Accelerating Multi-dimensional Search

Donald Whyte

BSc Computer Science (Industry)

2013/2014

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)_____

Summary

Many fields require or benefit from multi-dimensional search, including database querying, machine learning, computer vision, and visualisation. In multi-dimensional search, individual data items are represented as points, vectors or regions in n-dimensional space. These are arranged in an index structure so relevant data can be retrieved quickly, even when the volume of data is huge.

Different index structures are suited to different tasks. This report documents the findings of a project which explores the point query efficiency of several index structures for dynamic, high-dimensional datasets generated from scientific computation. Particular focus is given to two index structures: the pyramid tree and the point *kd*-tree.

Based on empirical findings, the project concludes with several hypotheses related to the performance of different classes of index structures when processing highly skewed datasets. Specifically, the pyramid tree and most dimension reduction or hash-based index structures are a poor fit for dynamic scientific datasets and other highly skewed data. It is hypothesised that tree-based index structures with adaptive decompositions, such as the point *kd*-tree, are better suited for point queries on these datasets.

Acknowledgements

First of all, I would like to thank my supervisor David Duke for his invaluable insights during our weekly meetings. His considerable experience in research helped me shape the project's overall evaluation. I would also like thank the project's assessor, Hamish Carr, for stepping in as supervisor while David was unavailable and equally providing useful insight on the project. Karim Djemame has been a fantastic personal tutor throughout the four years of my degree, and I would like to give him thanks for his continuous support.

Further acknowledgements go to Zhao Geng for his initial implementation of an index structure used in the project and fruitful discussion, as well as Elaine Duffin for her useful feedback on the mid-project and final report. I am grateful to my fellow students in the School of Computing for many useful discussions throughout the project, with special thanks to Pavol Klačanský.

Finally, I would like to thank my family and friends. All of these people provided a contribution at some point, whether they patiently listened when I started rambling about the project, or just helped me relax and forget about the project for much needed rest.

Contents

Chapter 1

Introduction

The real world is complex and multivariate, with high degrees of freedom. This is reflected in the complexity of the data it produces. Multi-dimensional search can be used to efficiently reason about this complex data.

Many fields require or benefit from multi-dimensional search, including database querying, machine learning, computer vision and visualisation. In multi-dimensional search, individual data items are represented as points, vectors or regions in n-dimensional space. These are arranged in an index structure so relevant data can be retrieved quickly, even when the volume of data is huge.

Since multi-dimensional search is a problem that appears in so many areas of computer science, developing fast, efficient ways of building and querying index structures has been the focus of many researchers over the past forty years. Dozens of multi-dimensional search structures have already been developed and there is still much ongoing work to develop even more efficient search structures.

1.1 Aim

This project will survey existing multi-dimensional search structures that attempt to solve the multi-dimensional search problem efficiently. The major challenges in the field, and how they have guided the development of structures throughout the past four decades, will be identified. The core aim of the project is to implement one or more index structures, optimising them specifically for high-dimensional datasets. These implementations will then be evaluated with respect to their proposed performance in the literature and pre-defined baselines, using chosen test datasets. This evaluation will highlight performance bottlenecks in the implementations and fundamental limitations of the algorithms themselves.

1.2 Objectives

The objectives of the project are:

1. Understand the current state and challenges of multi-dimensional search
2. Implement multi-dimensional search structures
3. Perform performance analysis on implementations and attempt to optimise structures for greater performance on high-dimensional data
4. Evaluate and compare performance of final structures to pre-defined baselines, unoptimised structures and their reported performance in the literature

1.3 Requirements

The minimum requirements are closely related to the objectives of the project. They are:

1. Produce literature review describing the current state of multi-dimensional search structures, comparing their strengths and weaknesses and highlighting core challenges of the field
2. Implement at least one index structure and perform performance analysis of the implementation
3. Perform one set of modifications to the implemented structure in an attempt to optimise its performance
4. Evaluate and compare performance of final implementations to pre-defined baselines, unoptimised structures and their reported performance in the literature

Possible extensions of the project include:

1. Parallelising implementations using technologies such as Haskell, CUDA (GPGPU) or MPI
2. Developing an entirely new index structure, which is also implemented and evaluated

1.4 Deliverables

The following will be delivered upon project completion:

1. Documented source code of implemented index structures
2. User manual describing how to use the index structures
3. Evaluation of the performance of the implemented index structures, with respect to pre-specified test data
4. Generated synthetic data that is used for the evaluation

1.5 Project Scope

A research group in the School of Computing at the University of Leeds have a particular interest in the applications of multi-dimensional search for scientific visualisation. This project will focus on *scientific visualisation* and not other applications, such as database indexing. Therefore, the performance of the implemented index structures will be evaluated primarily using high-dimensional scientific datasets with continuous domains, instead of discrete data pulled from databases.

Throughout the project, some assumptions have been made. These are used to narrow the scope of the project and allow more time to be spent focusing on the core aim of the project, which is high-dimensional scientific data. The assumptions are:

1. Datasets have a "high" number of dimensions (≥ 10), meaning the performance of the index structures will be measured using data with at least 10 dimensions. Data using less dimensions may be used to understand how the implementations behave with respect to dimensionality.
2. Datasets will be able to fit into the main memory (i.e. RAM) of the machine used for evaluation. That is, none of the data is paged to secondary memory, so page accesses causing reduced performance does not need to be considered.
3. Datasets are dynamic, meaning points may be inserted, deleted or updated at any time.
4. Structures only store *unique* points, meaning a structure cannot store two identical points

There are several ways of querying the data stored inside index structures. The relevant research group is interested in *point query* performance, so it was decided point queries will be the focus of the project.

1.6 Report Structure

Chapter 2 explains how the project was scheduled and the technologies used to produce the software deliverables. Chapter 3 reviews the existing work performed in this field and the core challenges which have driven the development of index structures throughout the past forty years. This leads into Chapter 4, which describes the index structures to be implemented and evaluated in detail, explaining why they were chosen.

Chapter 5 gives details on how the chosen index structures were implemented, documenting the low-level optimisation efforts performed to accelerate the structures. Chapter 6 covers what measurements and data will be used to evaluate the index structures' performance. Chapter 7 gives an evaluation of the implemented index structures using empirical performance timings.

Chapter 8 discusses the results from performance evaluation in greater detail, exploring the nature of scientific datasets and the suitability of different classes of structures for these types of datasets. Chapter 9 concludes the project by proposing future work to explore the hypotheses made in the previous chapter.

Chapter 2

Project Methodology

This section gives information on how the project will be tackled, describing the schedule and changes made to the initial schedule. The tools and technologies used throughout the project will also be discussed.

2.1 Schedule

The original schedule was devised on 31/01/14, near the end of the first week of the project. The schedule was broken down into the individual tasks required to complete the defined milestones. These tasks were then grouped into different **phases**, which mostly run in a serial fashion. Milestones were associated with these phases, each corresponding to the completion of specific deliverables.

On 20/02/14, this schedule was modified slightly to adapt to tasks taking longer than expected. The final project phases and their milestones are:

1. **Project Definition**
    - **Project Outline Defined** – outline of project's aims and objects is defined
2. **Literature Review and Data Collection**
    - **Literature Review Complete** – a literature review of multi-dimensional search has been produced
    - **Test Data Collected** – real test datasets have been decided and collected
3. **Mid-project Presentation and Report**
    - **Mid-Project Presentation Delivered** – mid-project presentation delivered to Computational Science and Engineering research group
    - **Mid-project Report Finished** – mid-project report submitted

4. **Design and Implementation** – iterative process that involves designing, implementing and optimising index structures as well as evaluating them to determine which is the fastest
    - **Software Deliverables Finished** – final, optimised implementations of the index structures have been produced, ready for a final evaluation
5. **Final Report Write-up**
    - **Final Report Finished** – final report has been submitted
6. **Student Symposium**
    - **Final Presentation Delivered** – final project presentation has been delivered

Figure D.2 in Appendix D shows a Gantt chart of the project phases, marking the start and end dates for each phase. Notice how some phases are running in parallel, with one phase being the primary focus and the other being a secondary focus where less time is spent. This is a research project by nature, so the exact tasks for most phases were not known in advance. Therefore, it was decided that a more detailed breakdown of the tasks throughout the project would not be created.

Figure D.5 shows a timeline marked with every milestone. The location of a milestone on the timeline marks the latest completion date of that milestone.

2.1.1 Iterative Design and Implementation

The performance tests and evaluation of the implementations could produce unexpected results, showing that it may be beneficial to implement an entirely different index structure or use a different technology. Therefore, deciding on a *fixed* set of index structures to implement before starting implementation would be unwise. With such an approach, there is no way to go backwards and revise the project plan if performance tests reveal an inefficient index structure.

A decision was made to choose a *single* index structure to implement before starting the Design and Implementation phase. This is evaluated and based on that evaluation, it is either optimised or discarded in favour of another index structure. This process was repeated in iterations, where each iteration contains the following sub-phases:

1. **Design** – plan what will be implemented in this iteration and produce design of implementation
2. **Build** – implement an index structure or perform a set of optimisations
3. **Test** – perform correctness tests on index structure to ensure it (still) works
4. **Performance Analysis** – perform performance analysis on index structure developed/optimised
5. **Evaluation** – evaluate the results of the analysis and use it to decide what to implement or optimise in the next iteration

Each iteration is a week long, which starts after with the weekly meetings with the project supervisor. This way, the supervisor gets a full summary of the last iteration and can give feedback on the plan for the next iteration.

When enough iterations have passed or there is no time left to spend on the Design and Implementation phase, the iterations will stop, with the evaluation of the optimised structure(s) in last phase being the final evaluation that is used in the Final Report Write-up phase. This iterative approach is illustrated in the full project process diagram shown in Figure D.3 in Appendix D.

2.1.2 Modifications to Original Plan

The plan has changed since its initial conception. The original plan stated that the initial, unoptimised implementations of the index structures should be finished before the mid-project presentation/report. By finishing the initial implementations of index structures before the presentation, an initial evaluation of the index structures could be performed. The results from this evaluation could be used in the presentation/report to justify decisions on what the remaining project will consist of.

However, due to the scope of the research field, the literature review took one more week than expected. This meant there was little time between the end of the literature review and the start of the mid-project presentation/report. Additional time was required to become fully informed about the field of multi-dimensional search and the nature of the data that will be used for evaluation. This knowledge informs the decisions on which index structures to implement, so it was felt that rushing into an initial implementation may result in a poor decision on which index structures to implement.

It was decided to delay any implementation until after the submission of the mid-project report. By doing this, there was more time to consolidate the research findings and make a more informed decision on the index structures to implement.

For completeness, the Gantt chart and milestone timeline for the original plan is shown in Figures D.1 and D.4 in Appendix D.

2.1.3 Splay Quadtree

The first index structure to implement was originally going to be the Splay Quadtree (Section 3.3.5.2). Due to the complexity of the structure, it was felt that there was not enough time to implement the structure and produce a comprehensive evaluation. It was abandoned in favour of the Pyramid Tree.

