated by Patterson and Hennessy (2016), it is important to realize that the CPU is limited in terms of internal data holding capacity during its computations and that it relies on memory to pass that data to and from the CPU. It is equally important to note that there is also a limit on memory, as the data in memory is completely discarded when the computer is turned off. This is where the long-term data storage unit comes in to fulfil the role of being a permanent data store in cases where data that was in memory can not be accessed anymore (Wise, 2016, p. 213). With the advent of cloud-based data stores and an increasing reliance on them, the source of the programs and data sent to memory and the CPU may come from a different computer's long-term storage across a network such as an internet (Wise, 2016, p. 213).

There exists a hierarchy of these data access elements that organizes them from the fasted-to-access, to the slowest-to-access called the *memory hierarchy* (Ajwani and Meyerhenke, 2010). Figure 5 illustrates the memory hierarchy and shows that the fasted data transfer occurs between the memory element and the CPU elements (registers and caches). Figure 5 also shows that the access of data between long-term disk storage and memory is slower than the former, this is because of the controlled way the data is stored in memory, a factor that is a part of the program (algorithm and data) that is being accessed and partly because disks are typically mechanical device, that have physics-based limitations (Wise, 2016, p. 213). Another impact on data access and transfer speed that is not included in the memory hierarchy is the transfer/access of data across the network drives. This is typically slower because of the number of OSI/ISO network layers the data has to go through to get from one computer to the next and network traffic (Wise, 2016, p. 213).

Due to the increasing number of spatial datasets that are too large for memory, the constraints of the computing process schema, and the slowness of components presented by the memory hierarchy, a considerable amount of thought and research have gone into developing techniques of making retrieving data in computer systems as efficient as possible while taking these factors into account. Three classes of techniques have been discovered thus far and include (Wise, 2016, p. 106):

1. Creating indexes that allow the correct data to be found on the disk or memory quickly.
2. Storing data on the disk in a way that means that it can be transferred to memory as quickly as possible.
3. Keeping a local copy of the data in case it is needed again, this is a process known as *caching*.

The first technique proposed by Wise (2016) has been introduced in this project report earlier on in the Introduction and will be the focus of the investigation as mentioned before. Using the spatial file model of storing and accessing spatial data

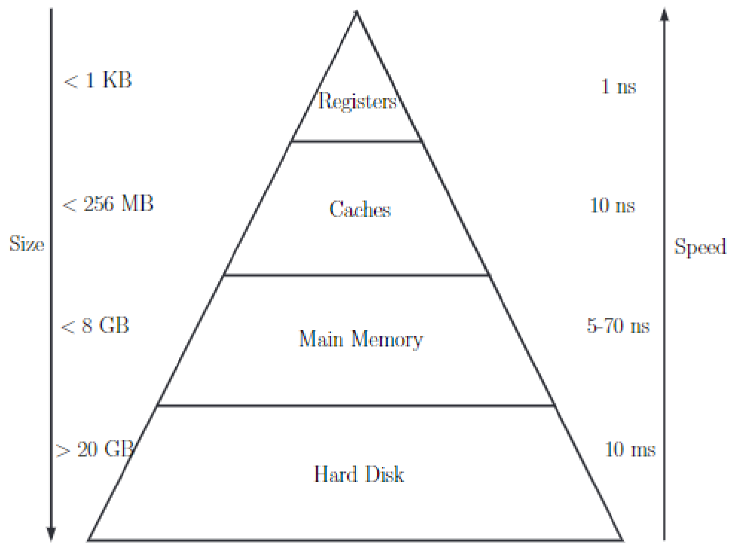


Figure 5: Memory hierarchy in modern computer architecture. (Ajwani and Meyerhenke, 2010, p. 102)

that Samet (1989) follows and knowing that files are the programs and data stored and transferred along elements of the memory hierarchy, it becomes simpler to introduce the concept of explicitly parametrizing collected spatial point data into a collection of records, where each record has many fields that hold the desired spatial aspect of the record (such as the geometry in a GeoJSON record) and using those to create indexes that allow for simple, yet **time efficient** retrieval of the data (Samet, 1995).

2.3.1 Point Data Retrieval

Now that the manner in which spatial data and data, in general, is stored and accessed in the computer system has been covered, the discussion of the report will narrow in on the major motivation of this investigation, point data retrieval/querying. Traditionally, spatial data retrieval operations were designed and developed as part of more complex GIS-based analysis operations (Fazal, 2008, p. 2), however with spatial computations becoming more and more ubiquitous due to the web, it is expected that search engines will begin integrating spatial search functionalities in their systems (van den Brink et al., 2019). This shows that it is worthwhile to look into spatial querying operations and how they have evolved over the years. The terms query, search, and retrieval may be used interchangeably.

Knuth (1973) lists the 3 most common types of queries of point spatial data performed by spatial programs:

1. **Record queries**, which determine if a given data point record is in the data store and, if so, yields the address in memory corresponding to the record in which it is stored.
2. **Range queries**, also known as region searches, retrieve a set of data point records whose specific key/fields have specific values or values within given ranges.
3. **Boolean queries**, which consists of combinations of types 1 and 2 with the Boolean operations AND, OR, NOT, and so on

The focus of the project will be on investigating the indexing of point data for range querying operations more specifically. Wei, Hsu, Peng, and Lee (2014) provides a clear and comprehensive explanation of how the range query algorithm should unfold irrespective of the data structure that it organizes the data in for indexing or the algorithm steps involved:

"Given a set of data points P and a spatial range R, a range query can be formulated as "searching the data points in P that are located in the spatial range R, where each data point record has spatially referenced fields (such as longitude and latitude values)"

The spatial range should be represented by a geometrically defined spatial object that can fully surround the spatial objects of interest. This may be a circularly or quadratically-shaped range (Wei et al., 2014). Figure 6 illustrates a use-case of the application of the range query in a digital mapping system interface, here 15 restaurants are marked by grey points, and a rectangular query range R is shown in red. The range query computation, as described by Wei et al. (2014) will search for and return single field values (usually the name field) of all the restaurants that have their latitudes and longitudes located within the bounds of the range R. That is all points that have latitude values between the northernmost latitude and the southernmost latitude values of R AND the easternmost longitude and the northernmost longitude values of R as well. The output of the range query shown in Figure 6 will be $\{p1, p2, p3, p4, p5\}$

Nievergelt, Hinterberger, and Sevcik (1984) grouped searching techniques into two broad categories, the first category includes techniques that organize the data to be stored, and the second category includes techniques that re-organize the embedding space from which the data are presented so that it fits the inherent data pattern. The embedding space on which the data in a file is represented matters when considering the data structure involved in the querying operation, this is particularly important

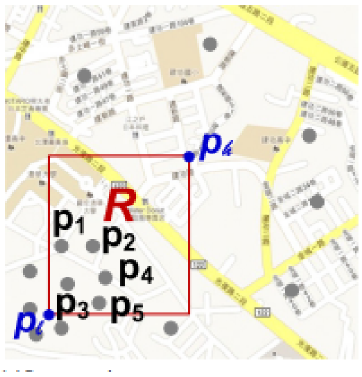


Figure 6: Illustration of Range query of point data in a digital mapping interface (Wei et al., 2014)

if a query involves the space occupied by the data. This is because the querying process becomes more complex than just organizing the data in a memory-efficient manner but also in a spatially efficient manner taking into account factors such as the proximity of other records and inevitably spatial auto-correlation (which was detailed in Chapter 2.2.5). If the spatial data was collected in a manner in which the spatial properties have been preserved this complexity of organizing the data will not impede the data accessing and querying operations as too badly, however, this is not always guaranteed and in such cases it will be imperative that there is a mechanism that will allow for the retrieval of records based on some spatial properties which are not stored explicitly in the spatial data file (Samet, 1995).

Due to this problem, Samet (1995) proposed that it is recommended that the data structuring and storage process should be as implicit as possible so that a wider class of spatial query types can be handled by the geospatial information system, this way developers need not anticipate the types of queries that will be performed a-priori. This does not mean that developers are not let off the hook in terms of maintaining the correctness of their querying operation solutions: the premium of having a flexibly queryable data set is maintaining its representation, especially if the querying operation is part of a digital mapping system which has visualization components. In order to deal with range queries and maintain representation, the data must be sorted (Samet, 1995). The question is, which keys/fields of the data points does the querying mechanism use to sort the data? In the case of spatial data, the sort should be based on all of the spatial keys, which means that it is essentially based on the space occupied by the data, which is defined by the attribute data of the data record. This is the fundamental idea that *spatial indexing* techniques are rooted in (Samet, 1995)and will be discussed further in Chapter 2.4.

2.4 Spatial data indexing

Zhang and Du (2017) define a spatial index as "*a data structure that allows for accessing a spatial object efficiently*". This implies that a spatial index is more than just an operation, it is a data structure and more importantly it is part of the computer abstraction process discussed in Chapter 2.2.2 and illustrated in Figure 2. It may be of interest to further define what a data structure is in the context of computational data retrieval; Levitin (2008) defines a data structure as *"a particular scheme of organizing related data items."* Since it is established that data structures are the essential ingredient of spatial indexes and the spatial indexing procedure, from here on the terms *index* and *data structure* will be used interchangeably to further the discussion.

Xiao (2015) illustrates how indexing a data set to make searching for particular data items in it, is not a new idea, by pointing out that the purpose of the *Index* section of a book is to provide all the page numbers where a word the user wants to find can be found, without the user having to go through all pages to find an instance of that word. For example, if the user is looking for the word "indexing", they can quickly find all the pages that contain it in the index, instead of going to page 1, looking for it, then page 2, then page 3, and so on. (Zhang and Du, 2017) further emphasizes that without indexing, any search for a record would require a "sequential scan" of every record in the data store, resulting in a much longer processing time.

Designing an efficient indexing method that would be operating in a spatial information system is a complex process (Wise, 2016, p. 277), due to issues mentioned in Chapter 2.3. Therefore a unique set of frameworks known as spatial data structures had to be established, researched, and developed within geospatial computing spheres. Spatial data structures give geospatial developers the freedom to apply operations appropriate for spatial data than the ones imposed by the pre-existing non-spatial operations that computer systems expect when facilitating spatial information system operations. (Samet, 1995)

2.4.1 Non-hierarchical spatial data indexing

The discussion on spatial indexing will continue by building upon the searching technique classifications of Nievergelt et al. (1984) that were discussed in Chapter 2.3.1. These 2 classes are termed differently in the different use cases and industries: In computer graphics, they are termed object-space hierarchies and image-space hierarchies, respectively (Sutherland, Sproull, and Schumacker, 1974), in computer architecture they are termed trees and tries (Fredkin, 1960), respectively and in algorithmic analysis they are differentiated as non-hierarchical and hierarchical data structures Samet (1989). In this report, the terms non-hierarchical and hierarchical

PointName	X	Y
Denver	5	45
Omaha	25	35
Chicago	35	40
Mobile	50	10
Toronto	60	75
Buffalo	80	65
Atlanta	85	15
Miami	90	5

Table 2: Sequential list (Samet, 1989)

data structures will be used. Non-hierarchical indexing techniques will be explored here, Hierarchical data structures will be covered in 2.4.2.

Also known as *Address computation methods*, *bucketing methods* or *Point Access Methods* (PAM), non-hierarchical spatial indexers are based on spatial occupancy and prioritize keeping the embedding space fixed regardless of the content of the file (Samet, 1989, 1995; Mamoulis, 2011). According to Mamoulis (2011), PAMs are most appropriate for indexing point data because their space decomposition process enforces disjoint partitioning, which means each point is guaranteed to be assigned to a unique partition of the space and each partition of the space can be directly stored in memory and retrieved by fetching the memory address that it is stored. (Mamoulis, 2011).

The easiest and most traditional non-hierarchical way to store point data records in a sorted way is through a sequential list (Samet, 1989; Wise, 2016). Usually, the data structure that handles this list is the *array* and simply lists the point records sequentially in a particular order, for instance considering a set of eight cities and their x and y coordinates as shown in Table2, it can be noted that they are ordered in ascending order according to their x-coordinates.

The array data structure is available in every programming language and this is because it is extremely efficient at data sorting and storing different data types. This is not a coincidence as computer memory manages all data using an array-like format. The memory element of the computer system schema shown in Figure 4 always reorganizes the data it transfers from and to long-term storage into fixed-length units of bytes or words (4 bytes) and each of these has an address which is simply a number beginning at the first byte in memory to the number of the last byte in memory (Wise, 2016). How the cities' data points would be stored in memory is shown in Table3.

A computer program that runs a data retrieval operation has to calculate the memory addresses in the memory that contains the data records that are needed. Since the array is analogous to computer memory, this operation involves very few extra computation steps. All the program has to do is look for the address of the first record it is looking for in memory, then the addresses of all the other records to be searched for in the array will be deduced from the first record's memory address (Wise, 2016, p. 102). What makes the array efficient spatial data like two-dimensional data is that no other memory resources are needed to manage the row or columns (Wise, 2016, p. 102).

Memory Address	PointName	X-coordinates	Y-coordinates
12905	Denver	5	45
12906	Omaha	25	35
12907	Chicago	35	40
...	...	...	...

Table 3: Storage of an array in computer memory. (Samet, 1989; Wise, 2016)

The main caveat of using arrays is that when very large point datasets are stored this way, more time will be taken to find the address of the first data record, for instance, Given N records and k fields to search on, searching and retrieving all records on such a data structure has an upper-bound time complexity of $O(N.k)$, which is linear (Samet, 1989), as will be seen in the time complexity section of this report in Chapter 2.5, this is not the most efficient computational time complexity.

Another PAM that is as old as cartography and is still popular in digital mapping and spatial analysis is the *fixed grid* data structure (Knuth, 1973; Bentley and Friedman, 1979). This spatial indexing method divides the embedding space into equally-sized cells, each having equal width and height. These cells are often called buckets. The fixed grid is basically a $k \times k$ dimensional array where each cell is a range of memory addresses and each memory address in that range holds the address of a point or has a null value Samet (1989)

Figure 7 is a grid representation for the data in Table2 for a query range of 100×100 unit cells. Each point in the data structure is indexed to exactly one cell according to a set of data indexing rules. A conventional rule-set is to explicitly use the lower and left cells as closed boundaries of each point and use the upper and right cells as potential buckets if a point is not explicitly placed in a bucket. This rule-set aims to ensure that points at cells intersections and boundaries are placed into cells Samet (1989), for instance, the point representing Toronto at (x,y) coordinates (60,75), in the grid data structure in Figure 7, will be put in the cell-centred at coordinates

(70,70).