2.2 Technology

This section compares some of the potential technologies to use when implementing the index structures and analysing their performance. Justifications are given for the chosen technologies, referring to the project's goals and the experience of the project developer.

2.2.1 Programming Language

There exist many programming languages, all developed for specific purposes. Four programming languages have been considered for this project. This section will describe these languages, discuss

their differences and state which will be used for this project and why.

Python is a high-level interpreted language built for general-purpose computing, which has a large standard library and many third-party libraries [1]. **C** is a low-level systems programming language commonly used for high-performance applications [2]. **C++** is a middle-level, compiled systems programming language, combining low-level features (e.g. manual memory management) and high-level features (e.g. object-orientated programming) [3]. C/C++ applications are compiled straight to the native machine's CPU instruction set, meaning less time is spent translating the code to instructions the hardware can directly execute. **Haskell** is a purely functional general-purpose programming language which, like C/C++, compiles to native CPU instructions [4].

The following aspects have been considered when deciding which language to use:

- **Performance** – how fast the final application can run. This is based on a number of factors, such as how many intermediary layers there are between the code and the physical machine. Higher-level languages require the computer to do more work, because it has to convert the application's instructions into CPU instructions that can be executed directly on the processor.
- **Ease/Speed of Development** – this encompasses multiple aspects of a language. To facilitate fast development, the language must have good documentation and support, a mature tool set and be easy to understand.

Executing a Python statement introduces more overhead than C/C++ or Haskell statements. This is because Python is a high-level, interpreted language, where statements are compiled into bytecode at runtime. The bytecode is then run on a virtual machine, which translates the bytecode to native assembly instructions. Therefore, many Python applications run slower than C/C++ or Haskell applications, which are pre-compiled to native CPU instructions. However, due to the large amount of experience the developer has in Python, its large standard library and the higher number of abstractions, it is likely Python would result in the fastest development time.

Haskell's purely functional nature makes it fundamentally different than most popular languages and the developer of this project also has little experience writing programs in a purely functional style. It is predicted this will increase the time it takes to implement the structures. With such little experience in Haskell, it is unlikely well-writen, optimised code can be produced in such a short time. If parallel index structures are implemented, however, the lack of state and built-in support for parallelisation [5] would make Haskell a powerful choice.

Despite development speed being important, the primary aim of the project is the acceleration of search. Haskell can be used to write highly optimised code, especially when implementing parallel algorithms to utilise multiple CPU cores. C++ gives the developer more control over the computation and management of memory, allowing for low-level performance tuning. The trade-off for this control is the added complexity of the language [6], leading to an increased amount of code to write and the decrease in development speed that follows.

There is much debate on whether C produces faster code than C++ (e.g. [7–9]). This report will not make assertions about the relative performance of these two languages, but if C *is* generally faster

than C++ and the primary goal of the project is speed, then it seems logical to choose C.

In a project such as this, where many ideas will be implemented, tested and potentially thrown away, programmer productivity is still vital. The importance of this is amplified by the project's short duration. While the developer has experience using C, they are still significantly more experienced with C++. Therefore, as a compromise between the goal of high performance and the programmer's current skill set, C++ has been chosen as the language to use for developing the initial implementation of the index structure.

2.2.2 Development Tools

Various tools will be used throughout the development of the structures. **CMake** [10] will be used to automate the C++ build process. Using a build automation tool allows one to write high-level build scripts that are cross-platform, increasing the portability of the implemented index structures.

Unit tests will be written to ensure the implemented index structures and all the associated algorithms are functioning correctly. A unit test ensures a single unit of the code is exhibiting the desired behaviour. These tests are performed in isolation, preventing other code in the application from being executed and interfering with the tests' results [11]. A unit test framework makes the process of writing unit tests easier and faster [12]. C++ has many unit testing frameworks, such as CppUnit [13], NullUnit [14] and Google Test [15]. **Google Test** has been chosen for this project because of its wide feature set and comprehensive documentation, which is available at [15].

Version control is a way of tracking changes to textual documents, as well as *who* made the changes [16]. A VCS (version control system) will be used to track the changes of the implementation's source code, tests and built automation scripts. Despite this project only having a single developer, there are multiple reasons it has been decided to manage the source code using a VCS. Firstly, it allows the developer to easily keep a log of the changes made to any file and *why* those changes were made. One can also revert source files to prior versions if bugs are found or previously deleted code is required again, Finally, using a VCS means there will be multiple copies of the source code, which can be used as backups to mitigate the damage caused by data loss. The distributed VCS **Git** [17] will be used for this project, due to the developer having prior experience in the tool.

2.2.3 Performance Analysis Tools

In addition to considering the theoretical performance of the chosen algorithms, the implementations' performance will be tested using profiling. Profiling is a way of measuring the performance of a program or system [18] and is used to provide insight into which parts of the code take the longest or use the most memory (i.e. where the performance bottlenecks are).

A profiler is a tool which measures some performance metrics of a program. Since the core goal of this project is accelerating multi-dimensional search, being able to measure the performance of the implementations and identify where the performance bottlenecks are is incredibly useful.

To optimise the implementations, a profiler that can measure heap usage and cache misses will be used. A profiler which can produce call graphs with timings, that measure the flow of execution and how much time is spent in each function, is also desired. Using a profiler that produces no overhead is useful, but not critical, since timings could be scaled to take the extra overhead into consideration.

Many profilers were considered when choosing one for this project. Intel VTune [19] is feature-rich profiler for Intel-based machines, but it is proprietary. Valgrind [20], Profiny [21] and gperftools [22] are freely available profilers. Out of these, only Valgrind can measure cache misses. gperftools can generate visualisations of program flow and where the program spends most of its time using call graph images. gperftools provides a similar feature for heap profiling as well. Valgrind and gperftools will both be used to measure cache misses and generate call graphs.

2.2.4 SSE Optimisation

SIMD stands for Single Instruction, Multiple Data and was defined by Flynn as a classification of parallel computing [23]. In SIMD, a single instruction is applied to multiple data items at the same time. If p is the maximum number of data items that can be operated on in parallel at a time, then ideally a computation can be made p times faster. However, it is only suitable for computations where the same operation can be applied to multiple data items independently, so the order in which those operations complete does not affect the final output.

Streaming SIMD Extensions, or SSE, is a specification of an instruction set that performs SIMD operations on the widely used x86 CPU architecture [24], which is the architecture used for the development and test environment of this project. SSE will be used to accelerate frequently executed code where possible.

2.3 Conclusion

The project schedule follows a waterfall approach, with the end of one phase leading to the start of the next. To take unexpected change into account, flexibility is required. For this reason, one of these phases is composed of several weekly iterations.

C++ will be used to implement all index structures in the project. Both gperftools and Valgrind will be used to profile the implementations.

Chapter 3

Background Research

This chapter contains a literature review of multi-dimensional search. First, the problem will be formally defined. Then, the field's core challenges and their implications will be summarised. Existing multi-dimensional search techniques will be described, highlighting their limitations with respect to the core challenges. A discussion on how to choose a suitable structure for a problem will be given. Finally, the existing work on parallelising search will be summarised.

3.1 Formal Problem Definition

We define **data space** as the space containing all possible data items for some domain. Let d be the number of dimensions in this data space. The universe of the space is defined as $[min, max]^d$, where min is the minimum possible value and max is the maximum possible value, such that $min \leq max$. All values are real numbers. A data item is represented as a point $p \in [min, max]^d$. Figure 3.1 shows an example of a 2D space, where $d = 2$, $min = 0$ and $max = 1$. This means all possible data items within the data space are represented as a point in the universe $[0, 1]^2$.

A **multi-dimensional search structure**, often referred to as an **index structure**[1] [25], is a data structure which stores data items in a way which allows operations and queries to be performed quickly [26]. These structures can be **static** or **dynamic**. Static index structures are built for a specific set of points which never change. Dynamic index structures support changing data, and provide mechanisms for adding, deleting or updating (i.e. moving) a point. These structures typically have some form of balancing procedure [26] that maintains certain properties which allow for fast queries. The three core operations supported by dynamic index structures are:

- `insert` – insert a new point into the structure
- `delete` – delete a point already contained within the structure
- `update` – change location (i.e. one or more attributes) of an existing data item

[1] this will be used to refer to such structures throughout the rest of this document

Some dynamic structures have a **construction stage** when then structure is initially created, which provide fast ways of pre-loading data [27]. Otherwise, an index structure can be incrementally constructed by repeatedly using the `insert` operation.

The purpose of an index structure is to query the stored data. Common types of queries include:

- **Point Query** – checks if a point p exists in the structure [28]
- **Range Query** – return all points contained within a spatial region r [28], which is often a d-dimensional hyper-rectangle [27, 29, 30]
- ***k*-Nearest Neighbour Search** – return the k nearest points to a point p [30]
- **Approximate Queries** – approximation of a range or k-nearest neighbour query (faster queries) [31]

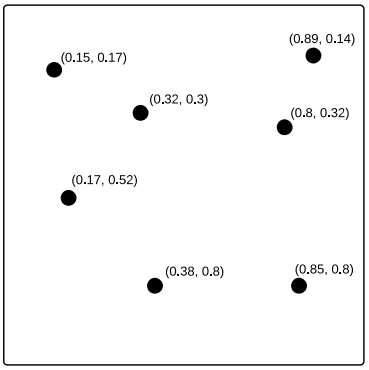


Figure 3.1: Data items represented as 2D points in the space $[0, 1]^d$ (not to scale)

An index structure provides a mechanism for accessing multi-dimensional data, but it may not actually store the data itself. In a database for example, a point query comes in two parts: **searching** for a data point in the structure and **retrieving** the data file represented by the point [32].

3.2 Challenges in Multi-dimensional Search

A significant number of index structures were developed in the 1970s and 1980s, which were shown to provide good performance through empirical usage. Despite the efficiencies gained from these foundational structures, there are still numerous challenges to overcome. As such, work on developing new index structures has continued throughout the past twenty years [33]. With the size and dimensionality of data increasing, some of these existing structures start to perform extremely poorly.

Four core challenges repeatedly discussed in the literature have been identified. Many attempts have been made to mitigate the effects of these challenges through new or modified index structures. This section will discuss these challenges, the impact they have on multi-dimensional search efficiency and *why* they have these impacts.

3.2.1 Curse of Dimensionality

The curse of dimensionality is a term used to refer to the issues that occur when data with large numbers of dimensions is processed [34]. Spatial partitioning of high-dimensional points becomes difficult because large regions of the data space are empty. High-dimensional space is sparse because the number of data points that must be sampled to fill the space increases exponentially with d.

Sparsity causes many index structures to contain many empty or near-empty regions, resulting in a large amount of memory overhead (e.g. many regions in octrees with high d are completely empty but exist anyway to due to uniformly sized regions). This is especially problematic when many of the dimensions might not tell you anything relevant about the data [35], causing large amounts of memory and computation to be wasted. Relating this back to multi-dimensional search specifically, Indyk and Motwani in [36] discuss how efficient nearest neighbour queries with a small number of dimensions is "well-solved", but with a high number of dimensions the problem is more challenging.

The majority of index structures discussed in the literature are hierarchical. Due to data sparseness caused by a higher number of dimensions, these structures often become very large and have many empty or near-empty nodes. Additionally, balanced splits and overlapping node regions cause most of the nodes to pass boundary intersection tests when a higher number of dimensions are used. This means the execution time of queries and dynamic operations tend to $O(n)$. In other words, the structures are no faster than a brute force sequential scan through a list, just with more memory overhead.

Weber, Schek and Blott in [37] show how hierarchical index structures which perform data space partitioning tend to perform poorly when 10 or more dimensions is used. They even show that there is no index structure "based on clustering or partitioning which does not degenerate to a sequential scan if the dimensionality exceeds a certain threshold" [37].

Therefore, for high d, sequential scan or methods based on linear data structures (e.g. lists) often provide better performance than methods which recursively decompose space into tree structures [33]. This leads to the conclusion that the curse of dimensionality removes many of the benefits gained from using traditional index structures when high dimensional data is used.

3.2.2 Variation in Data

When developing an index structure, one does not know the nature of the input data in advance. Therefore, it is unwise to make too many assumptions when evaluating the efficiency of your structure with test data. Some datasets may be uniform, some may have noticeable skew and others may have a large number of clusters with complex shapes in data space. Different index structures are more efficient with different kinds of data. For example, octrees become inefficient with skewed data and *kd*-tree variants often have poor utilisation of nodes [38] (i.e. nodes store very few data points).

For some applications, where the input data's structure may be known in advance, this is not an issue, since an index structure known to perform well with that type of data can be chosen. If the nature of the data is not known in advance however, developing a generic index structure that provides efficient queries for *all* kinds of data is challenging.

Wang et al. evaluated the PK-tree using uniformly distributed and clustered synthetic data, in addition to large real data sets [30]. Berchtold et al. also used a combination of carefully generated synthetic data and real data sets when evaluating the performance of Pyramid Tree [27]. Doing so allows for a comprehensive evaluation on how well a structure performs on different kinds of data. Comprehensively evaluating an index structure's performance requires a varied collection of data,

which includes:

- Uniformly distributed data
- Non-uniform data (skewed and clustered)
- Large real world data sets

Generating or finding these types of data *and* ensuring an index structure performs well on most inputs have been found to be significant challenges.

3.2.3 Dynamically Constructing Structures

To maintain performance, it is common to impose invariants that maintain a balanced structure. An invariant is a condition or property of the index structure that *must hold* to ensure good performance of queries. This is easier for static index structure because the structure is built once and never changes. When inserting, deleting or updating points with dynamic index structures, maintaining invariants can be non-trivial. An example of a dynamic index structure is the red-black tree, which repeatedly rotates components of the tree to fix broken invariants when a point is inserted or deleted [39].

These invariants can result in `insert`, `update` and `delete` operations being slow. Creating index structures that provide fast queries without requiring invariants that are expensive to maintain is difficult. The balance between dynamic construction performance and query performance depends on the application. If the index structure must be updated often, then prioritising fast construction over fast queries may be appropriate.

3.2.4 Memory Access and Paging

Different index structures have different storage requirements. It is important to consider this when choosing/developing an index structure. Structures whose size in memory grows quickly as the amount of data increases may not be able to fit entirely in main memory, meaning some of it will have to be *paged* in secondary memory (e.g. hard disk). This drastically affects performance, as it is much slower to access the hard disk than accessing main memory. This issue is amplified for high-dimensional data, as more memory is consumed per data item.

Some structures require little additional data to maintain the structure, such as splay trees and pyramid trees [27, 40]. These have *low memory overhead*, meaning they scale well to large data sets. Other structures, such as PK-trees, require more data because more complex book-keeping is required to enable fast dynamic operations and queries [30]. If a problem deals with massive amounts of data points, it may be worth using slower index structures with less overhead, in order to minimise the amount of memory used by the index structure. If the structure grows so large that it cannot fit in main memory, then *any* index structure's performance will take a drastic hit because data has to be retrieved from secondary memory frequently.

For large datasets (terabytes in size, say), it is inevitable that some of the index structure will be stored in secondary memory. This core bottleneck has been identified and researchers have con-

structed index structures, called bucket methods, that store data items in ways that minimise I/O operations [33]. Some structures have parameters that can be tweaked to find optimal paging with the target hardware or dataset, such as the PK-tree and pyramid tree [27, 30].

To conclude, if the index structure is used with large, complex data items then some of it may have to be stored in secondary storage, especially if there are many data points. Designing an index structure to facilitate quick access to secondary memory and reduce query times has been the focus of many researchers in the field [32, 41].

3.3 Existing Index Structures

Many index structures have been developed throughout the past forty years, such as quadtrees in the early 1970s [42] and splay quadtrees in 2012 [31]. New structures are often based on older index structures, either by modifying an existing structure or combining two different structures in some way. Figure D.6 in Appendix D shows a *taxonomy* of index structures. This section describes some of the widely used index structures, their limitations (with respect to the challenges discussed in Section 3.2) and why this led onto the development of new structures.

3.3.1 Basic Structures

Sequential scan refers to the brute-force approach of searching a data set. You simply store the dataset in a linear data structure (e.g. a list) and iterate through each data item until the desired point is found. This means searching is an $O(n)$ operation. Using a standard array, insertion is $O(1)$ if new points are inserted at the end of the array and deletion is $O(n)$. There is little memory overhead for this approach, but search times will be very long if there are large number of points. Nevertheless, if the number of data items is small, a more complex index structure may not be needed.

Search trees order data items in a hierarchical fashion to make search faster [39]. The **binary search tree**, for one-dimensional data, is one of the first of these trees and is the basis of many index structures [39]. Each node stores a *key*, which is a total orderable element. Each non-leaf node has at most two children, where the left child's key is less than its parent's and the right child's key is greater than or equal to its parent's. This property is referred to as *key order* [43]. When inserting or deleting new points, the key order invariant must be maintained. Maintaining this invariant can be done in $O(n)$ time [39], making insertion and deletion $O(n)$.

A point query is equivalent to checking if a certain key is stored in the tree. In the best case, we get running time $O(\log_2 n)$ for the query, because $\log_2 n$ is the *minimum* height of the tree. However, this performance is not guaranteed. Skewed or sorted data may result in taller trees, where there are many nodes with only left or only right children. This makes the binary tree *unbalanced* (height greater than $\log_2 n$). If balance is not guaranteed, the running time of a point query is $O(n)$ in the worst case, which is no better than a sequential search.

Balanced search trees re-order the nodes to maintain balance (logarithmic height) when a node is

inserted, updated or deleted, regardless of data skew. These operations cause dynamic index structure operations to become slower, but allow for faster queries. Maintaining balance is one-dimensional trees is well-solved, but much more difficult for multi-dimensional trees. Many one-dimensional balanced search trees exist, including red-black trees, AVL trees, splay trees and treaps [39].

3.3.2 Recursive Partitioning of Data Space

Index structures which decompose the data space into sub-spaces recursively are popular [33]. The decomposed space is represented as a search tree. If R is some d-dimensional data space, then $R_1, ..., R_x$ are the decomposed sub-spaces (often disjoint) of R . R becomes the root node and $R_1, ..., R_x$ become the children of R in the tree representation. Each of those sub-spaces may then be decomposed in a similar way, increasing the depth of the tree.

This is useful because it allows large amounts of the data space to be discarded from consideration at once. Consider a point query with a point p. $p \in R$ since R is the entire data space. Assuming a disjoint decomposition was used, if $p \in R_1$ then $p \notin R_2, ..., R_x$. Since it is known that p is only in R_1, only R_1's children need to be checked – the rest of the tree can be *ignored*.

3.3.2.1 Quadtrees

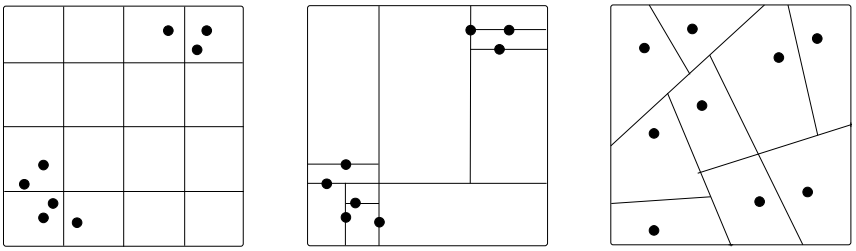
A quadtree is one of the earliest index structures based on disjoint recursive decomposition of space, specifically created for two-dimensional space [44]. **Octrees** are a generalisation of quadtrees to $d \geq 2$ dimensions. Throughout the past forty years, many variations of the quadtree have been produced. Hanan Samet produced a survey describing many uses and variants of this data structure in [42] and twenty years later, there are even more variants. Several variants are discussed in this review, but it is beyond the scope of this report to list all of them.

PR quadtrees are a commonly used quadtree which partition the data space into 2^d uniformly sized sub-spaces. This means quadtree nodes have up to 2^d children. This process can be repeated recursively to produce a grid with increasingly smaller cells. Figure 3.2a shows an example of a PR quadtree decomposition of some data. Notice how the underlying data space is decomposed in the same way, regardless of where the points in the dataset are in the space.

An advantage of PR quadtrees is that there is no overlap between the spatial regions represented by two sibling nodes. This greatly simplifies `insert`, `update` and `delete` operations. However, they do not perform well with **non-uniform** data. So if there are clusters or skews of data, then there will be many empty (or near empty) quadtree nodes/regions (see Figure 3.2a). These empty, unnecessary nodes may be searched during queries, reducing performance.

3.3.2.2 *kd*-trees

kd-trees are similar to quadtrees in that the data space is partitioned into disjoint regions [45]. Unlike quadtrees, nodes have at most two children regardless of dimensionality. Each node in the tree splits



(a) PR Quadtree (b) Point *kd*-Tree (c) Binary Space Partition

Figure 3.2: Spatial Decompositions Created With Various Tree-Based Index Structures

the current region of data space into two sub-regions along a single dimension i. These sub-regions are represented as two nodes, whose parent is the node representing the original region. A *pivot* value p is chosen for i. The left and right children contain all the data items whose ith value is less than p and greater than or equal to p respectively. How a *kd*-tree is built depends on how i and p are chosen, so there are many variations of this structure. There are two major types of *kd*-tree – point and bucket [33]. Point *kd*-trees store a single point in each node of the tree, meaning there are always n nodes. Bucket *kd*-trees only store points in the leaves, which can contain multiple points.

Unlike the PR quadtree, *kd*-trees can produce non-uniform partitions, making the structure better suited to skewed or clustered data. Figures 3.2a and 3.2b show a PR quadtree and point *kd*-tree partition of some clustered data. Notice how there are more empty regions in the quadtree than the *kd*-tree, thus having a less efficient partition of the data.

Like binary search trees, it is ideal to maintain a *balanced* tree to minimise height and ensure queries can be performed quickly. Dynamically inserting or deleting points can unbalance the tree. Variants of bucket *kd*-trees have been developed to maintain a balanced tree and fast query times. Examples of such variants include the BD-tree [46] and KDB-tree [47].