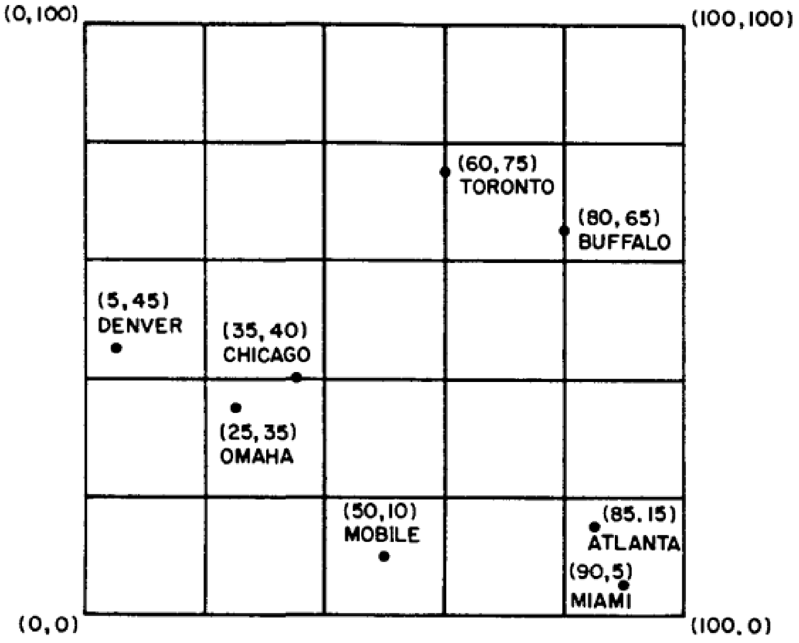


Figure 7: Grid representation corresponding to the point records in Table2 with search cell size of 20 units (Samet, 1989)

The average searching time complexity for performing a range query via the fixed grid method has been shown by Bentley, Stanat, and Williams Jr (1977) to be $O(2F)$, where F is the number of point records found in the query range (Samet, 1989). It is no surprise that this computational time complexity is linear, as the fixed grid is a factored version of the array. The major consideration one has to take into account when deciding on whether to use the fixed grid is the distribution of the points over the embedded space. The more uniformly distributed the points, the more allocated memory will be used, which is efficient considering that data clustering will lead to many cells/buckets having empty data in their address spaces in memory, which leads to the issue of overflow chains. The problem with bucket overflows is that that they lead to excessive memory accesses of the buckets that have clustered points, which negatively impacts performance (Samet, 1989). The solution to this is caching.

2.4.2 Hierarchical spatial data indexing

The focus of the report will turn towards hierarchical data structures. Hierarchical indexing is based on the principle of recursive decomposition of the embedded space that the data is represented on Samet (1988). Recursive decomposition, unlike fixed decomposition of space, denotes that the number of times that the decomposition process is applied is governed by the properties of the data, such as the number of data records, the range of one or more field values or in some cases the geometrical extent of the data (Samet, 1989, p. 2). As seen before, hierarchical data structures have traditionally been referred to as tree-based data structures, because they organize records of data into a top-down hierarchy where data records (also called nodes) near the top are linked to lower data records according to a recursive space decomposition rule-set (Xiao, 2015, p. 71).

To see comprehend how hierarchical data structures are used to index spatial data, some of their basic properties have to be laid out. Xiao (2015, p. 71) provides the most fundamental properties of hierarchical data structures as:

1. Node A is called the *parent node* of B if A is immediately higher than B in the hierarchy, and B is called the *child node* of A.
2. A tree typically has a *root node*, which is the main, if not the only, access to the rest of the data in the tree.
3. The *root* of a tree of course does not have a parent node.
4. There are nodes in a tree that do not have child nodes, and we call them *leaf nodes.*
5. A node and all its child nodes together are called a *subtree* of the original tree.
6. There is a *path* from the root to each of the nodes and the number of nodes on that path is called the *depth* or *height* of that tree.
7. When all the leaf nodes of the same tree have exactly the same depth, the tree is known as a *perfectly balanced tree.*

To gain a deeper understanding of how hierarchical indexing works, the commonly implemented and relatively simple tree-based data structure, the binary tree and its functions will be explored. Figure 8 shows a binary tree that represents a number sequence from 1 to 7 that has been ordered by a rule-set that makes sure that the nodes to the left of each branch have a smaller value than the nodes that are on the right branch of each parent node.

A query operation of all the numbers in the binary tree is demonstrated by the Python function 2 below (Xiao, 2015, p. 72). Here, the tree is provided as a function parameter as a node `t`, and the point to be searched for is `d` in `t` is also provided as a function parameter. The Boolean variable `is_find_only` specifies whether the query should retrieve the value of the node that matches `d` or retrieve the parent

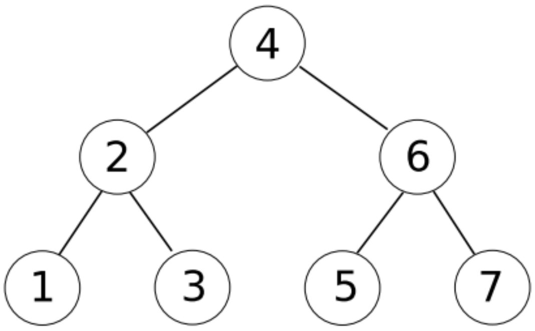


Figure 8: A binary tree that stores and sorts the numbers 1, 2, . . . , 7 (Xiao, 2015, p. 72)

node of the node carrying `d`. Therefore if the aim is to return the node `t`, carrying *d*, then `is_find_only = True`, else if the aim is to determine where to insert a new node, then the supposed immediate parent node of the new node should be returned and `is_find_only = False`.

When the query program is executed, it begins the search at the root node, therefore `t = root node`. The first step is to check if the root has a number (lines 2-3), if it is empty then return `t = root node` and exit the code, if not, the program takes the value in node `t` and compares it with the value of `d`: if `d < value in t`, move to the node attached to the left branch of `t` (lines 4-5). If `d > value in t` move to the node attached to the right branch of `t` (lines 6-7). If `d == value in t`, check if the value of `is_find_only == True`, if it is, return the pointer to node `t` (lines 8-12). The search function proceeds by calling itself recursively with the new node `d` of (line 15) until one of 2 conditions are met: (1) the node that carries the value that matches `d` is found (line 8), (2) the node that carries the value is not in the tree (lines 2 or 13) (Xiao, 2015, p. 74).

Listing 2: Binary Tree Search Python implementation (Xiao, 2015, p. 75)

```
1 def search_bt(t, d, is_find_only=True):
2     if t is None:
3         return
4     if d<t.data:
5         next = t.left
6     else:
7         next = t.right
8     if t.data==d:
9         if is_find_only:
```

```
            return t
        else:
            return
    if not is_find_only and next is None:
        return t
    return search_bt(next, d, is_find_only)
```

A Python function to insert numbers into the binary tree in Figure 8 can be developed by making calls to the search function above. The code of this function is shown in 3. As parameters, the function takes a node `t` that represents a tree and the number to `d` and a Boolean value for the parameter `is_find_only`. The first task of the program is to search and retrieve the parent node of the new node that will carry the number (line 2) by calling the search function in Listing 2. The value of the search is stored in the variable `n`. If `n == None` the tree is empty and the code is exited (lines 2 and 3). If `n` exists then a new node is created (line 5) and is attached to either the left branch or right branch (lines 7-9) of the parent node, depending on whether it is larger than or smaller than the node value.

Listing 3: Binary Tree Insert Python implementation (Xiao, 2015, p. 75)

```
1 def insert(t, d):
2     n = search_bt(t, d, False)
3     if n is None:
4         return
5     n0 = node(d, left=None, right=None)
6     if d<n.data:
7         n.left = n0
8     else:
9         n.right = n0
```

The code examples that have been shown above are not comprehensive and only serve to illustrate the most basic hierarchical insertion and retrieval operations down at the code level. They are, however, useful in serving as a reference point when discussing the slightly more complex spatial data hierarchical data structures that build up to the Point Quadtree (Xiao, 2015).

The benefits of utilizing hierarchical data structures as spatial data indexers allow computational resources to be programmed to focus on computing the subset of the dataset that will require those resources more., additionally, hierarchical data structures a conceptual clearer to understand and easier to implement than most data structures (Samet, 1995).

Other than the Point Quadtree, there are two hierarchical data structures which consider both the X and Y coordinates of the vector data points when indexing them:

1. The first hierarchical data structure is the k-dimensional tree which extends the binary tree to extra dimensions so that it has more than 2 branches (Wise, 2016, p. 222).
2. The second hierarchical indexer of 2-dimensional data is based on the idea of the minimum bounding rectangle (MBR), called R-trees. They are the brainchild of Guttman (1984) who designed them to act as a dynamic organization of a set of multi-dimensional geometric objects, represented by minimum bounding rectangles (MBR). Each node of the R-tree corresponds to a spatially defined MBR that bounds the children node of the node in question. The leaves of the tree contain pointers to the database objects instead of pointers to children's nodes (Manolopoulos, Papadopoulos, Papadopoulos, and Theodoridis, 2006)

There are many other hierarchical indexers that have been developed in the geospatial computing arena that is as optimized as possible to reduce processing latency. Now that hierarchical indexing methods have been looked into comprehensively and a basic understanding of how they function has been established, the discussion will move on to the heart of the project, the Point quadtree data structure in the following Chapter.

2.4.3 Point Quadtree indexing

A quadtree uses both X and Y coordinates to index geospatial data, which is a different strategy from the k-D tree where only one dimension is used at a time. (Xiao, 2015, p. 99) Quadtrees are a set of hierarchical spatial data structures that contain spatial elements that act as pivots of algorithmic operations. The quadtree representation of data is concerned with the representation of data within a fixed predefined spatial region Samet (1988). Node points in quadtrees divide the space region (usually a plane) into four quadrants: lower left (southwest), lower right (southeast), upper right (northeast), and upper left (northwest). Each node point has four subtrees that are a representation of a partitioning of the space around each node that is the same process that partitioned the space region Goldman and Goldman (2007). Each sub-tree contains point elements that have the same characteristics. This is illustrated in Figure 2 where spatial data node A is divided into 4 quadrants and the data points that fall within one of the quadrants, such as data node B, are also divided into 4 quadrants that have data nodes within that that will be divided in the same manner until there are no more sub-data nodes. It is important to note that in the tree visualization of the quadtree on the bottom half of Figure 2, sub-tree nodes that are null, are indicated by white boxes. This representation is important as emphasizes that null placeholders use up computer memory, although it may seem like do not explicitly use any operational memory (Finkel and Bentley, 1974). This detail assists in determining elements of the data structure that impact space complexity.

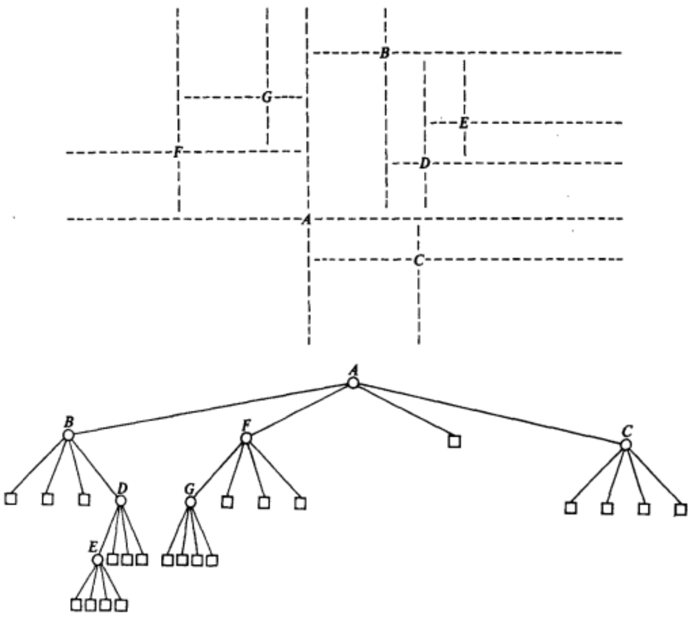


Figure 9: A geometric representation of spatial data points in a quadtree and the corresponding top-down tree structure view of the same quadtree implementation. (Finkel and Bentley, 1974)

At a closer look quadtree algorithms are more complex and less explicit than most tree-based data structures. Gahegan (1989) states that this is a necessary evil because the quadtree presents data in a semi-encapsulated form that makes data files more compact and thus reduces both the amount of data read from the disk and the number of iterations the algorithm has to perform.

According to Finkel and Bentley (1974), the quadtree is very efficient in insertion operations. The Big Oh time complexity they found for the insertion algorithm is logarithmic ($O(logn)$). When it comes to the cost of searching in a quad-tree, the tree's height impacts query runtime the most. Lee and Wong (1977) find that the Big-Oh time complexity of searching quadtree costs ($O(2\sqrt{n})$).

The issues that have been identified with quadtrees include their tightly coupled physical storage and logical structure Yang, Wu, Ren, Fu, Zhang, Wang, Tan, Liu, Ma, and Liang (2008). This means that quadtree data storage is extremely persistent. Although data persistence is good for data recovery, too much of it may impact data movement. This is an inherent shortcoming of lacking the flexibility to scale the data structure without having to change the connection to data in the main memory.

2.5 Indexing Efficiency Analysis

In this section, the analysis of the efficiency of spatial indexing techniques will be discussed. It has been demonstrated earlier that the indexing process involves following a set of instructions to structure data in order to make it much more easily accessible by a system program. This entails that indexing is algorithmic in nature and can be expressed and analysed in terms of algorithmic analysis. It is for this reason that the algorithmic analysis method of determining the efficiency of spatial data structures, particularly the Array and Point Quadtree will be discussed.

2.5.1 Algorithmic Analysis

To estimate the rationality of solving an instance of a compuTableproblem, computational resources (i.e. time and computer memory) of the algorithm involved need to be determined, the process of determining these resources and their feasibility is called *algorithm analysis* (Weiss, 2012, p. 187). It is intuitive to set an assumption that the larger the instance of a problem, the more computational resources will be demanded by the algorithm, and therefore it is expected that the most extreme upper bound of resource demand reflects the proportionality of the size of the problem instance in relation to the algorithm (Bovet and Crescenzi, 1996). Wise (2016, p. 88) puts it in a more straightforward manner: *"the efficiency of the algorithm is affected by the amount of data it will process."*

When it comes to spatial indexing algorithms, (Bentley and Friedman, 1979) points out that there are three main resource factors that can be analyzed to evaluate the efficiency of the indexer in question, these are:

1. Pre-processing time: The time it takes to order spatial data objects in multi-dimensional space before indexing them.
2. Storage: The amount of storage required by the data structure before and after structuring the data.
3. Querying time: The search time or query querying time.

In this report, the third "bottle-necking" resource, which is referred to as the *time complexity* of an algorithm (Bovet and Crescenzi, 1996, p. 42) will be focused on, as it aligns with the objectives of the project.

Continuing with the idea of using upper bounds of resource utilization to inform developers about the efficiency of an indexing algorithm, it is found that these bounds can be denoted by mathematical functions that describe the relationship between the problem size (in this case the number of spatial data records) that the data structures are operating on and the querying time (computational resource) that the same problem size requires to be operated on (Bovet and Crescenzi, 1996, p. 41). This focus on finding an upper-bound has gained this method the name *asymptotic time complexity*. The aim of using mathematical functions is to emphasise the rate of growth of the querying time used as the dataset size increases, rather than to focus on the preciseness of the algorithm's execution evaluation (the preciseness of the outcome is determined by an empirical analysis of the algorithm and this is not a focus of the project). Another reason the growth rates are used over fixed values is that it is near-impossible to retrieve the same running time values, even for the same algorithm running on the same machine or different machines (Weiss, 2012, p. 190). *asymptotic time complexity* seems to be the correct basis on which we can build our theory of complexity.(Bovet and Crescenzi, 1996, p. 43)

To demonstrate the significance of using asymptotic time complexity to determine the efficiency of an algorithm, consider an algorithm with an asymptotic time complexity function of $t_1(n) = 1024n$ and another whose asymptotic time complexity is $t_2(n) = 2^n$, it is clear that for a problem set size $n \leq 14$ the rate of growth of $t_1(n)$ is tremendously greater than $t_2(n)$, alluding that $t_2(n)$ uses less time resource than $t_1(n)$, but when $n > 14$ then the relation flips (Bovet and Crescenzi, 1996, p. 41). This highlights an important factor to consider: The growth rate function value is most important when n is sufficiently large (Weiss, 2012, p. 190). Therefore it can be determined which dataset sizes best fit the problems. This point can be seen in Figure 10 which illustrates four common time complexity functions that most algorithms exhibit when they run over dataset sizes that range from $n = 0$ to $n = 100$, they are the: linear, logarithmic ($NlogN$), quadratic, and cubic functions. As can be seen, the logarithmic function only becomes the most efficient after hitting the 40 data record mark, whereas the quadratic was the most efficient for data-sets of 10 to 4 records and the cubic, 5 to 10 records.