3.3.2.3 Binary Space Partitioning

BSP (binary space partitioning) trees are a generalisation of kd-trees that use hyperplanes (lines in 2D, planes in 3D) to recursively partition the data space [48]. That is, it partitions data space S into two disjoint sub-spaces S_1 and S_2. This process is repeated recursively using different hyperplanes for each split. A BSP tree decomposition is illustrated in Figure 3.2c.

Optimal BSP trees allow efficient point queries to be performed by discarding as many points as possible at each level of the tree (i.e. the tree has low height). Finding optimal BSP trees for a given set of data points is an $\mathbb{NP}$-complete problem [49]. When inserting, updating or deleting points a new optimal partitioning must be computed to minimise tree height, making dynamic operations very slow. Therefore, BSP trees are better suited for static data because an optimal tree can be *pre-computed* and loaded into main memory during program initialisation.

3.3.2.4 Skip Quadtree

Despite the original quadtree structure being developed approximately forty years ago, researchers are still enhancing the index structure's performance in new ways. A relatively new data structure, the skip quadtree, is "simple" to implement and benefits from low memory overhead [50]. **Skip quadtrees** were developed with range queries in mind and are based on the concept of compressed quadtrees. **Compressed quadtrees** compress paths of nodes which only have a single non-empty child into a *single node* [51], which combats the effects of skewed data.

Skip quadtrees combine one-dimensional skip lists [50] and compressed quadtrees to create an index structure with $O(\log n)$ point queries and $O(\varepsilon^{1-d} \log n + k)$ *approximate* range queries, where k is the size of the range and ε is the approximation factor that controls the accuracy of the range query [50]. However, skip quadtrees may not be efficient for very datasets because they were not constructed with paging in mind.

3.3.3 Bucket Methods

Bucket methods were developed to increase I/O efficiency for large datasets that are paged in secondary memory. Such structures were designed to minimise the number of I/O operations (secondary memory accesses) required to perform queries (see Section 3.2.4 for more discussion on this). These methods group points into contiguous blocks of memory called *buckets*, which are the same size as a page in secondary memory. To reduce the number of I/O operations required to perform a query, bucket methods aim to fill each bucket with as many nodes as possible [33].

Fanout refers to the amount of children each node have on average. Higher fanout results in a search tree with lower height, which means less nodes are accessed on average when searching for a point. Node access is particularly expensive when the tree has been paged into secondary memory, so some bucket methods aim to increase fanout (e.g. TV-tree [52] and A-tree [53]).

There are two types of bucket methods [33]:

- **Overlapping Decomposition** – these ensure that each bucket has a *minimum* number of data items it must contain, which increases fanout and thus, search times. However, in the process of ensuring each bucket has a minimum amount of used capacity, the bounding regions of buckets in data space may begin overlapping. Overlapping regions means more nodes must be checked when performing a query, increasing search times.
- **Disjoint Decomposition** – these ensure that there is *no overlap* between buckets. This means less nodes need to be checked in a query, but there is no guarantee on how much each bucket is filled, potentially increasing the number of I/O operations performed

3.3.3.1 B-Trees and B$^+$-Trees

The B-Tree was developed by Rudolf Bayer in 1972 and is often used for databases or file-systems [54]. It is a one-dimensional search tree which allows for point queries, insertions and deletions to be

performed in $O(\log n)$ time [55]. B-trees maintain key order, but allow nodes to have more than two children, making them a generalisation of binary search trees.

B^+-trees are B-trees where only the *leaves* contain the actual values of the data [54]. This means all non-leaves simply contain pointers to, or the keys of, their children. B^+-trees ensure that each bucket is at least 50% full [33, 54], which leads to less I/O operations. This is a very attractive property, as it can greatly speed up query times for large data sets. There have been many attempts to generalise the B-tree to a multi-dimensional setting while retaining this property. The k-d-B tree [47] and BV-tree [56] are two examples of such attempts. Freeston in [56] discusses how this "apparently simple" objective has proved extremely difficult to achieve.

3.3.3.2 R-Tree Family

An R-tree can be thought of as a B^+-tree that can handle multi-dimensional data, supporting data items that have non-zero size in data space (i.e. regions) [29]. This means nodes are represented using *hyper-rectangles* that define the region of the data space all of the node's children are contained in. The running time for queries is $O(n)$ in the worst case, but expected performance is much higher.

R-trees have attracted much attention from researchers. Many variants of the structure have been developed, such as R*-trees [57], R+-trees [28], SS-trees [58], RS-trees [41] and Hilbert R-Trees [59]. Combinations of these variations also exist, such as SR-trees [33] and RSR-trees [32].

R-tree based structures use overlapping decomposition, which is a key performance issue [27]. If you have a point contained in m nodes' regions, all m nodes could be searched. This severely limits the performance of R-tree based index structures with data spaces that have a **large** number of dimensions (see Section 3.2.1 for more details).

3.3.3.3 PK-Tree

PK-trees (Pyramid K-instantiable trees) are a family of structures based on kd-trees that were created specifically to handle high-dimensional data [30]. They are similar to bucket methods, but instead of imposing a minimum number of points per bucket, they impose a maximum. Imposing this maximum (called the k-instantiation value) results in a reduced amount of I/O operations [33]. PK-trees bound the height of the tree to $O(\log n)$ for some data. Through tests on synthetic and real data, the PK-tree has been shown in greatly outperform the SR-tree and X-tree [30], especially for higher dimensions.

However, there are some caveats. In order for the PK-tree to have $O(\log n)$ height, certain constraints must be applied to the distribution of the data. Furthermore, the pagination of the index structure which minimises I/O operations depends on the value of K (the number of children a node can have) and the size of the actual hard disk pages used by the operating system [30]. The amount to split each dimension at each level is also configurable, so there are many parameters to tweak to achieve the proposed performance. Additionally, as discussed in Section 3.2.1, the PK tree's performance on very high dimensional data will be still be poor, because most index structures that decompose the underlying data space perform poorly when d is high.

PK-trees are also complicated to implement. This problem is exacerbated by having to tweak many different parameters and be careful about what kind of data is stored in the tree. Therefore, due to their complexity and poor performance on very high dimensional spaces, PK-trees have not been widely adopted [33].

3.3.4 Structures Tailored to High-Dimensional Data

Section 3.2.1 discusses how higher dimensional data spaces cause index structures which perform well on low dimensional data to degenerate and provide poor performance. There have been efforts to develop index structures which still perform well in higher-dimensional space. Many different approaches have been used, such as tree-based spatial decompositions, distance-based methods, dimension reduction and non-hierarchical, sequential methods. Some of the more influential data structures from the literature have been identified and are described here.

3.3.4.1 X-Tree

R-tree based index structures often struggle to provide efficient queries in high-dimensional space, because the hyper-rectangles or spheres tend to overlap more [27]. X-trees [60] try to reduce overlap by extending the capacity of nodes/buckets if creating a new node would result in an overlap (these are called supernodes). X-trees outperform most R-tree variants [60]. However, Berchtold et al. show in [27] show the structure still degenerates at higher dimensions ($d \geq 10$) [27].

3.3.4.2 Distance-Based Methods

When there are a large number of dimensions, determining which dimensions are actually relevant for searching can be difficult. In these cases, *distance-based methods* may be useful. They use the similarity (distance in data space) between each pair of points to deduce more information about the relationships in the data and provide faster search [33]. There are two major types: *pivot-based*, which choose a subset of all points in the dataset to base distance measurements on, and *cluster-based*, which partition the dataset's points into spatial regions called clusters that are used to compute distances [33]. A notable distance-based index structure is the M-tree, which combines R-trees with distance-based techniques to perform high dimensional search with dynamic datasets [61].

3.3.4.3 Dimension Reduction

One method of mitigating the effects of the curse of dimensionality is reducing the dimensionality of the dataset and then using an existing structure which is known to have good performance with low dimensional data.

A classic example of dimension reduction is principal component analysis (PCA) [62] and an equivalent technique singular value decomposition (SVD) [33]. PCA transforms a data space into one with less dimensions, using correlations in the data to deduce which dimensions are the most useful

for discrimination [62]. However, by reducing the dimensions using PCA some information is lost and the results of queries will not be exact.

PCA struggles with dynamic data; if the data changes then the computed transformation will go out of sync and queries will be even less accurate. The transformation could be re-computed whenever a point is added, removed or changed but doing so takes a very long time, making dynamic operations inefficient. Therefore, these dimension reducing techniques are suitable to static datasets where full accuracy is not required.

3.3.4.4 Pyramid Tree

The pyramid tree is specifically targeted towards high-dimensional data [27]. It partitions the data space into $2d$ pyramids and map d-dimensional data items to *one-dimensional space*, storing the mapped 1D points in a B^+-tree. This one-dimensional representation is achieved by describing a data point in terms of *which* pyramid it is contained in and *where* in the specified pyramid it is. d-dimensional points are reduced to one dimension before querying, reducing the effects of the curse of dimensionality. Even though points are reduced to one-dimension, no data is lost because the original d-dimensional point is stored alongside its 1D representation. Hence, pyramid trees reduce the dimensionality of the data but retain all of the original information, unlike PCA.

B^+-trees provide efficient `insert`, `update` and `delete` operations [54], meaning the corresponding operations are also efficient with pyramid trees. The pyramid tree's range query performance, *relative* to other index structures such as X-trees, increases as d does [27]. This structure has low memory overhead because, in addition to B^+-tree overhead, one extra value is stored per point.

Since the pyramid tree mitigates the curse of dimensionality, provides dynamic operations, uses a bucket method to store the 1D points and handles skewed data (using the Extended Pyramid Technique [27]), the structure considers all four challenges discussed in Section 3.2. Tests performed by Berchtold et al. show that the Pyramid Tree is successful at tackling these challenges [27].

3.3.4.5 Embedding Methods

Embedding methods are combinations of dimension reduction and distance-based methods [33]. They use *approximated* distances between points in reduced space to perform *exact* queries. An example of an embedding method is FastMap [63].

3.3.4.6 Non-Hierarchical Methods

Focus is turning towards sequential scan and linear data structures for faster search with high-dimensional data [33, 37]. This is because hierarchical index structures based on data space partitioning perform poorly on a higher number of dimensions (see Section 3.2.1). Furthermore, for large datasets that must be stored in secondary memory, sequential scan may also outperform hierarchical methods, because data items are stored contiguously and read sequentially, thus requiring less I/O op-

erations. The VA-file is a method based on sequential scan, which is shown to consistently outperform sequential scan and the X-tree as the number of dimensions increase on real datasets [37].

Another non-hierarchical index structure that has been used are hash maps [33]. One approach is to define one primary bucket in the hash map for each grid cell (region of data space), which can hold x points. If a primary bucket has more than x points, then an overflow bucket is constructed. This bucket is linked to the primary bucket it spawned from, similar to the chaining conflict resolution mechanism for 1D hash tables [39]. Notable hashing techniques for multi-dimensional search include MDEH (multi-dimensional extended hashing) and PLOP (piecewise linear order preserving) hashing [33].

3.3.5 History-sensitive Structures

Given the same data twice, most structures will behave exactly the same. That is, how the structure builds itself and access times have little, if any, variation. There are some structures which are **history-sensitive**, meaning that their behaviour is either non-deterministic, involving some element of randomness that affects how the structure performs over time, or self-adjusting, changing itself based on how the stored data is accessed. These can allow structures to be much more dynamic and adjust themselves to perform optimal search based on the application it is being used in. Two of these structures, the quadtreap and splay quadtree, are described here.

3.3.5.1 Quadtreap

A treap is a randomised search tree used to store one-dimensional keys, which combines a binary search tree with a heap by maintaining both *key-order* and *heap-order* (nodes ordered by the randomly assigned probabilities) [64]. The main advantage of the quadtreap is that $\log n$ height can be maintained, even when dynamically inserting and removing points, with "high probability" [64].

A quadtreap is a combination of a compressed quadtree and a treap, which uses tree rotations to maintain balance [64]. Let h be the height of the tree. Running times for `insert` and point queries are $O(h)$, and `delete` is performed in $O(h^2)$. This means the *expected* times for `insert` and `delete` are $O(\log n)$ and $O(\log^2 n)$ respectively.

3.3.5.2 Splay Quadtree

Splay trees are one-dimensional index structures which make use of a splaying operation to achieve fast dynamic operations [39]. A splay tree is *self-adjusting*, which means it is "a data structure that reorganises itself to fit the pattern of access." [31] This is achieved with the splaying operation, which performs a series of $O(1)$ tree rotations to maintain a balanced tree with low height.

Park and Mount, the creators of the quadtreap [64], stated "there is no comparable self-adjusting data structure for storing multi-dimensional point sets" with regard to splay trees [31]. Hence, they developed the splay quadtree, which combines BBD-trees (balanced box decomposition trees) with quadtrees by making use of an equivalent splaying operation on BBD-trees. While the structure is

complicated to implement, Park and Mount proved good bounds on the running time of dynamic operations and queries.

At the time of writing, no papers empirically evaluating its performance have been published. Currently, the splay quadtree remains a purely theoretical structure.

3.4 Evaluating and Choosing an Index Structure

3.4.1 Measuring Efficiency

One can measure the efficiency of an index structure by simply measuring the amount of time it takes perform dynamic operations and queries [26]. In addition to execution time, one must also consider how much memory overhead is produced by the index structure. Such overhead may not be an issue for small data sets, but when your index structure becomes very large, it must be considered due to the issues discussed in Section 3.2.4. Measurements of index structure efficiency include:

- **Structure Size** – memory required to store the structure, relative to dataset size
- **Construction Time** – how long it takes to construct a structure storing n points
- **Dynamic Operation Execution Time** – execution times of `insert` and `delete` operations
- **Query Execution Time** – execution times of point, range or nearest neighbour queries

Big-Oh notation [65] refers to the worse case execution time of an algorithm, often with respect to the size of the input. This is used to bound the worst, average or expected execution time of an algorithm. This can be used to construct theoretical performance measurements for an index structure and if often used to guide the development and evaluation of index structures structures (e.g. in [31]).

The focus of bucket methods is to minimise I/O operations while still retaining well-balanced search trees. Therefore, in addition to the runtime of each operation, the number of I/O operations, or **page accesses**, is often measured (e.g. [27, 30, 60]). In conjunction with CPU processing time, page access count can give good insight into how efficient an index structure is for large datasets.

3.4.2 Deciding on an Index Structure

While some structures are generally less efficient than others, such as the R-tree when compared to X-tree, the behaviour of index structures typically depends on the input data. In other words, there is no "best" structure. Choosing the best index structure is *task-dependent*. There are many factors to consider when choosing an index structure, some of which are listed below.

- how many points will the structure contain?
- how many dimensions does the data have?
- how frequently will the data be modified, if at all?
- do queries have to be performed near real-time or is it acceptable to wait a few minutes?
- what distributions will the input data have? will they be uniform or highly skewed?

Note that this is by no means an exhaustive list. Table D.1 in Appendix D lists some of the index structures discussed in this review and their strengths and weaknesses.

The strengths and weaknesses of the structures discussed in the review will now be summarised. B^+-trees are very popular for one-dimensional data because they are fast, simple and have low memory overhead [54]. Quadtrees, kd-trees and R-tree variants have good performance on multi-dimensional data (e.g. for geometric optimisation of a 3D scene [66]) and are simple to implement (no complex invariants to maintain). However, performance starts to decrease as you increase the number of dimensions. There is a need for structures which can perform efficient queries in high-dimensional space, so structures such as the pyramid tree and PK-tree were developed [27, 30]. For even higher dimensions (e.g. $d \geq 10$), it has been shown that non-hierarchical methods, such has hash-based structures or sequential scan variants, may provide better performance.

3.5 Parallel Search

When it is desired to increase the efficiency of some computational task, it is common to consider parallelisation. In the context of multi-dimensional search, this means we want to determine whether or not it is possible to perform the dynamic operations and queries of an index structure in parallel, so that we reduce runtime by solving multiple parts of the problem simultaneously.

Multi-core parallelisation runs tasks in parallel on different CPU cores, which can be achieved by executing a program on multiple processes or threads on the host operating system. While it is possible to parallelise index structures on multi-core processors, prior research appears to focus on many-core (e.g. [67–72]) and distributed (e.g. [73–75]) parallelisation. *Many-core* parallelisation involves a much larger number of cores than multi-core processors, achieving higher parallelisation. GPUs (graphics processing units) are examples of many-core processors, which have thousands of cores. GPUs were initially developed for real-time graphics, but they are increasingly being used for general-purpose computing (GPGPU) [76]. *Distributed computing* uses multiple physical machines, which communicate with each other over a network, to parallelise computation [77].

Embarrassingly parallel is a term used to describe problems which can be easily split up into independent tasks that can be parallelised [78]. Queries using sequential scan can be considered embarrassingly parallel, as the linear structure can be partitioned and allocated to multiple processors which each search their own sub-structure [79]. Many index structures are non-linear, hierarchical structures that use some form of tree. This can make them difficult to parallelise, especially if queries require backtracking (traversing back up the tree to take another path). This makes the parallelisation of many search structures incredibly difficult and for some, the level of communication that would be required between each parallel process is so high it dominates the time savings achieved by parallelisation, making it less efficient than a serial implementation.

A good amount of research has been performed on parallelising one-dimensional search structures, especially the B-tree [67–69, 73, 80, 81]. There has also been research into parallel multi-dimensional

structures, such as the KDB-tree [70] and R-trees [71, 72, 82, 83]. Recent years have seen much focus on running search on the GPU. Both the B-tree and R-tree have been successfully parallelised on the GPU and have achieved increased performance [68, 71]. However, these techniques are complex and difficult to implement. It appears that parallelising multi-dimensional, even for small gains in efficiency, requires significant effort.

Furthermore, most of the multi-dimensional index structures which have currently been parallelised in the literature (e.g. R-tree) are known to degenerate when given high dimensional data (see Section 3.2.1). Since this project's focus is high-dimensional data and many of the existing parallel techniques are difficult to implement, especially for a developer with little experience with parallel or GPGPU programming, it has been decided that parallelisation will not be the focus of this project.

3.6 Conclusion

In this review, the multi-dimensional search problem has been defined. Sequential scan could not handle the volumes of data required and was deemed an inefficient solution, resulting in a huge collection of index structures being developed throughout the past forty years.

Each index structure has its own advantages and disadvantages because they were developed to tackle specific challenges. While some structures generally outperform others, there is no "best" structure. When considering which to use for a given application, it is important consider some key aspects about the data and the application it is being used in (see Section 3.4.2). The answers to these questions can help guide the decision on which index structure to use.

The one-dimensional search problem is generally well solved and fast, dynamic index structures capable of storing huge amounts of data exist. For a low number of dimensions (e.g. 2 or 3), research has focused more on developing simple index structures with lightweight memory requirements. For high-dimensional data, focus appears to be shifting away from tree-based structures that recursively decompose space and towards linear or hash-based structures. This is because tree-based approaches have been shown to have limited, if no, performance gain over sequential-based approaches when dealing with large numbers of dimensions [33]. There has been research into parallelising search, but it remains a difficult problem, with little focus being given to parallelising high-dimensional search specifically.

Chapter 4

Chosen Index Structures

Six index structures will be implemented and evaluated in this report. These structures are sequential scan, PR octree, Pseudo-Pyramid Tree, Pyramid Tree, Bit Hash and the point *kd*-tree. This section will describe these structures in detail and discuss the reasons for choosing them.

4.1 Baselines

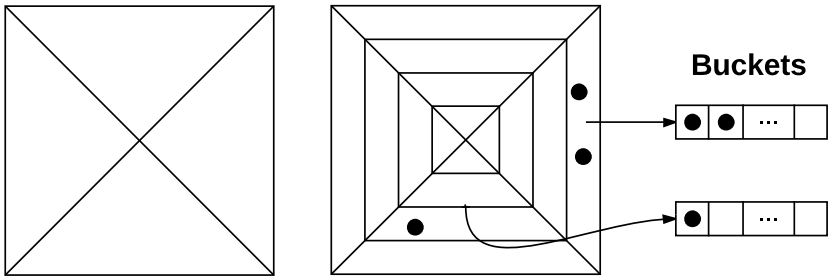
There is little use in measuring the performance of the implemented index structures if there is nothing to compare the results to. Two baseline structures have been chosen: sequential scan and the PR octree (see Sections 3.3.1 and 3.3.2 respectively).

Sequential scan has been chosen as a baseline because it is the naïve brute-force approach to search. The PR octree was chosen because it is the basis of many index structures, both old and new. Therefore, both of these structures are well-known in the field, making them suitable baselines.

4.2 Pyramid Tree

As described in Section 3.3.4, the pyramid tree reduces multi-dimensional points to one dimension, which can then be used in a one-dimensional index structure. The reason this structure has been chosen is because dimension reduction techniques have been found to perform well for high-dimensional data, compared to tree-based approaches [33]. The original Pyramid-Tree paper showed good performance on uniformly distributed data and a real dataset (with the Extended Pyramid-Technique), when compared to Sequential Scan and the competitive X-tree [27]. There are two fundamental differences between the focus of the paper's analysis and what this project is trying to achieve:

1. the paper's analysis is for *range* queries, whereas this project is focused on point queries
2. the real dataset, "a large text database extracted from WWW-pages" [27], contains discrete values, whereas the real datasets considered in this project are continuous, scientific datasets



(a) $2d$ Pyramids in 2D Data Space

(b) Segmented $2d$ Pyramids, Each Associated with a Bucket

Figure 4.1: Pyramid Tree's Decomposition of Underlying Data Space

While the database is mapped to a spatial domain when stored in an index structure, the data may have different properties or distributions than continuous, spatial-based data. Scientific datasets, the target of this project, tend to model continuous domains. Therefore, it was decided that the Pyramid Tree will be implemented to explore if the structure can provide equally good performance with point queries on dynamic scientific datasets.