Now that it's established that the time complexity of an algorithm is related to the size of a problem, a standard notation for communicating the efficiency of an algorithm that takes into account the problem size can be introduced that has been used by computer scientists since the 1970's called the 'Big-O notation' (denoted as $O(n)$) (Wise, 2016, p. 90). The Big-Oh notation is used to capture the most dominant term in an asymptotic time complexity function and use it to represent the upper-bound limit of the growth rate of that function. (Weiss, 2012, p. 190). Levitin (2008, p. 53) provides a formal definition and rule set for determining the Big-O notation:

"A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exists some positive constant c and some non-negative integer n_0 such that":

$$t(n) \leq cg(n) \text{ for all } n \geq n_0$$

Figure 11 illustrates this definition, were given a function $t(n)$ that has been determined via experimentation, we can classify it as being no less efficient that a known function $cg(n)$ because its upper bound resource utilization limit is given by $cg(n)$.

There are more precise notations of expressing time complexity but Bovet and Crescenzi (1996, p. 42) have discovered that by using Big-oh notation to express complexity, it is possible to prove that algorithms can be "sped up" by a constant factor by demonstrating that there exists another algorithm that can execute the operations of the first in half the amount of time.

Furthermore, Big-oh notation values can be analysed in terms of their average-case or worst-case (Zhang and Du, 2017). This point is linked to the fact that no execution of an algorithm will produce the same run-time value as any other,

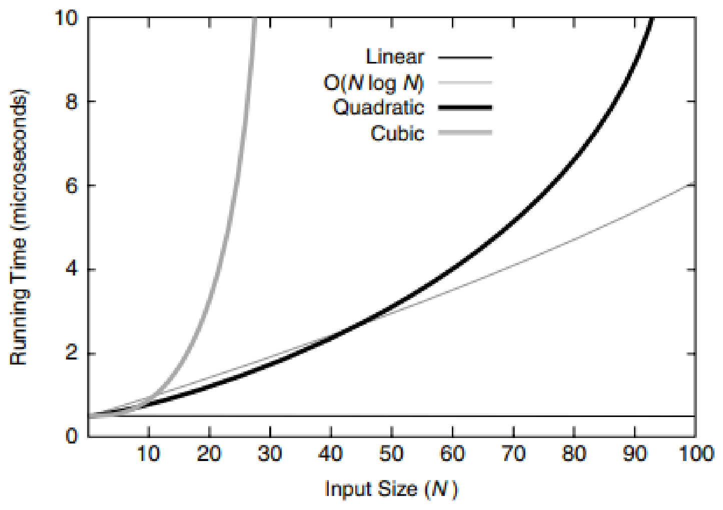


Figure 10: Running times for small inputs (Weiss, 2012, p. 188)

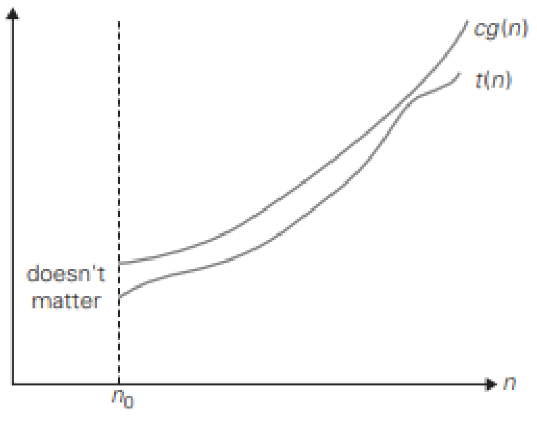


Figure 11: Big-oh notation: $t(n) \in O(g(n))$, since every value of $t(n) \leq cg(n)$ (Levitin, 2008, p. 54)

therefore a relative range of cases can be used to assess efficiency. It is common in GIS literature and geospatial computing literature that there is more concern for determining worst-case (maximum run-times) and average-case (mean run-times) of geospatial algorithms than best-cases (minimum run-time)(Wise, 2016, p. 90).

3 CHAPTER THREE: METHODOLOGY

3.1 Introduction

To meet the objectives of this project an experiments-based methodology that comprises completing these 12 steps will be actioned:

1. Partition a significantly large spatial problem instance (n) into different sub-problem instances of varying, yet increasing sizes of (n).
2. Develop an instance of a vector point data record.
3. Develop an instance of a Point Quadtree indexing data structure.
4. Develop an instance of a Dynamic Array indexing data structure.
5. Determine the minimum, average and maximum run-times of the Point Quadtree for inserting spatial data records into memory.
6. Determine the minimum, average and maximum run-times of the Point Quadtree for range-retrieving spatial data records from memory.
7. Determine the minimum, average and maximum run-times of the Dynamic Array for inserting spatial data records into memory.
8. Determine the minimum, average and maximum run-times of the Dynamic Array for range-retrieving spatial data records from memory.
9. Compare the average and maximum run times of Dynamic Array insertions to Point Quadtree insertions on the same spatial sub-problem instances.
10. Compare the average and maximum run times of Dynamic Array range queries to Point Quadtree range queries on the same spatial sub-problem instances.
11. Determine and compare the average-case asymptotic time complexities of the Dynamic array insertion operation and the Point Quadtree insertion operation.
12. Determine and compare the average-case asymptotic time complexities of the Dynamic array range querying operation and the Point Quadtree range querying operation.

These steps are based off the works of Kothuri et al. (2002), Xiao (2015), Brisaboa et al. (2017) and Wei et al. (2014).

The program development and program run-time executions to complete the experiment steps were conducted using the Python 3.9.13 programming language running a physical machine with these computing specifications:

- Operating System: Windows 10 Home Single Language
- System Type: 64-bit operating system, x64-based processor
- Processor: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz to 2.70 GHz
- Main (Random Access) Memory: 8,00 GB
- Long-term storage 1 TB Hard Disk Drive.

3 programmed applications were written, that work interactively by applying Object-Orientated Programming (OOP) design principles. The OOP design of the classes involved, as well as their interaction with other classes, is laid out in Figure 12. This UML gives a concise overview of the implementation of OOP in the data structures in the experiments. Detailed explanations of each implementation will be provided.

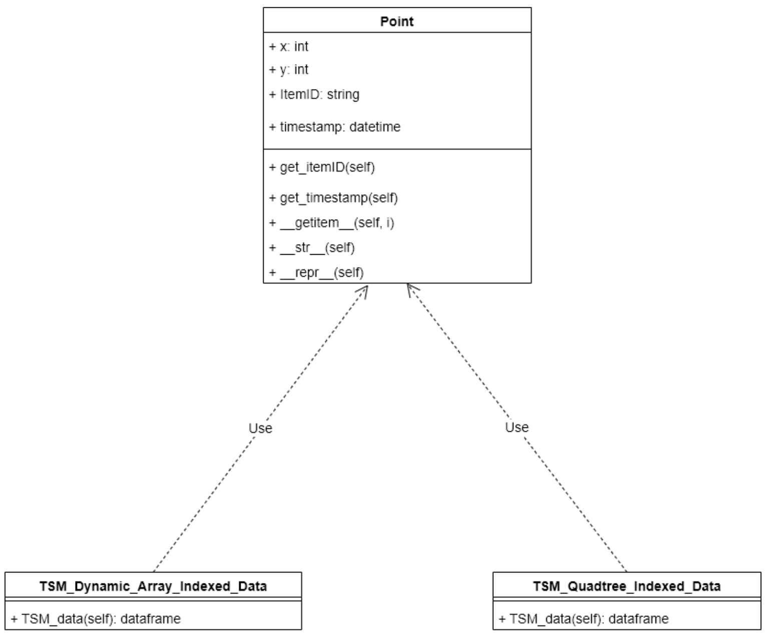


Figure 12: OOP relationships of the class implementations of the data structures involved in the experiment

The choice of using a Dynamic Array as a reference/control data structure is because of the ease of implementation in Python 3 as a simple list data type that does not have a statically set number of address spaces (Goldman and Goldman, 2007, p. 186) and the ease of comprehending its functionality as has been mentioned in Chapter 2.4.1.

3.2 Problem Instances

The spatial dataset that will act as the main problem instance which the indexing algorithms will run over is the City of Cape Town's Town Survey Marks (TSM) point vector dataset that was retrieved from City of Cape Town ArcGIS Data Hub in GeoJSON format. The dataset represents the locations of TSMs, Bench Marks and control points that are spread across the municipal area of the City of Cape Town as vector point data (City of Cape Town, 2021). According to the City of Cape Town (2021), the geospatial vector data of these objects were collected using ground surveying methods of the highest possible accuracy standards and measured in the Hartebeesthoek 94 Datum and the World Geodetic System 1984 (WGS84) coordinate reference system. This means that the issues surrounding autocorrelation, the coordinate system of the vector data points and ease of decomposition of data (as mentioned in Chapter 2.2.5) are taken care of by this dataset.

The data set, which includes approximately 11900 data records with the following field attributes: OBJECTID, PNT, Y_WGS_84, X_WGS_84, SRC, HGHT, HGHT_SRC, RMRK, STS, CTGY and the most important field to this project is the geometry field that carries the coordinates of the vector point data in WGS84 latitude and longitude coordinates. A majority of the fields above come inherently with the dataset and are of no use for the purposes of this experiment, therefore a data ingestion and cleaning process will be needed in the data structures scripts that will handle the unnecessary data.

3.3 Data-set Partitioning

To measure the rate of growth of time resource utilization as the problem instances increase, the size of the input data sets will be partitioned into sub-data set sizes that increase in size until they are the size o the original data set. These subsets are of size $n = 1000$,$n = 2380$, $n = 4760$, $n = 7140$, $n = 9520$ and $n = 11900$ and each subset is visualized in Figures 13, 14, 15, 16, 17 and 18 respectively. The GeoJSON source codes for each partition are too large to present in this report however they may be accessed via these links:

- $n = 1000$
- $n = 2380$
- $n = 4760$
- $n = 7140$
- $n = 9520$

- $n = 11900$

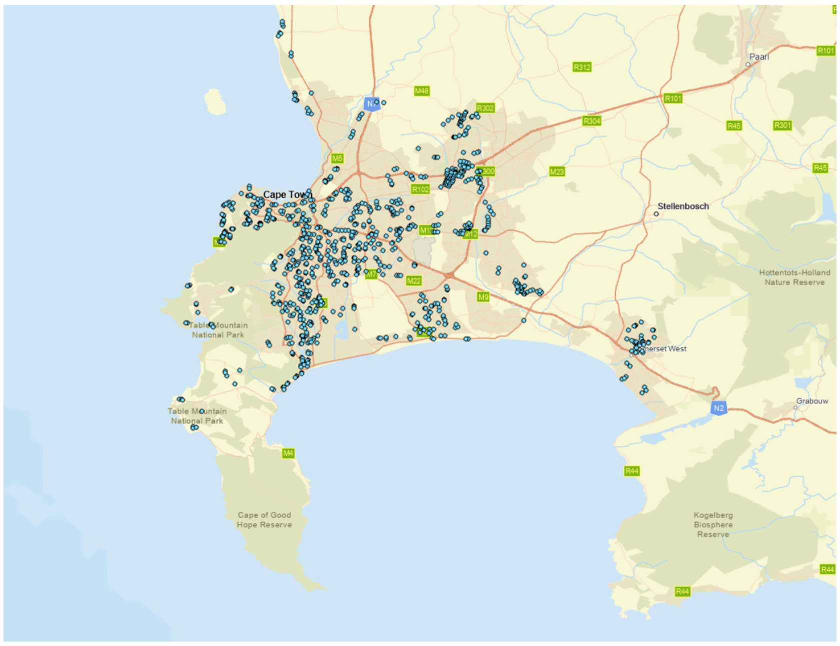


Figure 13: Visualization of $n = 1000$ data records of the TSM dataset

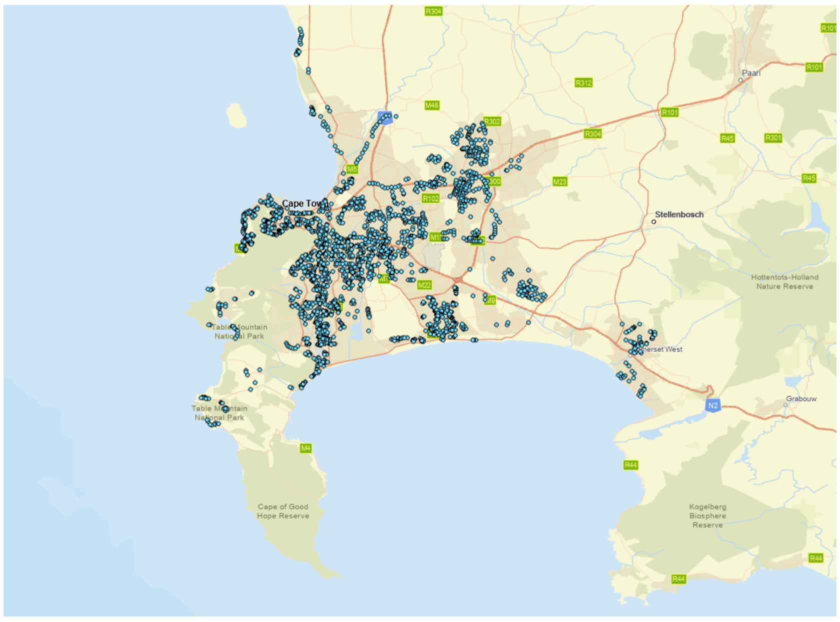


Figure 14: Visualization of $n = 2380$ data records of the TSM dataset

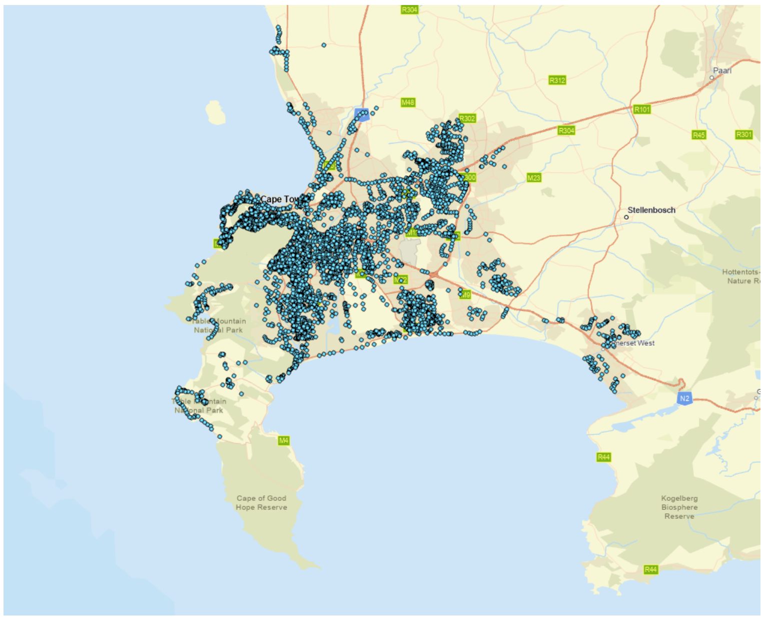

Cape Town
Paarl
Stellenbosch
Table Mountain National Park
Somerset West
Hottentots-Holland Nature Reserve
Cape of Good Hope Reserve
Kogelberg Biosphere Reserve

Figure 15: Visualization of $n = 4760$ data records of the TSM dataset

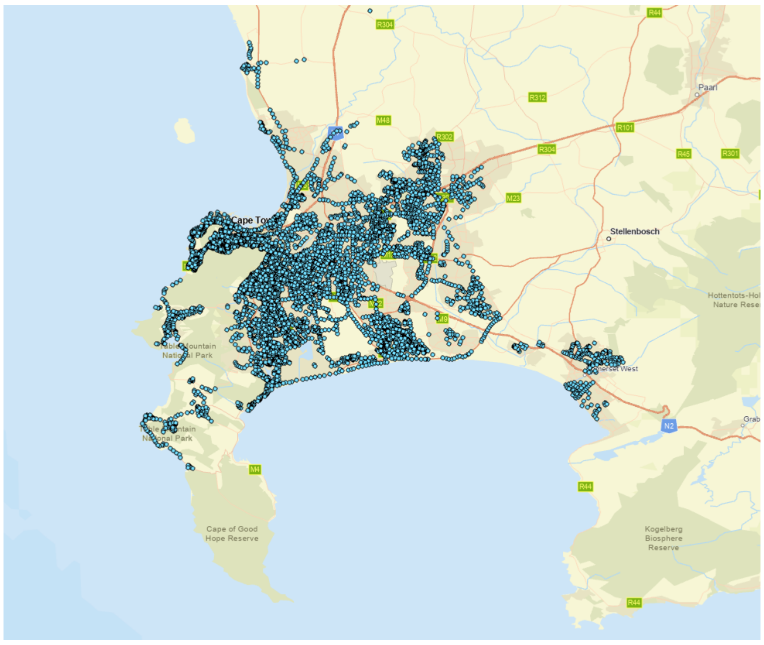

Cape Town
Paarl
Stellenbosch
Table Mountain National Park
Somerset West
Hottentots-Holland Nature Reserve
Cape of Good Hope Reserve
Kogelberg Biosphere Reserve

Figure 16: Visualization of $n = 7140$ data records of the TSM dataset

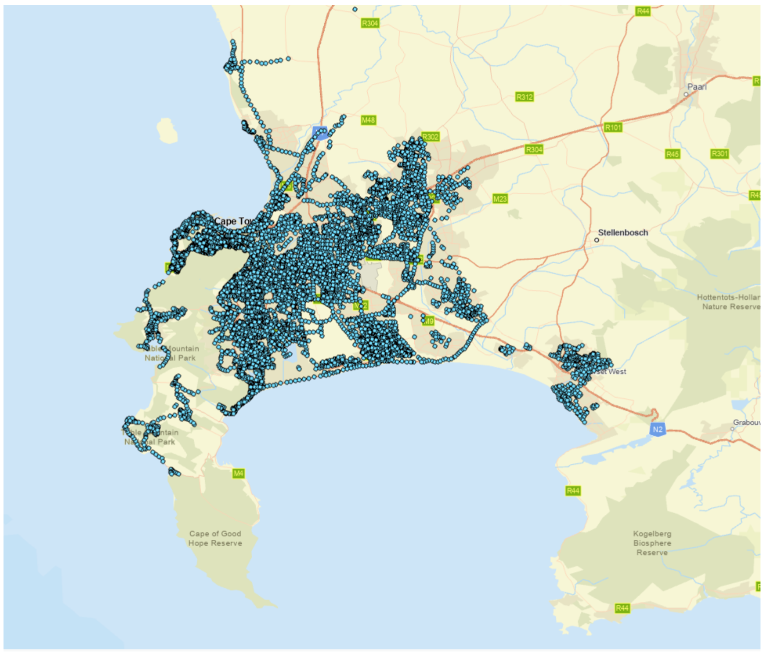


Figure 17: Visualization of $n = 9520$ data records of the TSM dataset

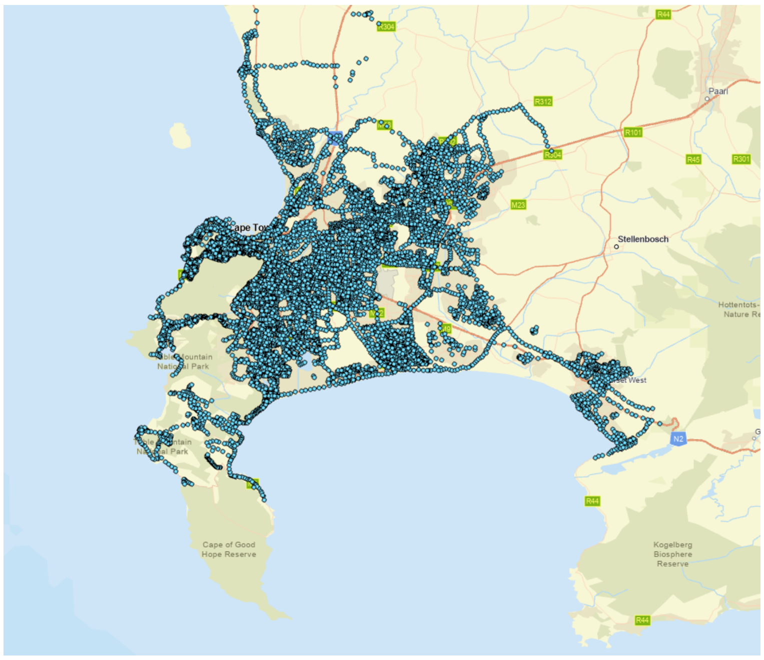


Figure 18: Visualization of $n = 11900$ data records of the TSM dataset

3.4 Vector Point Class

To take advantage of the cohesion and decoupling benefits of the OOP design pattern, an object abstraction of a vector Point data record has been implemented as a class. This allows the data structure programs to treat each data record as an independent unit of data with attribute-based fields, instead of set of unrelated fields that have to be joined together before any computation can be made on them.

The implementation of this Point class is based on Xiao (2015, p. 12-13) implementation with tweaks made to it to fit the use-case of our experiments, the code implementation can be found in Appendix A as Listing 4 These are the methods that are utilised the most during the insertion and retrieval steps of executions of the data structures:

- The `getitem()` method, allows the calling function to explicitly return either the X or Y coordinates of the object 0 or 1 respectively.
- The `__repr__()` and `__str__()` methods allow the calling function to get the coordinates of the point as either a string or numerical value respectively.
- `getitemID()` method returns the identifier of the point, this value was taken from the PNT attribute of the TSM data set.
- `get_timeStamp()` this returns the timestamp of when the point data item was pulled from memory to the dataframe of the respective data structure.

3.5 Data Structure Classes

Following the OOP design pattern, the Point Quadtree and the Dynamic Array indexing data structures were implemented as classes with driver main methods and inherent methods of their own. Both Classes had very similar functionality and differed only in the indexing operation calls. Figure 19 shows the path of instructions the respective classes undergo to read the TSM GeoJSON files from memory, create tabular dataframes to manage the data, perform data cleaning steps and eventually perform the 100 insertion and range queries and collect the runtimes of each operation and place it in a comma-separated values file for further analysis.

The Python 3 packages that were used in the code base to fulfil the executions of the processes in Figure 19 are mentioned below:

- `pandas`: Handles the dataframe instantiation and data manipulation methods of the dataframe.
- `time`: Handles the operating system time manipulation functionality that allows creating a run-time stopwatch to record the runtimes in seconds

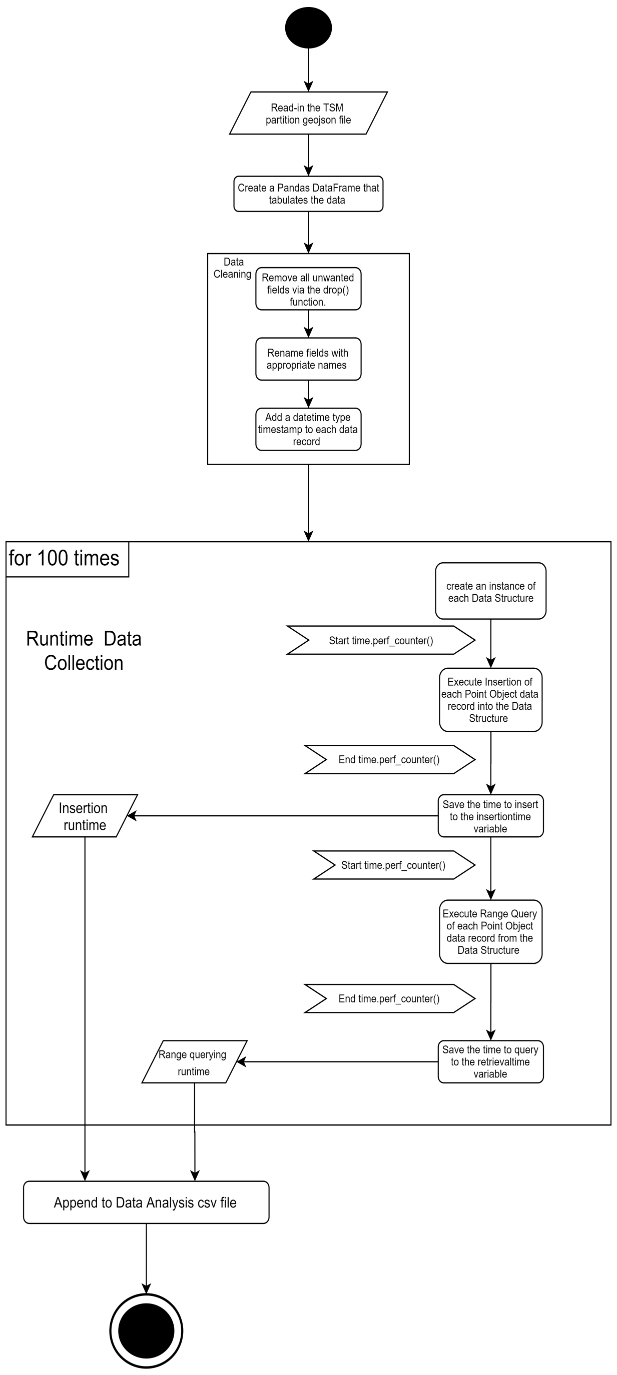
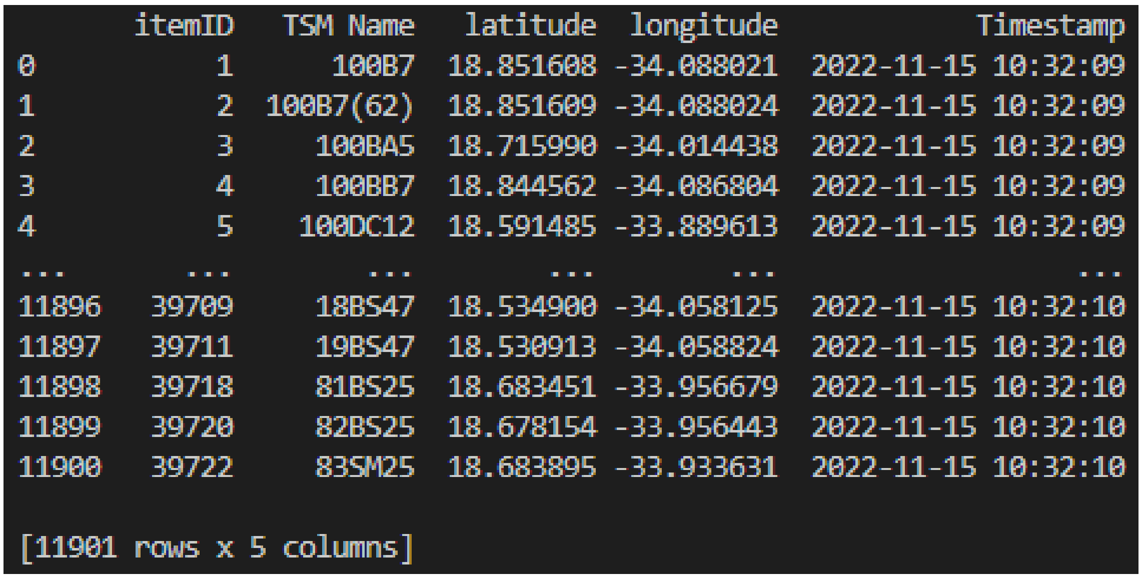

Read-in the TSM partition geojson file
Create a Pandas DataFrame that tabulates the data
Data Cleaning
Remove all unwanted fields via the drop() function.
Rename fields with appropriate names
Add a datetime type timestamp to each data record
for 100 times
Runtime Data Collection
create an instance of each Data Structure
Start time.perf_counter()
Execute Insertion of each Point Object data record into the Data Structure
End time.perf_counter()
Save the time to insert to the insertiontime variable
Insertion runtime
Start time.perf_counter()
Execute Range Query of each Point Object data record from the Data Structure
End time.perf_counter()
Save the time to query to the retrievaltime variable
Range querying runtime
Append to Data Analysis csv file

Figure 19: The set of steps that each data structure script will perform in order to produce the 100 runtime values for each data subset.

- `pandas_geojson`: Handles the conversion of GeoJSON data to pandas framework and vice-versa.
- `csv`: Handles the functionality to write the recorded runtime values to CSV file.

The reason dataframes were used to structure the data is for ease of execution and confirmation of data cleaning processes. As can be seen in Figure 20, the data is more human-readable as compared to the GeoJSON format of the same data.



```
       itemID   TSM Name   latitude  longitude            Timestamp
0           1      100B7  18.851608 -34.088021  2022-11-15 10:32:09
1           2  100B7(62)  18.851609 -34.088024  2022-11-15 10:32:09
2           3     100BA5  18.715990 -34.014438  2022-11-15 10:32:09
3           4     100BB7  18.844562 -34.086804  2022-11-15 10:32:09
4           5    100DC12  18.591485 -33.889613  2022-11-15 10:32:09
...       ...        ...        ...        ...                  ...
11896   39709     18BS47  18.534900 -34.058125  2022-11-15 10:32:10
11897   39711     19BS47  18.530913 -34.058824  2022-11-15 10:32:10
11898   39718     81BS25  18.683451 -33.956679  2022-11-15 10:32:10
11899   39720     82BS25  18.678154 -33.956443  2022-11-15 10:32:10
11900   39722     83SM25  18.683895 -33.933631  2022-11-15 10:32:10

[11901 rows x 5 columns]
```

Figure 20: Snippet of the dataframe of the City of Cape Town's Town Survey Marks data set.

3.5.1 DynamicArray Class

The Dynamic Array data structure class was implemented using Python 3 and it was named `TSM_Dynamic_Array_Indexed_Data.py` and the source code can be found in Appendix A as Listing 5

Its insertion operation involves using the Python Lists method `append(point)` (line 57 in Listing 5) to append a point which is an object of the Point class described in that Chapter 3.4.

Its range query operation executes by taking advantage of the Python 3 `range(List)` method that automatically fetches all the items in the parameter, which is an instance of a Dynamic Array, via the List datatype. To run this range query a `for-loop` sequentially visits every address checking if there is a Point object or a `None` (null) value in it (lines 72 to 80 in Listing 5).

3.5.2 Point Quadtree Class

The Point Quadtree data structure class was implemented using Python 3 and it was named `TSM_Quadtree_Indexed_Data.py` and the source code can be found in Appendix A as Listing6. This class is a combination of the Point Quadtree class provided by Xiao (2015, p. 118) and the experiment functionalities shown in Figure 19. An inner class called `PQuadTreeNode` will handle the instantiation and methods of a Point Quadtree that will be able to execute the insertion (`insert_pqtree`) and range query (`search_pqtree`) operations needed, meet the objectives of the experiment.

The insertion operation executed by the `insert_pqtree(quadtree)` is similar to the insertion process of the binary tree that was explained in Chapter 2.4.2 and shown in Listing 3 except that instead of just comparing numbers along a number line, it compares x coordinates and y coordinates and places the output into one of four memory addresses (lines 45 to 55 in Listing 6).