The dimension reduction technique will now be explained in further detail. The Pyramid tree partitions the data space into $2d$ Pyramids, where each pyramid uses a $(d-1)$ hyperplane for its base and the centre of the data space as its tip. Figure 4.1a shows how the $2d$ pyramids extend from the centre point to the data space's boundaries. Each pyramid is segmented by splitting them along $(d-1)$ hyperplanes parallel to the base of the pyramid.

The one-dimensional value of a point, called the *pyramid value*, defines which pyramid it is in and its height in said pyramid. The resultant spatial partition is shown in Figure 4.1b, where each pyramid segment has a bucket that stores the points inside that segment.

More formally, the pyramid value pv_v of a point v is given by $pv_v = (i + h_v)$, where i represents the pyramid v is contained in, and h_v is height of v in pyramid i. i and h_v are given in Equations 4.1 and 4.2 respectively, where MOD refers to the modulo operator.

$$i = \begin{cases} j_{max}, & \text{if } v_{j_{max}} < 0.5 \\ j_{max} + d, & \text{if } v_{j_{max}} \geq 0.5 \end{cases} \tag{4.1}$$

$$j_{max} = (j \mid \forall k \, 0 \leq j,k \leq d, j \neq k : \ |0.5 - v_j| \geq |0.5 - v_k|)$$

$$h_v = |0.5 - v_{i \, MOD \, d}| \tag{4.2}$$

The following lemma shows that two distinct points can be mapped to the same pyramid value. It follows that multiple points may be stored in the same bucket. As such, bucket size will be a key performance factor for the structure and will be measured alongside execution time.

Lemma: *For $d \geq 2$, there exist two points x and y, such that $x \neq y$, with the same pyramid value.*
Let d be the number of dimensions and $x = (x_0, x_1, \ldots, x_{d-1})$, $y = (y_0, y_1, \ldots, y_{d-1})$ be two d-dimensional points. Without loss of generality, assume $0 \leq x_i \leq 1$ and $0 \leq y_i \leq 1$ for all $i = 0, 1, \ldots, d-1$. Suppose $x_0 = y_0 = 0$, $x_{d-1} = 0.1$, $y_{d-1} = 0.2$ and $x_i = y_i = 0.1$ for all $i = 1, \ldots, d-2$. Clearly, $x \neq y$. The following holds:

1. $\forall k\ 0 \leq k \leq d, k \neq 0 :\ |0.5 - x_0| \geq |0.5 - x_k|$,
2. $\forall k\ 0 \leq k \leq d, k \neq 0 :\ |0.5 - y_0| \geq |0.5 - y_k|$.

Since $x_0 \leq 0.5$ and $y_0 \leq 0.5$, both x and y are mapped to pyramid 0, so $i = 0$. It follows that $h_x = h_y = |0.5 - v_{0\,MOD\,d}| = |0.5 - v_0| = |0.5 - 0| = 0.5$. $pv_x = pv_y = 0 + 0.5 = 0.5$. But $x \neq y$, meaning two distinct points can have the same Pyramid value.

□

4.3 Pseudo-Pyramid Tree

University of Leeds' School of Computing have an implementation of an index structure which is similar to the Pyramid Tree. Like the Pyramid Tree, it reduces d-dimensional points to a single dimension, which is used to search for the original point in a one-dimensional search structure. Each 1D key has its own bucket that contains references to the original points which are mapped to it. This index structure will be called the **Pseudo-Pyramid Tree**, because the reduction technique used is *inspired* by the Pyramid tree, but is not the same. To gain further insight into the performance of dimension reduction techniques for dynamic scientific data, the Pseudo-Pyramid Tree will be implemented.

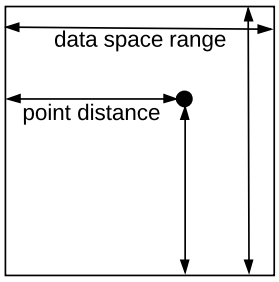


Figure 4.2: Distance of Point From Data Space Boundaries

The Pseudo-Pyramid Tree reduction will now be given. To reduce a point p, how far along the boundary the point is in each dimension is determined (see Figure 4.2 for an illustration). These distances are computed using Equation 4.3, where min_i and max_i define the minimum and maximum bounds for dimension i respectively.

$$h_i(p) = \frac{p_i - min_i}{max_i - min_i} \tag{4.3}$$

Let B be a parameter which controls how likely points will be hashed to the same value and stored in the same bucket. Let $M = \lceil B^{\frac{1}{d}} \rceil$. The hashed value $h(p)$ of a point p is then given by Equation 4.4. The function takes $O(d)$ time to compute because of the summation.

$$h(p) = \sum_{i=0}^{d} \lfloor h_i(p) \times M^i \rfloor \tag{4.4}$$

The coefficient M is used to increase the magnitude of $h_i(p)$ before it is truncated with the floor function. This spaces out the mappings of points, decreasing the likelihood two distinct points will be reduced to the same value (and be stored in the same bucket). Increasing B will increase M because M is a power of B. Therefore, B can be used to increase or decrease discrimination between points.

B will be set to 300,000 throughout the performance evaluation in this report, as it was found to provide best performance on the target hardware and datasets.

4.4 Bit Hash

Bit Hash is another structure which reduces the dimensionality of points to one dimension. Instead of using a point's spatial properties, Bit Hash accumulates a hash value by hashing each individual coordinate. Each coordinate is hashed using a 32-bit floating point hashing function. The algorithms used for hashing a point and 32-bit floating point numbers are given in Appendix F.

The structure is susceptible to floating point inaccuracy because it uses the *bits* of the point to produce the hash value. On the controlled performance tests executed in this project, a point is inserted and queried using the exact same floating point values for the coordinates. This means the two points have the same identical bit patterns and these errors will not occur.

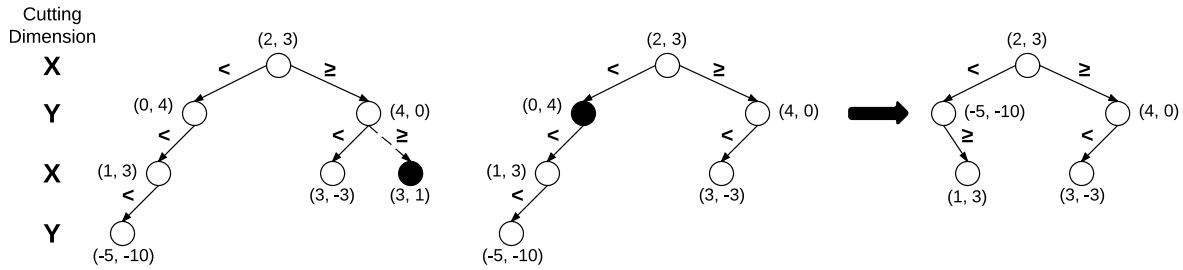
However, suppose the point being queried was the output of some other computation, where rounding errors may come into play. Even if the point is *mathematically* identical to a point stored in the structure, rounding errors can cause the two points to have different bit patterns and thus, have different hashed values. Despite the point being stored in the structure, it will appear as if it is not.

This makes Bit Hash unreliable for certain applications, especially ones where the input points are the result of computations involving many arithmetic operations. Nevertheless, for the small subset of applications where all the points are pre-loaded into memory and not computed again, this can be a useful structure. Initial experiments have also shown this structure has good point query performance, so it will be explored further in this project.

4.5 Point *kd*-Tree

The *kd*-tree is a widely used structure for both low and high dimensional data. In the literature, there is a lot of discussion as to how and why this structure performs poorly in a high-dimensional setting [84, 85]. *kd*-trees are typically used for *approximate* queries in higher-dimensional space, because they degenerate to sequential scan for exact range and nearest neighbour queries [86].

Similar to the Pyramid Tree, the focus of the literature on *kd*-trees is typically for range and nearest neighbour queries, and not point queries. Unlike the Pyramid Tree however, dynamic variants of *kd*-trees have been explored much more (e.g. in [38, 47]), with the original paper proposing operations for incremental insertion and removal [45]. Assessing the suitability of multiple classes of structures provides a more comprehensive evaluation. The *kd*-tree has been chosen because it belongs to a



(a) Inserting Point $(3,1)$ (b) Deleting Point $(0,4)$

Figure 4.3: *kd*-tree Operations for 2D Dataset (Cutting Dimension Alternates Between X and Y)

different class of index structure than the other chosen structures, which use dimension reduction.

The structure is a multi-dimensional binary tree, meaning the number of nodes does not increase exponentially with d like the octree. Instead of splitting by all dimensions at each level, *kd*-trees split the data space along a single dimension, called the *cutting dimension*.

The variant of the tree implemented for this project is the point *kd*-tree (Section 3.3.2). The cutting dimension at each level is chosen by cycling through the dimensions. If n is a *kd*-tree node, i is the cutting dimension on that node's level and p is the point stored in this node, then:

1. Let q be any point in the left subtree rooted at n: $p_i > q_i$ holds
2. Let q be any point in the right subtree rooted at n: $p_i \leq q_i$ holds

When inserting or querying a point p, the tree is traversed top-down in a similar fashion to a standard binary tree, by comparing p_i to q_i, where q is the point stored in the current node and i is the cutting dimension of the current level.

The `delete` operation used is described in [87]. Relative to `insert` and point queries, this operation requires more computation because it has to maintain the *kd*-tree invariant. To remove a point p, first the node containing p is found using the aforementioned top-down traversal approach. Let n be this node and i be the current level's cutting dimension. If n is a leaf, then the node is simply deleted. Otherwise, let n_L and n_R be the left and right children of n respectively.

If n_R exists, a node m containing a point with the *minimum* value for dimension i is found in the sub-tree rooted at n_R. The points stored in nodes n and m are swapped and `delete(`p`)` is called recursively on the sub-tree rooted at n_R.

If n_R does not exist, node m is found in the sub-tree rooted at n_L instead. In this case, after `delete(`p`)` has been recursively called on the sub-tree rooted at n_L, the ith value of all points stored in this sub-tree are greater than or equal to the ith value of the new point in n. Therefore, n_L then becomes the *right* child of n to maintain the *kd*-tree invariant.

Figure 4.3 illustrates the `insert` and `delete` operations.

Chapter 5

Design and Implementation

This section describes the implementation details of the chosen structures, documenting the efforts that went into optimisation them. All execution times given in this chapter were measured by executing sequences of structure operations five times and using the average of these times. How the times were measured is described in detail later in the report, in Section 6.2.

5.1 Evaluation Framework

Evaluation forms a major part of this project. It was decided a framework should be used to streamline this process and reduce the time it takes to generate the measurements required for evaluation. The framework was created because performance analysis would be done very frequently, multiple times per iteration. Manually timing and profiling the code each time is time-consuming and error-prone. Putting more effort initially into creating a system that allows large sets of operations and index structures to be tested at once can save a significant amount of time later on. No tool for this task existed, so it was created for this project.

The evaluation framework takes a specification containing all the datasets and index structures to test, and executes sequences of operations using the given datasets, on all specified structures. The output is a set of text files containing the performance measures described in the next chapter (in Section 6.1). These text files are fed into Python scripts to automatically generate tables and plots.