The range query operation executed by the `search_pqtree(quadtree)` also runs similar to its binary tree counterpart shown in Listing 2, but with a focus on comparing the x and y coordinates and placing them in one of four memory addresses (lines 26 to 43 in Listing 6).

3.6 Runtime analysis

Runtime is the amount of time it takes for an operation to complete its computation, therefore it only makes sense to measure it by using a universally standard unit of time measurement: the second. Although simple and intuitive, it (Levitin, 2008, p. 44) states that there are drawbacks to this approach, namely the fact that the time it takes to run a program on a computer is dependent on the specifications of the computer, the quality of the factorization of the source code and the programming language compiler used. It is for this reason that multiple measurements of the runtimes of the insertion and range query were taken for each dataset subset. 100 runtime measurements for each sub-problem instance is the ideal number of measurements and this was the number of measurements actually taken. Therefore 100 runtime measurements for dataset size were recorded into CSV files, each corresponding to the problem subset that the programs ran over. A `while-loop` was implemented in the source code of the data structure classes to enforce 100 runtime measurements are taken and recorded (lines 95 to 153 in Listing 5 and lines 39 to 103 in Listing 6)

To gain a clearer perspective of the run time behaviour as the sub-problem size n increased, the longest (maximum), shortest (minimum) and average (sum of all runtimes divided by the number of measurements) runtimes were calculated and recorded for each corresponding dataset size from $n = 1000$ to $n = 11900$ in tables. The data in these tables were used to determine the values and make the comparisons that were stipulated in steps 5 to 11 mentioned in the Introduction of the Methodology.

3.7 Asymptotic Time complexity analysis

To determine the efficiency of the Point Quadtree effectively, the asymptotic time complexity of its growth rates will have to be determined and compared to the asymptotic time complexity of the Dynamic Array on the range of dataset sizes of the experiment. To do this the mathematical growth functions $t(n)$ of the wort-case insertion and range query runtimes of each data structure will be determined by performing a correlation coefficient (R^2) curve-fitting on the runtime vs dataset size scatter plot of the data in the Microsoft Excel software package. The aim is to get the mathematical functions of the fitted curves that have values of $R^2 > 0.9$. Once these functions are determined, they will be treated as the $t(n)$ functions of the respective and the Big-Oh notation will be applied to the functions so as to determine their dominant terms. Using the Basic asymptotic efficiency classes Tablepresented by Levitin (2008, p. 59) shown in Table4, classifications of the efficiency of the Point Quadtree and Dynamic Array can be made and compared according to the same table.

Class	Name	Description
1	constant	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size rows infinitely large
$logn$	logarithmic	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm. Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	linear	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$nlogn$	linearithmic	Many divide-and-conquer algorithms, including mergesort and quicksort in the average case, fall into this category.
n^2	quadratic	Typically characterizes the efficiency of algorithms with two embedded loops. Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples
n^3	cubic	Typically characterizes the efficiency of algorithms with three embedded loops. Several nontrivial algorithms from linear algebra fall into this class.
2^n	exponential	Typical for algorithms that generate all subsets of an n-element set. Often, the term exponential is used in a broader sense to include this and larger orders of growth as well.
$n!$	factorial	Typical for algorithms that generate all permutations of an n-element set.

Table 4: Basic asymptotic efficiency classes (Levitin, 2008, p. 59)

4 CHAPTER FOUR: RESULTS AND ANALYSIS

The results of implementing the steps that were stipulated in Chapter 3 are presented in this Chapter. The focus will be on the results that contribute towards meeting the objectives of this project (see 1.5) and eventually answer the research questions raised (see 1.4).

4.1 Dynamic Array operations

Table5 illustrates the Average, minimum and maximum runtimes in seconds of Dynamic Array insertion and range query retrieval operations for each of the 100 experimental runs, that is out of a set of 100 runtime measurements of the Dynamic Array's insertion and range query recordings for a dataset size n, the minimum runtime value, maximum runtime value and average of all 100 runtime values are be are represented in the table.

Dataset size (n)	Average Insert time (sec)	Maximum Insert time (sec)	Minimum Insert time (sec)	Average retrieval time (sec)	Maximum retrieval time (sec)	Minimum retrieval time (sec)
1000	0.000815804	0.0068955	0.0003563	0.003473501	0.0134206	0.0015922
2380	0.001608171	0.0097696	0.0007179	0.01017401	0.0277435	0.0047864
4760	0.003006749	0.020333	0.0012524	0.018802387	0.0507466	0.0063156
7140	0.003907521	0.0187257	0.0013127	0.022622538	0.0855918	0.0096539
9520	0.004157443	0.0112565	0.0018159	0.028268529	0.0832944	0.0123603
11900	0.0080	0.0224861	0.0024525	0.050755636	0.200774	0.020683

Table 5: Average, Maximum and Minimum run-times for Dynamic Array operations for each subset of the spatial data

The Line graph in Figure 21 and line graph in Figure22 were generated from Table5 and they display the relationship between the insertion operation and range query point retrieval operation and each operation's respective runtime of the Dynamic Array and the dataset size that the Dynamic Array experimental run executed on for the respective operations. The runtime is represented in seconds on the y-axis and the dataset size is represented as the number of TSM vector data points (N) that were either inserted or retrieved on the x-axis.

It can be seen, particularly in the graph in Figure 21 that the insertion operation of the Dynamic Array does not exceed 23 milliseconds, with the largest maximum runtime (0.0224861 seconds) corresponding to the largest dataset size of 11900 vector points and the largest minimum runtime (0.0024525 seconds) also corresponds to the largest dataset size of 11900 vector points. The average insertion time and minimum insertion time curves illustrate a growth pattern relationship between dataset size and runtime for each of the dataset sizes used in the experiment. However, the maximum insertion time curve does not indicate this relationship, with dataset sizes in the high ranges having lower corresponding runtime values than their predecessors, particularly for the dataset with $N = 9520$ points (with a maximum runtime of 0.0112565 seconds), where its runtime is significantly lower than the preceding dataset $N = 7140$ with a maximum runtime of 0.0187257 seconds). The Average runtime values of the Dynamic Array insertion operations are sandwiched between the minimum and maximum runtimes, although the differences between the runtimes of the same dataset's values are larger between the maximum and average than for the average and minimum (i.e. the average runtimes are closer to the minimum runtimes than they are to the maximum runtimes).

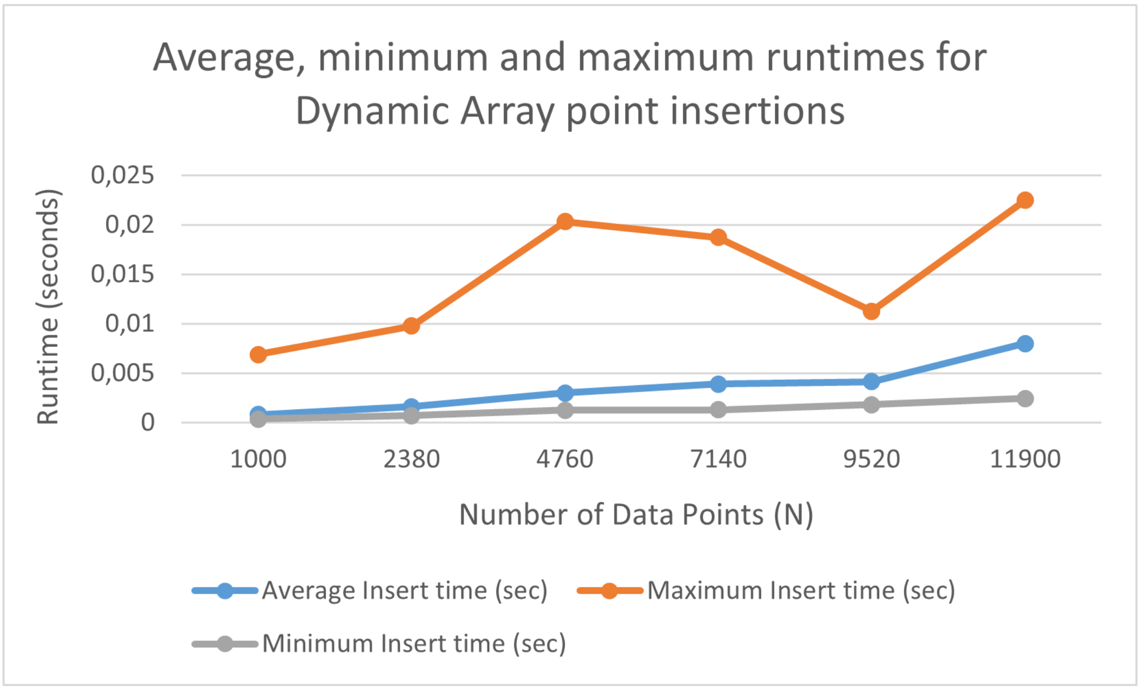


Figure 21: Average, minimum and maximum runtimes for Dynamic Array point insertions

The graph in Figure 22 reveals the runtime-to-dataset size patterns about the range query point retrieval operations of the Dynamic Array. The runtime values of retrievals range from centiseconds (a hundredth of a second) to deciseconds (a tenth of a second) with the largest maximum runtime of a Dynamic Array having the value of 0.200774 seconds (which is nearly 10× more than for the largest maximum runtime of for an insertion operation by the Dynamic Array) and it also corresponds to the largest dataset size of 11900 vector points. The largest minimum runtime for a Dynamic Array retrieval is 0.020683 seconds and approximately the value of the largest maximum insertion time of the Dynamic Array indexer. Unlike for the insertion operations, the maximum, average and minimum retrieval time curves of the Dynamic array show an increase in runtime as the dataset size increases. However, these increases are not at the same rates as can be hinted by the sharp increase in runtime from dataset $N = 9520$ to $N = 11900$ for the maximum retrieval times. Just as in the graph in Figure 21, the average retrieval runtime curve is sandwiched between the minimum and maximum runtime curves, with its values being closer to the minimum curve's values.

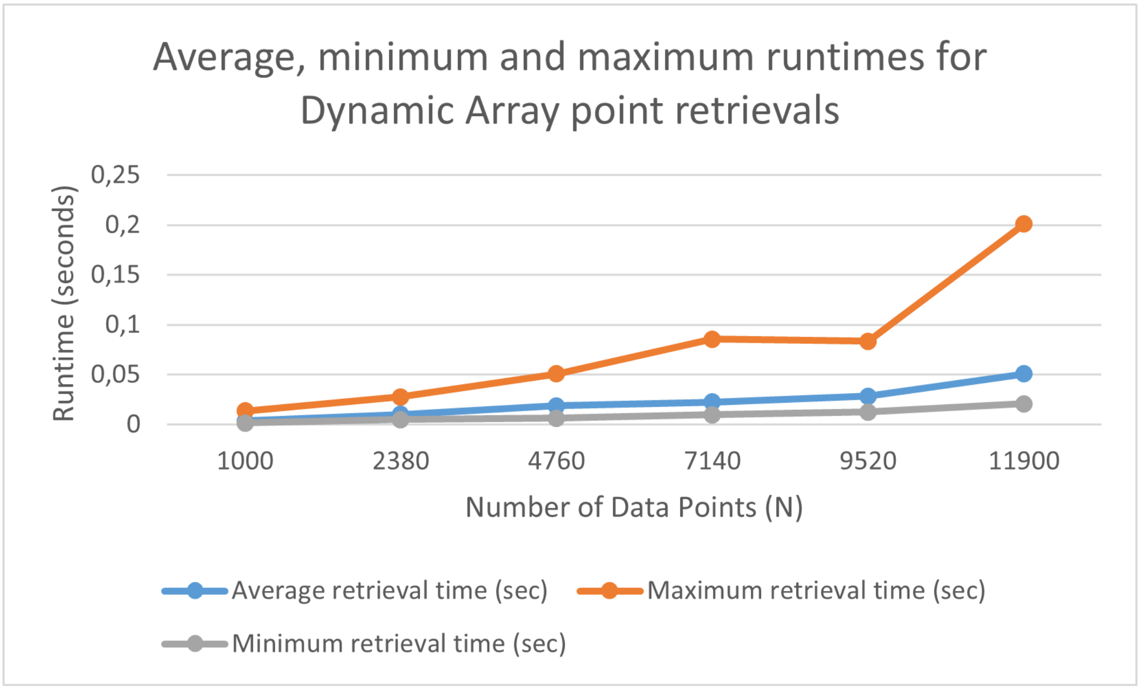


Figure 22: Average, minimum and maximum insertion and range query retrieval runtimes for Dynamic Array point retrievals

4.2 Point Quadtree operations

Table6 shows the average, minimum and maximum insertion and range query retrieval operation runtime values for each of the 100 experimental runs performed by the Point Quadtree on the dataset partitions of the experiment. It presents the data using the same format as Table5.

Dataset subset size	Average Insert time (sec)	Maximum Insert time (sec)	Minimum Insert time (sec)	Average retrieval time (sec)	Maximum retrieval time (sec)	Minimum retrieval time (sec)
1000	0.028928774	0.100703	0.0112441	0.050421935	0.1195692	0.0216348
2380	0.059992209	0.1554239	0.0282311	0.110723451	0.4791164	0.0587069
4760	0.131782637	0.4776162	0.0661706	0.232842234	0.5450318	0.1256467
7140	0.233348967	0.3742927	0.1024027	0.432947863	0.7617558	0.1919375
9520	0.252912389	0.4783397	0.1333603	0.454215907	0.8908975	0.2469484
11900	0.262244116	0.8700111	0.1695314	0.468011174	1.1742595	0.3177265

Table 6: Average, Maximum and Minimum run-times for Point Quadtree operations for each subset of the spatial data

The data shown in 6 is visualised by the line graphs 23 and 24 for better interpretation. These graphs show how the insertion and range query point retrieval operation's respective runtimes change as dataset size changes when it is executed by the Point Quadtree. As with the Dynamic Array operations graphs, the runtimes are represented in seconds on the y-axis and the dataset size is represented as the number of TSM vector data points (N) that were either inserted or retrieved on the x-axis.

Table23 shows that the insertion operation runtimes of the Point Quadtree range from 0.0112441 seconds (lowest minimum runtime) to 0.8700111 (highest maximum runtime). The minimum and average insert time curves increase in runtime value as dataset value increases for all dataset values, whereas the maximum insert time curve does not fully reflect this growth. This is indicated by dataset size $N = 4760$, having a runtime to insert the data being longer than for inserting $N = 1000$, $N = 2380$, $N = 7140$ and $N = 9520$ vector data points, this is seen as a kink in the graph in Figure 23. An interesting thing that is highlighted by the graph in Figure 23 and can also be noted in Table6 is that the average runtime values express an asymptotic behaviour for the dataset sized $N \geq 7140$, with differences in the runtime of fewer than 5 centiseconds between the average runtimes of datasets $N = 7140$, $N = 9520$ and $N = 11900$. The minimum runtimes show a relatively flat linear growth rate in runtime values as dataset size increases. Regarding the largest minimum runtime, its value is 0.1695314 seconds and it corresponds to the largest dataset partition of $N = 11900$.