There are three core modules of the framework, which are shown in Figure 5.1 and listed below:

- `data` – library that contains standard data types used throughout framework and index structures, as well as dataset generators/loaders
- `index_structure` – library that contains every index structure implemented in the project
- `evaluator` – executable program that takes specification file and runs a set of automated performance tests to produce text files containing evaluation measures

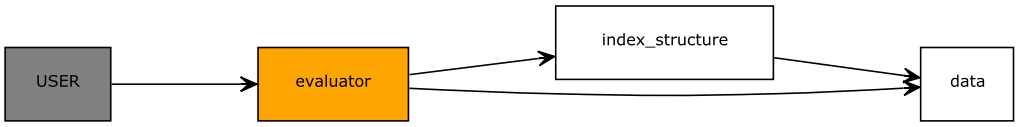


Figure 5.1: Dependency Graph of Evaluation Framework Modules

5.2 Baseline Implementations

Sequential scan was implemented using the C++ container `std::vector`, a dynamically resizeable array. Point queries are performed in $O(n)$ time, as the query iterates through the array sequentially until the given point is found or the end of the array is reached. The structure must check if a point exists in the structure before inserting it, making `insert` take $O(n)$. `delete` is a standard array deletion, making it an $O(n)$ operation.

The octree variant implemented is a bucket PR octree. The structure partitions the underlying data space into uniformly sized boxes, without using the points as pivots (unlike point quadtrees). It is a bucket octree because *multiple* points can be stored in a single leaf node. The octree dynamically decomposes spatial regions based on the stored points. When the number of points in a leaf node exceeds a certain number, say M, then the region represented by the leaf is sub-divided, creating 2^d children. The $M+1$ points are then scattered across the children. Therefore, dense regions of space have a finer decomposition than sparse regions.

When a point is deleted from a leaf, the leaf's contents and its siblings are checked. If all of these nodes are empty, they are removed and the sub-regions are collapsed into a single node representing the whole region. If the collapsed node and its siblings are empty, then they are collapsed into their parent. This is repeated until there is a parent whose children contain at least one point.

5.3 Compiler Optimisation

GCC[1] was the C++ compiler used in this project. GCC can automatically modify the compiled program to make it more efficient on the target platform. There are multiple optimisation levels, going from 0 (no optimisation) to 3 (maximum optimisation).

Table 5.1 shows the execution times of the two baselines with 10,000 random 10D points, when level 0 and level 3 optimisation is used. Level 3 optimisation consistently increased performance, showing that the potential speed gains are huge. For example, the octree's point queries are approximately 6.5 times faster, just by enabling level 3 optimisation. In order to make the structures as fast as possible, level 3 optimisation will be used when compiling all index structures in the project.

[1] GCC, the GNU Compiler Collection – `http://gcc.gnu.org/`

Structure	Operation	Level 0	Level 3
Sequential Scan	Insert	1.33509	0.111372
	Delete	5.29548	0.623759
	Point Query	1.33468	0.111909
Octree	Insert	1.23108	0.243256
	Delete	0.73156	0.13589
	Point Query	0.559736	0.0860926

Table 5.1: Total Execution Time (in Seconds) of Structure Operations With Different Levels of Compiler Optimisation (10,000 10D Random Points)

5.4 Hash-Based Structures

This section discusses the low-level implementation details of the hash-based structures developed in this project. Multiple implementations are developed and analysed to determine which is the fastest.

5.4.1 Initial Hash Structure Implementation

The Pyramid Tree, Pseudo-Pyramid Tree and Bit Hash are essentially hashing functions, where the one-dimensional value acts as the hash to use when searching a hash table. An implementation of a generic hash structure, where the hashing function can be varied, will be created for these hash-based structures to use. The structure stores all the points in a single array, which hashes each point to an integer representation that acts as the key to a bucket stored in a hash map (specifically `boost::unordered_map`, part of the Boost[2] library). A bucket is an array that contains indices to points in the large array.

To determine if a point p is stored in the structure, p is first hashed into its one-dimensional representation. The corresponding bucket is then sequentially scanned until the point is found or the end of the bucket is reached. To reduce the number of floating point comparisons, especially for large d, the *sums* of each point are stored when they are inserted. This way, a $O(d)$ point comparison only needs to be performed if the $O(1)$ point sum comparison passes.

To delete a point p, the point is first hashed and the bucket containing the point is found. The index in the bucket pointing to p is removed, but the actual point itself is not removed from the point array. This will not release the memory used by the deleted point. This is not suitable if the structure is used as part of a long-running process because there is the potential to run out of memory. One of the core assumptions of this project is that data may be dynamic, meaning points may be inserted and deleted frequently. For this reason, memory must be released when deleting points.

The **Rebuild Hash Structure** releases memory when R points have been removed, where R is a configurable parameter. When a point is removed, the index of the element storing the deleted point is marked. Once R elements are marked, a rebuild procedure is performed. The entire hash structure is rebuilt by first clearing the structure and then inserting all the points *not* marked for

[2] `http://www.boost.org/`

deletion. $n-(R+1)$ points will be re-inserted and insertion in the worst case is $O(n)$, meaning the worst case complexity of `delete` is $O((n-R)n)$. The larger R is, the less often the rebuild procedure executes, but a larger amount of allocated memory goes unused at a time.

5.4.2 Bucket Implementation

The rebuild implementation stores indices in buckets, meaning that points in the same bucket may not be stored contiguously in memory. Searching a bucket in the rebuild implementation is likely to incur a lot of cache misses, because accessing a point will involve a random access into the point array. Reading a point in a bucket also requires the CPU to fetch two elements from main memory – the index of the point and the point itself. Furthermore, having a large point array makes `delete` operations more complicated because the indices stored in buckets must be maintained.

Bucket Hash Structure is another implementation that was created which does not use a single array to store the points. Buckets contain actual points instead of indices to them. This should reduce cache misses since point array is searched sequentially. No rebuild procedure is necessary because the memory for a point is released immediately after it is removed, by simply erasing it from the corresponding bucket's array. The order the points are stored in a bucket do not matter, so the *erase-remove* idiom[3] has been used to efficiently delete elements from a bucket's array.

5.4.3 Splay Tree Variant

The **Splay Hash Structure** uses a splay tree as the underlying one-dimensional index structure, instead of a hash map. The splay tree is a self-adjusting binary search tree that uses a a heuristic to allow faster access to recently accessed elements. [40]. Through amortised analysis and empirical experiments, it has been shown splay trees can be more efficient than standard binary trees for a series of non-random operations [40], despite the worst case bound being worse than binary search trees.

Nodes in the Splay Pseudo-Pyramid Tree correspond to individual buckets in the Bucket Hash Structure. Since the splay tree is implemented as a collection of heap-allocated nodes, deletions are cheap as a low amount of memory needs to be de-allocated per `delete` operation. This, combined with the self-adjusting nature of the splay tree, could produce an more efficient implementation for non-random operations used in real applications.

5.4.4 Cache Misses

Table 5.2 displays the cache miss rate when inserting 10,000 10D random points into each implementation, using the Pseudo-Pyramid Tree hashing function. Notice how the Rebuild Hash Structure, which stores indices in each bucket, has the highest cache miss rate. The Bucket Hash Structure incurs the least amount of cache misses, showing how contiguous memory storage can have a significant impact on cache misses.

[3] `http://en.cppreference.com/w/cpp/algorithm/remove`

Structure	Cache Miss Rate (%) (2 dp)
Rebuild Hash	4.62
Bucket Hash	0.28
Splay Hash	0.59

Table 5.2: Cache Hit Rate for `insert` Operations with Hash Structure Implementation Using Pseudo-Pyramid Tree Hashing Function (10,000 10D Random Points)

Structure	Insert	Delete	Point Query
Rebuild Hash	0.002467	0.019018	0.000977
Bucket Hash	0.003546	0.001448	0.000958
Splay Hash	0.005368	0.002074	0.001542

Table 5.3: Execution Time (in seconds) of Hash Structure Implementations Using Pseudo-Pyramid Tree Hashing Function (10,000 10D Random Points)

5.4.5 Fastest Implementation

Table 5.3 shows the execution times of each operation on each of the structures, for 10,000 random 10D points. The Splay Hash Structure is the slowest for insertions and queries. Rebuild Hash Structure has the slowest deletions because of the rebuild procedure, but its insertion and query performance almost matches the Bucket Hash Structure. The Bucket Hash Structure outperforms the other implementations, coinciding with their cache miss rates. All hash-based structures will use the Bucket Hash Structure implementation for the performance evaluation in this report.

5.4.6 B$^+$-Tree for Underlying Structure of Pyramid Tree

Berchtold et al. originally used a B$^+$-tree when developing the Pyramid Tree [27]. Using the same underlying search structure allows a more fair comparison of the Pyramid Tree's performance to the literature to be made. Two B$^+$-tree implementations, bpt [88] and cpp-btree [89], were tested.

Table 5.4 provides total execution times of each operation using `boost::unordered_map` and the two B$^+$-tree implementations, measured using 10,000 10D random points. There is a noticeable difference between the speed of the hash table and B$^+$-tree implementations. This matches the theoretical performance analyses of the two structures, where it is shown that hash tables and B$^+$-trees have amortised $O(1)$ and $O(\log n)$ operations respectively. After profiling, it was discovered the main cause of the decrease in speed was the additional overhead incurred by splitting and merging nodes in the B$^+$-tree. Based on these results, it has been decided to continue using the hash table as the underlying search structure for the Pyramid Tree.

5.5 SSE Optimisation

Point equality tests have been accelerated using SSE. Considering point equality is used throughout all the index structures, there should be some noticeable speed gains in the structures themselves. It is

Operation	`boost::unordered_map`	bpt	cpp-btree
Delete	0.002206	0.004544	0.003617
Insert	0.005275	0.010445	0.012766
Point Query	0.001534	0.002501	0.002808

Table 5.4: Total Execution Time (in seconds) of Pyramid Tree With Different Underlying 1D Index Structures (10,000 10D Random Points)

Operation	Without SSE	With SSE
Delete	0.002194	0.001613
Insert	0.005188	0.003856
Point Query	0.001577	0.001122

Table 5.5: Total Execution Time (in seconds) of Bucket Hash Structure Using Pyramid Tree Hashing Function, With and Without SSE Optimisation (10,000 10D Random Points)

also possible to parallelise the hashing functions used in this project. In this section, the Pyramid tree hashing function is considered. Both this hashing function and point equality are executed in $O(d)$ time. In both operations there is loop which iterates once for each dimension, where each iteration is independent of the others. If 32-bit floating point numbers are used, then four dimensions can be processed at once using 128 bit SSE registers. If the number of dimensions is less than four, there is little benefit using SSE. When $d < 4$, the structures use the sequential version of the code instead.

Table 5.5 shows the execution times of the Pyramid Tree with and without SSE optimisation for its hash function and point equality. Across all three operations, the average speedup is approximately 1.37 when there are 10 dimensions. Considering the ideal speedup is 4, it was expected the speed up would be higher. Nevertheless, parallelising point equality consistently achieves speed up for all structures. Other parts of the implementations have been accelerated using SSE, but for brevity have not been included in this report.