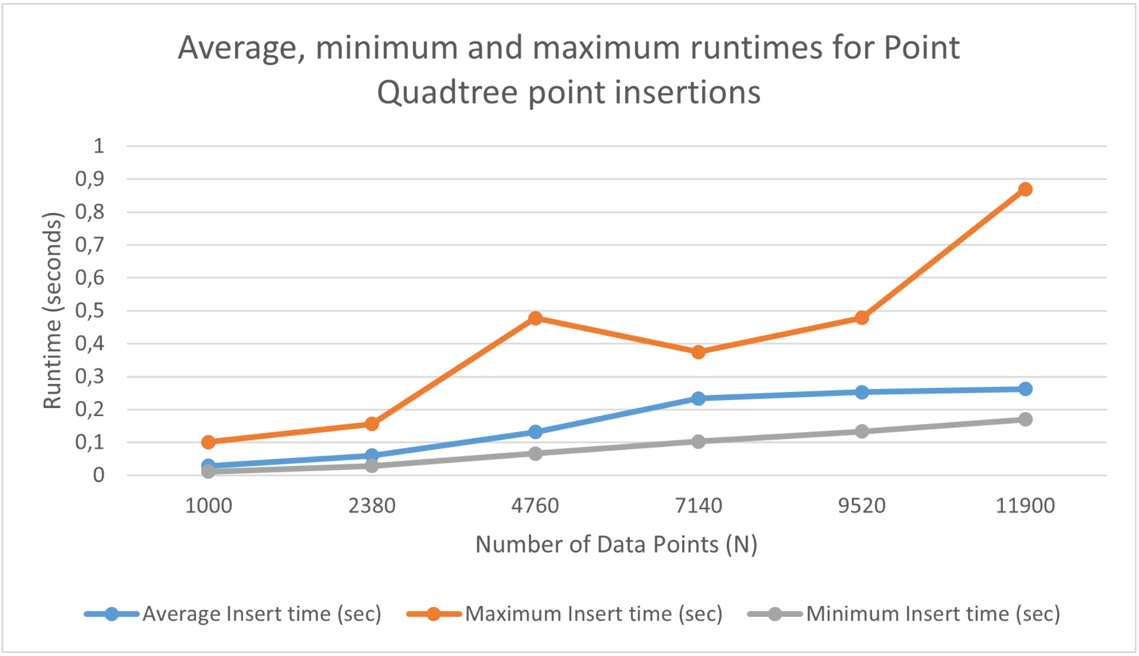


Figure 23: Average, minimum and maximum runtimes for Point Quadtree point insertions

Looking at the runtime-dataset size relationship of Point Quadtree range query retrievals illustrated by the graph in Figure 24, the first significant point to note is that the execution begins approaching 1 second as a runtime figure. The graph in Figure actually shows that the highest maximum runtime for a Point Quadtree retrieval exceeds 1 second (1.1742595 seconds to retrieve $N = 11900$ vector data points). The lower runtime bound for executing a Point Quadtree retrieval is 0.3177265 seconds (longest minimum time). The maximum and the minimum retrieval times display linear growth patterns but with the maximum retrieval time curve having a steeper gradient than the minimum retrieval time curve. Just like the average insertion runtimes, the average retrieval runtimes of the Point Quadtree display asymptotic behaviour from dataset size $N \geq 7140$.

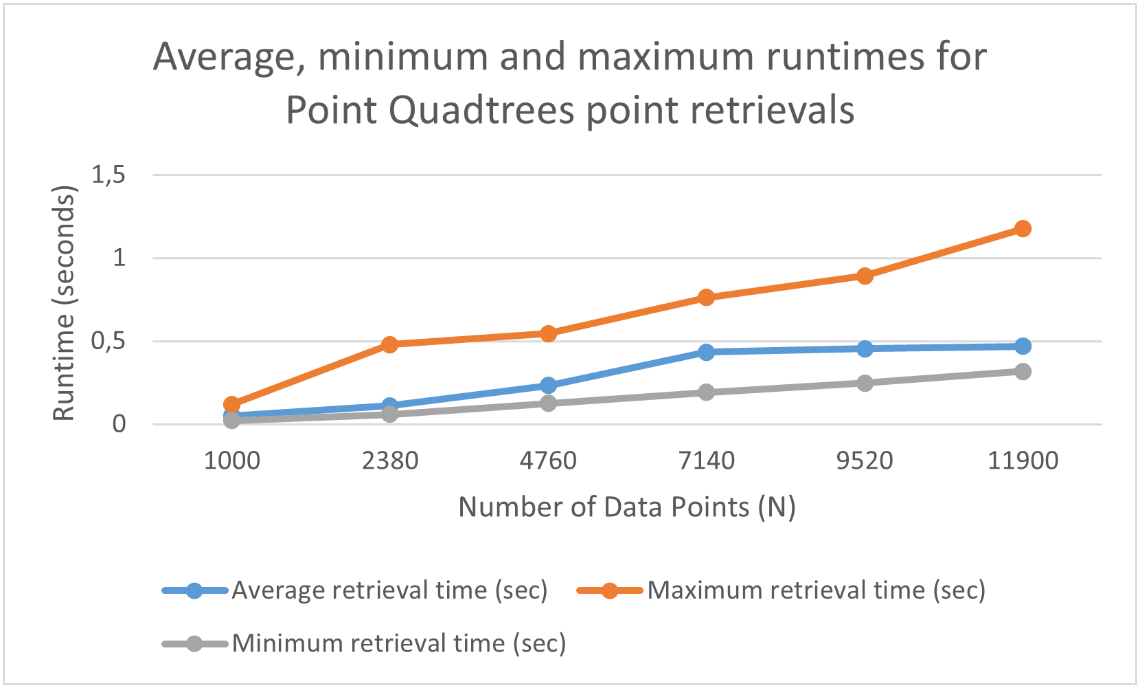


Figure 24: Average, minimum and maximum runtimes for Point Quadtree point retrievals

4.3 Comparisons

Now that the insertion and range query average, minimum and maximum runtimes for the Dynamic Array and the Point Quadtree data structure classes have been explored respectively, the focus will be turned toward comparing the two indexing data structures' average and maximum runtimes for both operations. A comparison of the minimum runtimes is not conducted because the dataset is geospatial in nature and as seen in Chapter 10, the minimum runtime communicates the best-case resource consumption scenario, which is less likely to occur in typical geospatial computing operations (Wise, 2016, p. 90). The graphs discussed in this subsection were all generated using both Table5 and Table6 and follow the format of displaying runtime values in seconds on the y-axis and dataset sizes as the number of vector point processed denoted as N on the x-axis.

The insertion runtimes of the Dynamic Array and Point Quadtree are compared first. In the average-case, illustrated by the Graph Figure 25, what is shown is that as the data subset increases in size, the Point Quadtree takes extensively more runtime than the Dynamic Array to insert the same dataset sizes. The difference in runtime is the greatest for the largest dataset size $N = 11900$, where the Dynamic Array inserts the vector points in 0.0080 seconds and the Point Quadtree inserts the data in 0.262244116 seconds. That means that the Point Quadtree takes 0.254244116 seconds more than the Dynamic Array to insert the same amount of data ($N = 11900$) in the average-case. It is a point of interest that the differences in insertion runtime between the Dynamic Array and the Point Quadtree increase rapidly when $1000 \leq N < 7140$, then increases at a slower rate when $N \geq 7140$ due to the asymptotic runtime rate of the Point Quadtree.

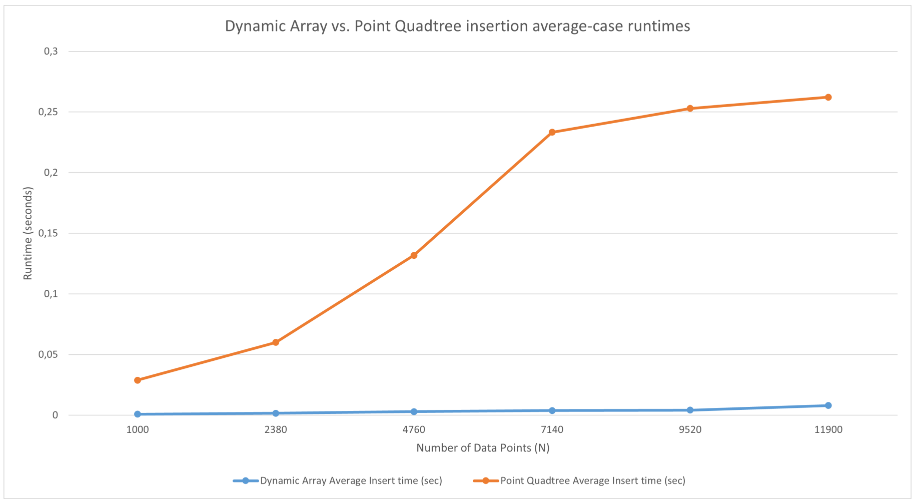


Figure 25: Dynamic Array vs. Point Quadtree insertion average-case run-times

The description above is of the average-case insertion runtimes, now the worst-case runtimes are described for the insertion operations. The graph in 26 will be used as a reference. Again what is seen is a great disparity between the Dynamic Array insertion runtimes and the Point Quadtree insertion runtimes. The major difference is the growth rate that of the worst-case runtime of the Point Quadtree, instead of an asymptotic pattern, it displays an exponential pattern, however, this cannot be fully asserted due to the "kink" runtime value of $N = 4760$. The difference in insertion runtime for the largest dataset size $N = 11900$ between the Dynamic Array (0.020683 seconds) and the Point Quadtree (0.3177265 seconds) is 0.2970435 seconds, meaning that Point Quadtree takes 0.2970435 seconds more than the Dynamic Array to insert the same amount of data ($N = 11900$) in the worst possible case.

Now the range query data retrieval runtimes of the Dynamic Array and Point Quadtree are compared. The average-case is begun with and is visualized in Graph 27. Runtime growth rate patterns and differences that are similar to those of the average-case insertion comparisons (Graph 25). The only major difference between the average-case insertion and retrieval runtimes is the runtime range they cover: The Point Quadtree insertion runtimes range from 0.028928774 seconds to 0.262244116 seconds in the average case, whilst the retrieval runtimes range from

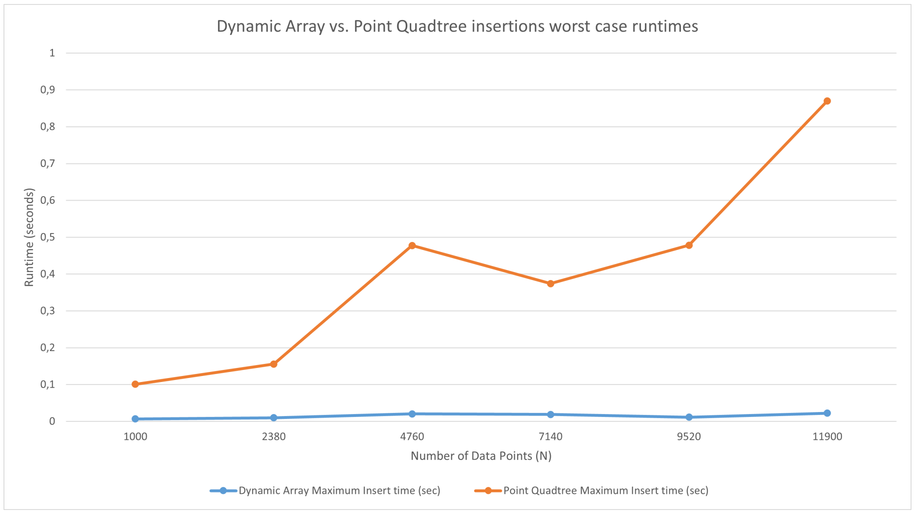


Figure 26: Dynamic Array vs. Point Quadtree insertion worst-case run-times

0.050421935 seconds to 0.468011174 seconds, approximately a 2× increase in the time range from inserting to retrieving. For the Dynamic Array, the ranges are 0.000815804 seconds to 0.0080 seconds for inserting and 0.003473501 seconds to 0.050755636 seconds for retrieving.

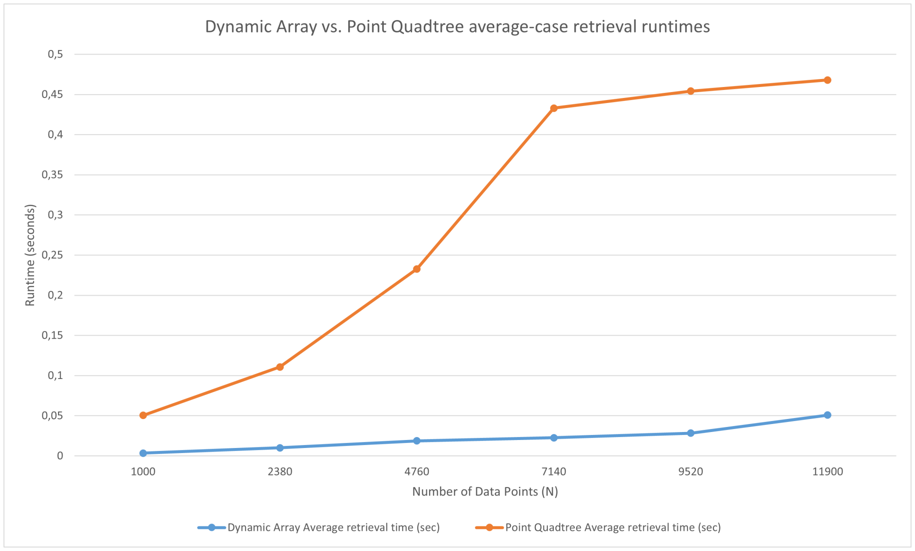


Figure 27: Dynamic Array vs. Point Quadtree retrieval average-case run-times

In the worst-case runtimes of the retrieval operations, the Graph 28 will be used as a reference. As in the worst-case for insertions, a growing disparity in the growth rates of the Dynamic Array retrieval runtimes and the Point Quadtree retrieval runtimes exist. The difference is that the growth rate of the worst-case runtime of the Point Quadtree is neither an asymptotic pattern nor exponential but linear.

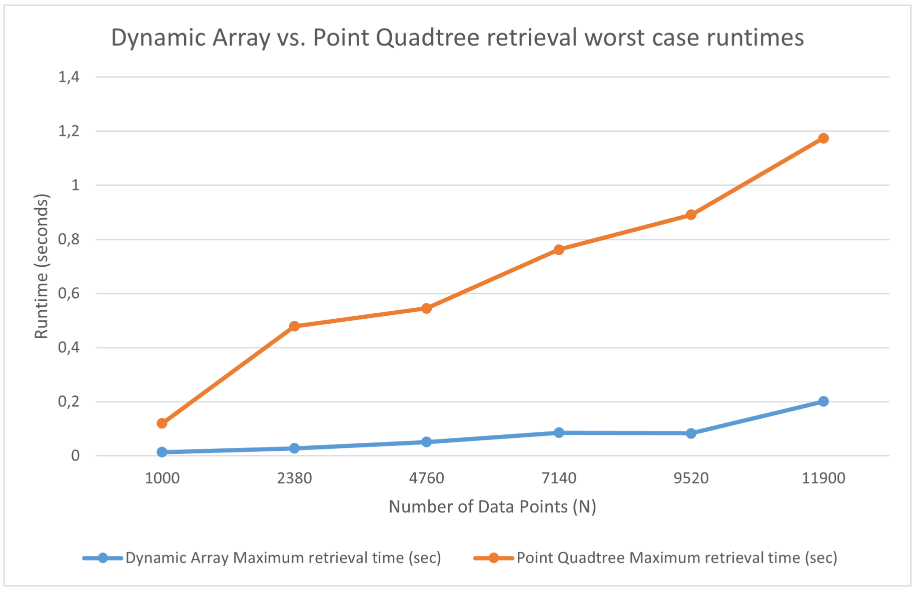


Figure 28: Dynamic Array vs. Point Quadtree retrieval worst-case run-times

4.4 Time Complexity differences

The complete the analysis of the results produced by the experiment runs, the time complexity analysis of the average-case insertion operation is compared to the time complexity analysis of the average-case retrieval operation. This means that the asymptotic time complexity functions ($t(N)$) of both operations are determined and compared. The graphs 29 and 30 show the scatter plot and application of the correlation coefficient (R^2) curve-fitting results, where the equations of the curves that have a $R^2 > 0.9$ are displayed and showed and will be used as the asymptotic time complexity functions from which the asymptotic efficiency classification of each operation is determined. Under each graph is the values of $t(n)$, R^2 and the Big-Oh upper-bound limit function $O(n)$, that $t(n)$ falls under.

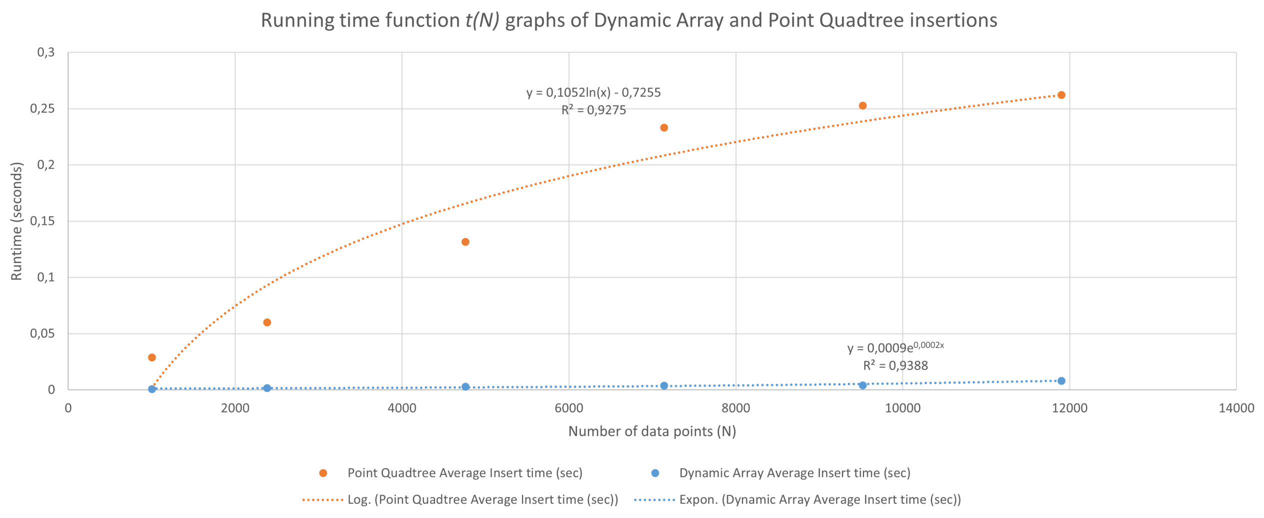


Figure 29: Running time function $t(N)$ graphs of Dynamic Array and Point Quadtree insertions

Point Quadtree average-case insertion

$$t(n) = 0.1052\ln(n) \ + 0.7255$$

$$R^2 = 0.9275$$

$$t(n) \in O(logn)$$

Dynamic Array average-case insertion

$$t(n) = 0.0009e^{0.0002n} \approx 1$$

$$R^2 = 0.9388$$

$$t(n) \in O(1)$$

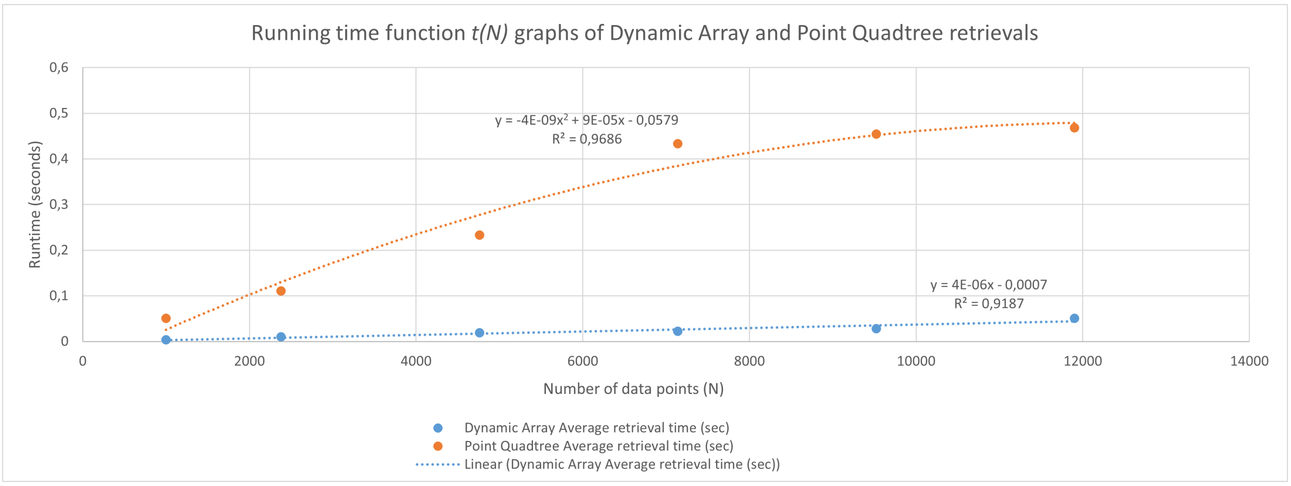


Figure 30: Running time function $t(N)$ graphs of Dynamic Array and Point Quadtree retrievals

Point Quadtree average-case retrieval

$$t(n) = (-4 \times 10^{-9})n^2 + (9 \times 10^{-5})n - 0.0579$$

$$R^2 = 0.9686$$

$$t(n) \in O(n^2)$$

Dynamic Array average-case retrieval

$$t(n) = (4 \times 10^{-6})n - 0.0007$$

$$R^2 = 0.9187$$

$$t(n) \in O(n)$$

5 CHAPTER FIVE: DISCUSSION, RECOMMENDATIONS AND CONCLUSION

5.1 Discussion

The objectives of this project aligned with answering the research questions brought forth in Chapter 1.4. How the results (Chapter 4) of this project answered these questions is discussed in this Chapter:

5.1.1 Question 1: What is the querying computational time complexity of Point Quadtree indexing?

The results of the experiment report that the Big-Oh time complexity class of the Point Quadtree range query retrieval operation in the average-case is $logarithmic \rightarrow t(n) \in O(n^2)$. The Point Quadtree time complexity of $O(n^2)$ is puzzling and unexpected because the documented and expected average-case of time complexity of a range query is $O(\sqrt{n}) = O(n^{1/2})$ (Lee and Wong, 1977). However, an interesting find is that this time complexity corresponds with the Big-Oh time complexity Brisaboa et al. (2017) found for the $k^2 - tree$, a variant of the k-dimensional data structure mentioned in Chapter 2.4.2 partitions raster data sets into 4 addresses just like the Point Quadtree, but does not partition them using the same set of algorithmic steps. This speaks to the preciseness of the implementation of the Point Quadtree that was used to conduct the experiments and determine the insertion computational time complexity of Point Quadtree in indexing. A similar experiment conducted by Kothuri et al. (2002) which compared the runtime vs dataset size of the Polygon-based Quadtree to the R-tree via Oracle Database commands also found that the range querying growth rates of the Quadtree is logarithmic, suggesting that the measurement techniques used to determine growth rate could impact the time complexity function determined, as mentioned in the Methodology 3, the drawback of using time as a time complexity determinant is that programming languages compile source codes differently and this might affect runtimes (Levitin, 2008, p. 44).

5.1.2 Question 2: What is the insertion computational time complexity of Point Quadtree indexing?

The results of the experiment report that the Big-Oh time complexity class of the Point Quadtree insertion operation in the average-case is $logarithmic \rightarrow t(n) \in O(logn)$. This is in coherence with the expected Big-Oh time complexity found by the originators of the Point Quadtree, Finkel and Bentley (1974) and stipulated in Chapter 2.4. This also corroborates with the findings of Kothuri et al. (2002).

5.1.3 Question 3: Does a Point Quadtree indexed data structure deliver spatial data more or less efficiently than non-spatial data structures?

From Graph 30 the time complexity functions of the Point Quadtree graph curve illustrate an $O(logn)$ complexity and the Dynamic Array graph curve illustrates an $O(1)$ complexity. Using the basic asymptotic efficiency classes Table(4) provided by Weiss (2012), it can be stated that the Dynamic Array overtrumps the Point Quadtree in terms of inserting vector point data, as a constant efficiency class of $O(1)$ loosely indicates that computational time resources taken by the Dynamic Array to insert data barely ever change, even if the dataset instance increases to large sizes, it is near impossible to use less efficient than that, even though the Point Quadtree's efficiency class is relatively impressive for inserting (Brisaboa et al., 2017; Kothuri et al., 2002). The Dynamic Array's extreme efficiency is not a surprise as it is a direct abstraction of the memory access model discussed n Chapter 2.2.2

From Graph 29 the time complexity functions of the Point Quadtree graph curve illustrate an $O(n^2)$ complexity and the Dynamic Array graph curve illustrates an $O(n)$ complexity. Weiss (2012, p. 189) lays it out that the linear class $O(n)$ of algorithms is the most practically efficient to implement in a computing system, whereas the quadratic class ($O(n^2)$) algorithms are seen as impractical for dataset sizes larger than a few thousand.

As the Dynamic Array is seen to be more efficient in both insertions and range query retrievals of vector point data, it can be stated that the Point Quadtree is less efficient in indexing vector point data than non-spatial data structures in terms of time complexity.

As expected and in coherence with algorithmic analysis literature and experiments such as Wei et al. (2014) and Brisaboa et al. (2017). The runtimes of both operations performed by both the Point Quadtree and the Dynamic Array increased as the data size increased.

The quality of the results of the experiment could have been improved by the following:

1. Using statistical methods such as least squares to fine-tune the determination of the time complexity functions. This could assist in improving the precision of the time complexity function before applying the Big-Oh notation efficiency classification.

2. As mentioned earlier, computational runtime is the easiest way to collect time complexity data, but not the most reliable. The operation counting method that counts the number of relational comparisons ($<, >, ==$) and mathemati-

cal operations $(+, -, \div, \times)$ in each data structure and compares them against dataset sizes is independent of machine and programming language compiler issues.

3. To make the experiment more "geospatial" in nature, a mapped visualization of the insertions and queries would have helped achieve this cause, however, this would not assist in answering the research questions posed by this project.

4. Using synthetic data as Wei et al. (2014) did, could of helped to control skewing in the distribution of the data.

5.2 Conclusion

In the conclusion, the project has helped to establish a basic demonstration of the time efficiency of the spatial indexing capabilities of the Point Quadtree data structure and point out that as a spatial data indexer it is less time efficient than a non-spatial data indexer, particularly the Dynamic Array. The project did not investigate whether the Dynamic Array maintained the geospatial properties of the vector data that was experimented upon, so it can not be conclusively stated that it securely and correctly inserted and retrieved the spatial data. The differences in the memory space complexities of the Dynamic Array and the Point Quadtree have not been compared as well. Given this, it can not be conclusively stated that in all respects of geospatial computing, the Dynamic Array is more efficient than the Point Quadtree, only with respect to time efficiency can it be concluded

5.3 Recommendations

These are the recommendations proposed based on the discussion and conclusions:

1. A carrying out of an investigative comparison between the Point Quadtree data structure and another spatial data structure would provide a more comprehensive and streamlined revelation of the efficiency of the time complexity of the Point Quadtree within the domain of the spatial sciences. In modern geospatial computing systems, the R-tree is considered the *de facto* standard in dealing with vector information and according to Brisaboa et al. (2017) and it is difficult to improve on its insertion, point and range querying efficiencies.

2. Considering that the Point Quadtree isn't particularly fast, it is recommended to avoid integrating them within time-sensitive applications like location-based services, but to rather limit their use to data integrity applications such as high concurrency geospatial databases updates, geospatial data store audits, geospatial batch processing and the likes (Brisaboa et al., 2017).

References

Ajwani, Deepak and Henning Meyerhenke (2010), *Chapter 5. Realistic Computer Models*, 194–236. Lecture Notes in Computer Science.

Bentley, Jon L, Donald F Stanat, and E Hollins Williams Jr (1977), "The complexity of finding fixed-radius near neighbors." *Information processing letters*, 6, 209–212.

Bentley, Jon Louis and Jerome H Friedman (1979), "Data structures for range searching." *ACM Computing Surveys (CSUR)*, 11, 397–409.

Bolstad, Paul J. (2016), *GIS fundamentals: A First Text on Geographic Information Systems*, 5th edition edition. Elder Press, White Bear Lake, Minnesota, USA, URL `http://link.springer.com/10.1007/978-3-319-01689-4_15`.

Bovet, Daniel Pierre and Pierluigi Crescenzi (1996), *Introduction to the theory of complexity*, nachdr. edition. Prentice Hall, New York [u.a.].

Brisaboa, Nieves R, Guillermo de Bernardo, Gilberto Gutiérrez, Miguel R Luaces, and José R Paramá (2017), "Efficiently querying vector and raster data." *The Computer Journal*, 60, 1395–1413.

Butler, Howard, Martin Daly, Allan Doyle, Sean Gillies, Stefan Hagen, Tim Schaub, and Erik" Wilde (2016), "The GeoJSON Format." RFC 7946, RFC Editor, URL `https://www.rfc-editor.org/rfc/pdfrfc/rfc7946.txt.pdf`.

Chen, Yiqun, Soheil Sabri, Abbas Rajabifard, and Muyiwa Elijah Agunbiade (2018), "An ontology-based spatial data harmonisation for urban analytics." *Computers, Environment and Urban Systems*, 72, 177–190.

City of Cape Town, GIS Corporate Division (2021), "Town survey marks." URL `https://hub.arcgis.com/datasets/4ee4fef293d74436afe31c2b979dfb30_14/about`.

Costa, Stefano, Damien Gaignon, and Luca Bianconi (2020), "Input formats." URL `https://totalopenstation.readthedocs.io/en/latest/input_formats/main.html`.

Croce, Antonello Ignazio, Giuseppe Musolino, Corrado Rindone, and Antonino Vitetta (2019), "Transport system models and big data: Zoning and graph building with traditional surveys, fcd and gis." *ISPRS International Journal of Geo-Information*, 8, 187.

Fazal, Shahab (2008), *GIS basics*, 100–264. New Age International.

Finkel, R. A. and J. L. Bentley (1974), “Quad trees a data structure for retrieval on composite keys.” *Acta informatica*, 4, 1–9.

Forlizzi, Luca, Ralf Hartmut Güting, Enrico Nardelli, and Markus Schneider (2000), “A data model and data structures for moving objects databases.” *ACM SIGMOD Record*, 29, 319–330.

Fredkin, Edward (1960), “Trie memory.” *Communications of the ACM*, 3, 490–499.

Gahegan, Mark N (1989), “An efficient use of quadtrees in a geographical information system.” *International Journal of Geographical Information System*, 3, 201–214.

Geography, GIS (2021), “Gis software.” URL `https://gisgeography.com/wp-content/uploads/2021/06/GIS-Software.png`.

Getis, Arthur (2010), “Spatial autocorrelation.” In *Handbook of applied spatial analysis*, 255–278, Springer.

Goldman, Sally A. and Kenneth J. Goldman (2007), *A Practical Guide to Data Structures and Algorithms using Java*. Chapman and Hall/CRC, London, URL `https://www.taylorfrancis.com/books/9781420010336`.

Goodchild, Michael F. (1992), “Geographical information science.” *International Journal of Geographical Information Systems*, 6, 31–45, URL `https://doi.org/10.1080/02693799208901893`.

Goodchild, Michael F. (2018), “Reimagining the history of gis.” *Annals of GIS*, 24, 1–8, URL `https://www.tandfonline.com/doi/abs/10.1080/19475683.2018.1424737`.

Guttman, Antonin (1984), “R-trees: A dynamic index structure for spatial searching.” In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, 47–57.

Kadochnikov, A, N Shaparev, A Tokarev, and O Yakubailik (2019), “Software tools for web mapping systems.” In *IOP Conference Series: Materials Science and Engineering*, volume 516, 012007, IOP Publishing.

Knuth, Donald E. (1973), “The art of computer programming.” *Sorting and Searching*, 3, Ch–6.

Kothuri, Ravi Kanth V, Siva Ravada, and Daniel Abugov (2002), “Quadtree and r-tree indexes in oracle spatial: a comparison using gis data.” In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, 546–557.

Kumar, Sameer and Kevin B Moore (2002), “The evolution of global positioning system (gps) technology.” *Journal of science Education and Technology*, 11, 59–80.

Lee, Der-Tsai and Chak-Kuen Wong (1977), “Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees.” *Acta Informatica*, 9, 23–29.

Levitin, Anany (2008), *Introduction to Design and Analysis of Algorithms, 2/E*. Pearson Education India.

Longley, Paul A, Michael F Goodchild, David J Maguire, and David W Rhind (2005), *Geographic information systems and science*. John Wiley & Sons.

Ma, Xiaolei and Yinhai Wang (2014), “Development of a data-driven platform for transit performance measures using smart card and gps data.” *Journal of Transportation Engineering*, 140, 04014063.

Mamoulis, Nikos (2011), “Spatial data management.” *Synthesis Lectures on Data Management*, 3, 1–149.

Manolopoulos, Yannis, Apostolos N Papadopoulos, Apostolos N Papadopoulos, and Yannis Theodoridis (2006), *R-Trees: Theory and Applications: Theory and Applications*. Springer Science & Business Media.

Nievergelt, Jürg, Hans Hinterberger, and Kenneth C Sevcik (1984), “The grid file: An adaptable, symmetric multikey file structure.” *ACM Transactions on Database Systems (TODS)*, 9, 38–71.

Patterson, David A and John L Hennessy (2016), *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann.

Samet, Hanan (1988), “An overview of quadtrees, octrees, and related hierarchical data structures.” *Theoretical Foundations of Computer Graphics and CAD*, F40, 51–68.

Samet, Hanan (1989), *The design and analysis of spatial data structures*, 1.print. edition. Addison-Wesley Publ. Co, Reading, Mass. [u.a.].

Samet, Hanan (1995), “Spatial data structures in modern database systems: The object model, interoperability, and beyond.”

Steenson, Rachel (2019), “The history of geographic information systems (gis).” URL `https://www.bcs.org/articles-opinion-and-research/the-history-of-geographic-information-systems-gis/`.

Sutherland, Ivan E, Robert F Sproull, and Robert A Schumacker (1974), "A characterization of ten hidden-surface algorithms." *ACM Computing Surveys (CSUR)*, 6, 1–55.

Tobler, Waldo R (1970), "A computer movie simulating urban growth in the detroit region." *Economic geography*, 46, 234–240.

van den Brink, Linda, Payam Barnaghi, Jeremy Tandy, Ghislain Atemezing, Rob Atkinson, Byron Cochrane, Yasmin Fathy, Raúl García Castro, Armin Haller, and Andreas Harth (2019), "Best practices for publishing, retrieving, and using spatial data on the web." *Semantic Web*, 10, 95–114.

Wei, Ling-Yin, Ya-Ting Hsu, Wen-Chih Peng, and Wang-Chien Lee (2014), "Indexing spatial data in cloud data managements." *Pervasive and Mobile Computing*, 15, 48–61.

Weiss, Mark Allen (2012), *Data structures and algorithm analysis in Java.* Pearson Education, Inc.

Wise, Stephen (2016), *GIS Fundamentals.* CRC Press.

Worboys, Michael F and Matt Duckham (2004), *GIS: a computing perspective.* CRC press.

Xiao, Ningchuan (2015), *GIS algorithms.* Sage.

Yang, ChongJun, Sheng Wu, YingChao Ren, Li Fu, FuQing Zhang, Gang Wang, Jian Tan, DongLin Liu, ChaoJi Ma, and Li Liang (2008), "Loose architecture of multi-level massive geospatial data based on virtual quadtree." *Science in China Series E: Technological Sciences*, 51, 114–123.

Zhang, Xiaoyi and Zhenghong Du (2017), "Spatial index." *Geographic Information Science &amp Technology Body of Knowledge*, 2017, URL `https://doi.org/10.22224%2Fgistbok%2F2017.4.12`.

Appendices

Appendix A

Listing 4: Python 3 Vector Point Class implementation.

```
from math import sqrt
class Point():
#A class for points in Cartesian coordinate systems.

        def __init__(self, x=None, y=None, itemID=None, timeStamp = None, GPSLastupdatedDateAndTime = None):
                self.x, self.y = x, y
                self.itemID = itemID
                self.timeStamp = timeStamp
                self.GPSLastupdatedDateAndTime = GPSLastupdatedDateAndTime

        def get_itemID(self):
                return self.itemID

        def get_timeStamp(self):
                return self.timeStamp

        def get_GPSLastupdatedDateAndTime(self):
                return self.GPSLastupdatedDateAndTime

        def __getitem__(self, i):
                if i==0: return self.x
                if i==1: return self.y
                return None
        def __len__(self):
                return 2
        def __eq__(self, other):
                if isinstance(other, Point):
                        return self.x==other.x and self.y==other.y
                return NotImplemented
        def __ne__(self, other):
                result = self.__eq__(other)
                if result is NotImplemented:
                        return result
                return not result
        def __lt__(self, other):
                if isinstance(other, Point):
                        if self.x<other.x:
                                return True
                        elif self.x==other.x and self.y<other.y:
                                return True
                        return False
                return NotImplemented
        def __gt__(self, other):
                if isinstance(other, Point):
                        if self.x>other.x:
                                return True
                        elif self.x==other.x and self.y>other.y:
                                return True
                        return False
                return NotImplemented
        def __ge__(self, other):
                if isinstance(other, Point):
                        if self > other or self == other:
                                return True
                        else:
                                return False
                        return False
                return NotImplemented
        def __le__(self, other):
                if isinstance(other, Point):
                        if self < other or self == other:
                                return True
                        else:
                                return False
                        return False
                return NotImplemented
```

```
    def __repr__(self):
        if type(self.x) is int and type(self.y) is int:
            return self.x, self.y
        else:
            return self.x, self.y

    def __str__(self):
        if type(self.x) is int and type(self.y) is int:
            return "({0:.10f},{1:.10f})".format(self.x,self.y)
        else:
            return "({0:.10f}, {1:.10f})".format(self.x,self.y)
    def distance(self, other):
        return sqrt((self.x-other.x)**2 + (self.y-other.y)**2)
```

Listing 5: Python 3 Dynamic Array Data structure class implementation

```
import pandas as pd
import time
from pandas_geojson import to_geojson
from pandas_geojson import write_geojson
from pandas_geojson import read_geojson
from point import *
from csv import writer
print("Packages have imported successfully")

def TSM_data():
    TSMsFrame = read_geojson('Town_Survey_Marks.geojson')
    TSMFeaturesList = TSMsFrame["features"]
    TSM_Upperlevel_List = pd.DataFrame(TSMFeaturesList)
    TSM_Upperlevel_Dict = pd.DataFrame.from_dict(
       TSM_Upperlevel_List)
    TSM_Properties = pd.json_normalize(
       TSM_Upperlevel_Dict['properties'])
    TSM_Properties = TSM_Properties.drop(columns=['Y_WGS_
       84','X_WGS_84', 'SRC', 'HGHT', "HGHT_SRC", "RMRK",
        "STS", "CTGY"])

    TSM_Geometry = pd.json_normalize(TSM_Upperlevel_Dict[
       'geometry'])
    TSM_Geometry = TSM_Geometry.drop(columns=['type'])

    TSM_df = pd.DataFrame()
    TSM_df['itemID'] = TSM_Properties["OBJECTID"]
    TSM_df['TSM Name'] = TSM_Properties["PNT"]
    TSM_df['coordinates'] = TSM_Geometry["coordinates"]
    TSM_df[['latitude','longitude']] = pd.DataFrame(
       TSM_df.coordinates.tolist(), index= TSM_df.index)
    timestmp = []
    for i in range(len(TSM_df.index)):
        timestmp.append(str(pd.Timestamp.now())[:19])
    TSM_df['Timestamp'] =  timestmp
    TSM_df = TSM_df.drop(columns=['coordinates'])
    return TSM_df

TSMlocs = TSM_data()
```

```
print(TSMlocs)
print()

if __name__ == '__main__':
    gjnum = 0
    while(gjnum < 100):
        DynamicArray = []
        data = TSM_data()

        # Convert each record into a Point object
        points = []
        for i in range(len(data)):
            points.append(Point(data.iloc[i,2], data.iloc
               [i,3], data.iloc[i,0], data.iloc[i,4],
               data.iloc[i,1]))


        # #insertions
        print("Inserting Points...")
        tick_insertion = time.perf_counter()
        for i in range(len(points)):
            DynamicArray.append(points[i]) #THE ACTUAL
               INSERTION CALL
        tock_insertion = time.perf_counter()
        print("Insertion Procss Done.")
        insertiontime = tock_insertion - tick_insertion
        print("Insertion time =", insertiontime, "seconds
           ")
        print()

        lons = []
        lats = []
        objectIDs = []
        timestamps = []

        #retrieval
        print("Retrieving All Points...")
        tick_retrieval = time.perf_counter()
        for p in range(len(DynamicArray)):
            if DynamicArray[p] is None:
                print("None") #THE ACTUAL QUERY CALL
            else:
                #print(DynamicArray[p], DynamicArray[p].get_busID())
                lons.append(DynamicArray[p][1])
                lats.append(DynamicArray[p][0])
                objectIDs.append(DynamicArray[p].get_itemID())
                timestamps.append(DynamicArray[p].get_timeStamp())


        tock_retrieval = time.perf_counter()
        print("All Points Retrieved.")
        retrievalime = tock_retrieval - tick_retrieval
        print("Retrieval time =", retrievalime, "seconds")
        print()

        # List that we want to add as a new row
        List = [gjnum, insertiontime, retrievalime]

        # Open our existing CSV file in append mode
        # Create a file object for this file
        with open(’TESTarrayoutputs.csv’, ’a’) as f_object:

            # Pass this file object to csv.writer()
            # and get a writer object
            writer_object = writer(f_object)

            # Pass the list as an argument into
            # the writerow()
            writer_object.writerow(List)

            # Close the file object
            f_object.close()

        gjnum = gjnum + 1s
```

Listing 6: Python 3 Point Quadtree Data structure class implementation

```python
import pandas as pd
import time
from pandas_geojson import to_geojson
from pandas_geojson import write_geojson
from pandas_geojson import read_geojson
from point import *
from csv import writer
print("Packages have imported successfully")

class PQuadTreeNode():
    def __init__(self, point, nw = None, ne = None, se =
       None, sw = None):
        self.point = point
        self.nw = nw
        self.ne = ne
        self.se = se
        self.sw = sw

def __repr__(self):
    return point.__repr__(self.point)

def is_leaf(self):
    return self.nw==None and self.ne==None and \
    self.se==None and self.sw==None


def search_pqtree(q, p, is_find_only=True):
    if q is None:
        return
    if q.point==p and is_find_only:
        return q
    dx,dy=0,0
    if p.x>q.point.x:
        dx=1
    if p.y>q.point.y:
        dy=1
    qnum = dx+dy*2
    child = [q.sw, q.se, q.nw, q.ne][qnum]
    if child is None:
        if not is_find_only:
            return q
        else: # q is not the point and no more to search
           for
            return
    return search_pqtree(child, p, is_find_only)

def insert_pqtree(q, p):
    n = search_pqtree(q, p, False)
    node = PQuadTreeNode(point=p)
    if p.x<n.point.x and p.y<n.point.y:
        n.sw = node
    elif p.x<n.point.x and p.y>=n.point.y:
        n.nw = node
    elif p.x>=n.point.x and p.y<n.point.y:
        n.se = node
    else:
        n.ne = node

def pointquadtree(data):
    root = PQuadTreeNode(point = data[0])
    count = 0
    for p in data[1:]:
        insert_pqtree(root, p)
    return root

def TSM_data():

    TSMsFrame = read_geojson('Town_Survey_Marks.geojson')
    TSMFeaturesList = TSMsFrame["features"]
    TSM_Upperlevel_List = pd.DataFrame(TSMFeaturesList)
    TSM_Upperlevel_Dict = pd.DataFrame.from_dict(
       TSM_Upperlevel_List)
    TSM_Properties = pd.json_normalize(
       TSM_Upperlevel_Dict['properties'])
    TSM_Properties = TSM_Properties.drop(columns=['Y_WGS_
       84','X_WGS_84', 'SRC', 'HGHT', "HGHT_SRC", "RMRK",
        "STS", "CTGY"])

    TSM_Geometry = pd.json_normalize(TSM_Upperlevel_Dict[
        'geometry'])
    TSM_Geometry = TSM_Geometry.drop(columns=['type'])

    TSM_df = pd.DataFrame()
    TSM_df['TSM ID'] = TSM_Properties["OBJECTID"]
    TSM_df['TSM Name'] = TSM_Properties["PNT"]
    TSM_df['coordinates'] = TSM_Geometry["coordinates"]
    TSM_df[['latitude','longitude']] = pd.DataFrame(
       TSM_df.coordinates.tolist(), index= TSM_df.index)
    timestmp = []
    for i in range(len(TSM_df.index)):
        timestmp.append(str(pd.Timestamp.now())[:19])
    TSM_df['Timestamp'] =  timestmp
    TSM_df = TSM_df.drop(columns=['coordinates'])
    return TSM_df

TSMlocs = TSM_data()
print(TSMlocs)
print()


if __name__ == '__main__':
    gjnum = 0
    while(gjnum < 100):
        data = TSM_data()

        # Convert each record into a Point object
        points = []
        for i in range(len(data)):
            points.append(Point(data.iloc[i,3], data.iloc
               [i,2], data.iloc[i,0], data.iloc[i,4],
               data.iloc[i,1]))

        # Insertions
        print("Inserting Points...")
        tick_insertion = time.perf_counter()
        q = pointquadtree(points) #THE INSERTION CALL
        tock_insertion = time.perf_counter()
        print("Insertion Procss Done.")
        insertiontime = tock_insertion - tick_insertion
        print("Insertion time =", insertiontime, "seconds")
        print()

        lons = []
        lats = []
        objectIDs = []
        timestamps = []

        #Retrieval
        print("Retrieving All Points...")
        tick_retrieval = time.perf_counter()
        for p in points:
            if search_pqtree(q, p) is None:
                pass
            else:
                location = search_pqtree(q, p)
                lons.append(location.point[1])
                lats.append(location.point[0])
                objectIDs.append(location.point.get_itemID())
                timestamps.append(location.point.get_timeStamp())
        tock_retrieval = time.perf_counter()
        print("All Points Retrieved.")
        retrievalime = tock_retrieval - tick_retrieval
        print("Retrieval time =", retrievalime, "seconds")

        # List that we want to add as a new row
        List = [gjnum, insertiontime, retrievalime]

        # Open our existing CSV file in append mode
        # Create a file object for this file
        with open('TESTquadtreeoutputs.csv', 'a') as f_object:

            # Pass this file object to csv.writer()
            # and get a writer object
            writer_object = writer(f_object)

            # Pass the list as an argument into
            # the writerow()
            writer_object.writerow(List)

            # Close the file object
            f_object.close()

        gjnum = gjnum + 1
```