5.6 *kd*-tree Implementation

The point *kd*-tree variant described in Section 4.5 was implemented. Each node is represented as a C-struct which stores a single point as well as pointers to its children. Every node is allocated using a call to `malloc()`, which allocates enough memory on the heap to store the node. This means each node may reside in different areas of the heap, resulting in fragmented memory. Array-based implementations that store *kd*-tree contiguously in memory exist, but these have not been explored due to time constraints.

Chapter 6

Evaluation Methodology

The heart of this project is to determine which index structure performs the best for high-dimensional, scientific datasets. Doing this requires an evaluation of the implemented index structures. This section details the measurements and datasets used for this evaluation.

6.1 Performance Measures

The project's scope, as defined in Section 1.5, considers dynamic data, meaning points may be inserted or deleted at any time. Additionally, the project's focus is on point query performance. Therefore, index structures are evaluated against the performance of their **insert**, **delete** and **point query** operations. The **total execution time** of each operation is measured.

How much memory a structure uses is also an important factor to consider, especially if the input data is large. Therefore, the **memory overhead factor** will also be measured. The overhead factor measures how much additional memory is needed to store points. An overhead factor of 1 means there is no overhead. A factor of 2 would mean *twice* the minimum amount of memory (which is the data required for the actual points) is used.

6.2 Timing Operations

Instead of analysing the performance of a single operation, one can analyse performance of an algorithm over a *sequence* of operations. Amortised analysis, introduced by Tarjan in 1985, is a technique which does this [90]. This type of analysis is typically used for algorithms where some operations take longer than others in order to make future operations quicker. The amortised time complexity of an algorithm describes the *average* runtime of an operation over a sequence of operations. The splay tree, quadtreap and splay quadtree (see Section 3.3.5) are examples of index structures that have used amortised analysis to bound the running time of their operations.

This project will use a similar concept for measuring the execution time of the structures. Instead of measuring the time of a single operation, the times of many thousands of operations will be measured. This is to get an idea on what the runtime of an operation is on average, over a sequence of operations. This prevents single, slow operations from causing the execution time to appear worse, or short operations making the performance appear better. Each performance test will invoke the **Insert-Query-Delete** operation list on each structure, which has the following steps:

1. **Initial Build** – incrementally build structure by adding each point in test dataset one-by-one
2. **Query Points** – query each point in the dataset
3. **Delete Points** – delete each point in the dataset from the structure until it is empty

Using this process allows the runtime of large numbers of `insert`, `delete` and point query operations to be measured. Any performance tests using the Insert-Query-Delete operation list will be performed five times, and the recorded time will be the *average* of those five times.

6.3 Datasets

This section defines the datasets that will be used for the evaluation, which includes both synthetic, artificially generated data and real datasets.

6.3.1 Synthetic Datasets

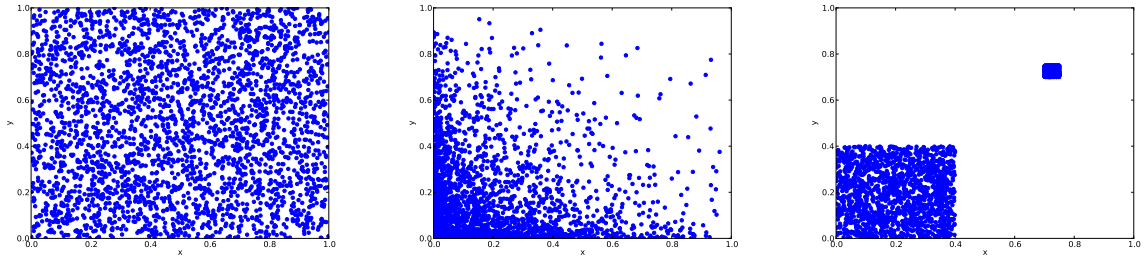
A variety of point distributions and number of dimensions may tease out the types of data the structures are suited to and which they struggle with. This project uses three kinds of synthetic data, which have uniform, skewed and clustered distributions respectively. Figure 6.1 shows 2D randomly generated points using uniform, skewed and clustered distributions. All generated points are in $[0,1]^d$.

Multiple instances of these datasets will be used, with varying numbers of dimensions. Using such datasets was inspired by Wang et al. and Berchtold et al., who used similar datasets to evaluate the performance of the PK-tree and pyramid tree respectively [27, 30].

To ensure the synthetic data is stastically fair, the Mersenne twister pseudorandom number generator (PRNG) was used to generate the points. The Mersenne twister passes "stringent statistical tests" of randomness [91], ensuring a fair distribution of points. C/C++ provides built-in random number generator, used by invoking the `rand()` function. However, the underlying PRNG used by `rand()` is platform-dependent, so there are no guarantees it will be statistically fair. The Boost.Random[1] library implements the Mersenne twister, and has been chosen over `rand()`.

Skewed data was generated by applying a power to the number. Let p be a generated point. Since $0 \leq p_i \leq 1$ for all $i \in \{1,...,d\}$, this means $p_i^e \leq p_i$ for all $e \geq 1$. This makes smaller values more likely, generating a skewed distribution of points. Increasing e increases the skew. $e = 1.5$ is used to the generate skewed datasets so there are still some points in the upper regions of the data space.

[1] `http://www.boost.org/doc/libs/1_48_0/doc/html/boost_random.html`



(a) Uniform Distribution (b) Skewed Distribution (c) Clustered Distribution

Figure 6.1: Three Types of Synthetic Dataset Used For Evaluation

The clustered datasets contain two clusters of points, each contained within the hypercubes $[0, 0.5]^d$ and $[0.7, 0.8]^d$. There is zero probability that points will appear outside of those cubes.

Several synthetic datasets with 10,000 points have been generated for each distribution, varying in dimensionality. The structures will also be evaluated against datasets of varying size. Datasets with different sizes have been generated by sampling from a uniform distribution. These datasets have 16 dimensions to match the Pyramid Tree tests performed by Berchtold et al. in [27].

6.3.2 Real Datasets

Three real datasets will also be used for the evaluation. The first is the result of an **astrophysics turbulence simulation**, where a $600 \times 248 \times 248$ regular mesh was used to simulate "three-dimensional radiation hydrodynamical calculations of ionization front instabilities" [92]. Each point of the mesh has ten scalar fields, which include particle density, temperature and eight chemical species. 200 timesteps were recorded, each being approximately 126 to 128 years apart [92].

A single timestep is used for evaluation, which is timestep 100. This timestep was chosen because there is a greater amount of *"activity"* in the data. The phenomena being modelled starts develop as more time passes, meaning there is greater variance in points. The timesteps near the beginning of simulation are sparse and uniform. Timestep 100 will still contain some amount of sparsity, resulting in dense regions of activity surrounded by relatively sparse, uniform regions. Distributions like this are common in scientific computation, so this timestep should act as a good test on how well the index structures can handle scientific datasets.

The second real dataset contains 13 scalar fields for every point on a $500 \times 100 \times 100$ mesh, over 48 timesteps. It is the result of a simulation of hurricane Isabel from 2003 [93]. Timestep 24 of this simulation will be used because, again, the data is highly sparse with dense regions of activity. In the original simulation, nothing was recorded in regions of space that contained ground. The points inside those regions have values of 1.0e+35 to mark they have not been recorded. Since these points given no useful information about the simulation, they have been removed from the dataset.

The astrophysics and hurricane datasets were chosen because they are large and have a high num-

Dataset(s)	**Fixed Parameter(s)**	**Varying Parameter**	**Parameter Values**
Uniform, Skewed, Clustered	10,000 Points	Dimension	1, 2, 3, 5, 10, 50, 100, 200
16D Uniform	16 dimensions	Input Size	10,000, 100,000, 500,000
Astrophysics ($t = 100$)	10 dimensions, 500,000 points	-	-
Hurricane Isabel ($t = 24$)	13 dimensions, 500,000 points	-	-
Armadillo Mesh	3 dimensions, 435,544 points	-	-

Table 6.1: Evaluation Datasets

ber of dimensions (10 and 13 respectively). The core focus of the project is high-dimensional scientific data, so this data will provide a true test of how well the implementations perform with real instances of such data. Additionally, both datasets are used for scientific visualisation and contain continuous values. As discussed in Section 1.5, the project is targeting this type of data.

Each of these datasets contains millions of points. It was decided that a smaller instance of these datasets would be used, sampling 500,000 points from both datasets. Sampling a sub-region of the grid or uniformly sampling can introduce *bias* in the data, which affects the evaluation. Therefore, smaller instances of the dataset are generated by randomly sampling n points from the original dataset, discarding and picking other points whenever a point that has already been sampled was chosen. The law of large numbers [94] shows that, as the number of samples increase, the sampled data becomes more representative of the whole dataset, making this method statistically fairer than uniform sampling.

The final real dataset is a point cloud of a 3D armadillo from The Stanford 3D Scanning Repository [95], which contains 435,544 points that represent a geometric mesh. This mesh has been used by other researchers to evaluate index structure performance applied to 3D graphics [66, 96]. Appendix D.3 shows visualisations of the three real datasets.

Table 6.1 lists the evaluation datasets, their size and dimensionality and lists parameters that will be varied.

6.4 Environment

Table 6.2 lists the hardware, operating system and tool versions used to implement the index structures and measure their performance.

CPU	Intel(R) Xeon(R) CPU E3-1225 V2 @ 3.20 GHz (four cores)
Main Memory (RAM)	15.5 GB
Operating System	Linux CentOS 6.5 (Final)
Linux Kernel Version	2.6.32-431.3.1.el6.x86_64
C++ Compiler	GCC 4.4.7 (Released 13/03/2013)
Boost Library Version	1.41.0

Table 6.2: Hardware, OS and Tool Versions Used for Development and Performance Testing

6.4.1 Measuring Execution Time

Individual index structure operations are typically much lower than a second, sometimes even less than a millisecond. A high level of precision and accuracy is required because even millisecond savings are important when optimising index structures.

C/C++ comes with several timing mechanisms, such as `clock_gettime()`, `time()` and `clock()`[2]. `clock_gettime()` will be used for timing all operations because it has nanosecond precision, the highest available precision on Linux systems [97, 98].

6.5 Initial Hypothesis

Since there has been no published evaluations of the Pseudo-Pyramid Tree or Bit Hash, no hypotheses will be made regarding the performance of these structures. It is suspected these structures will perform well on high-dimensional data, because they are hash-based methods, but little is currently known about how well the hashing functions perform, so there is a little basis for such suspicion. Based on the reported performance of the Pyramid Tree and *kd*-tree in a high-dimensional setting in the literature (discussed in Chapters 3 and 4), the following hypothesis was devised:

HYPOTHESIS: Pyramid Tree will be faster than the baselines and *kd*-trees on scientific datasets containing high-dimensional data ($\geq$ 10 dimensions).

6.6 Conclusion

Each index structure will be evaluated using a collection of synthetic and real datasets. For each dataset, all points will be inserted into each structure, then all points will be queried and finally, all points will be deleted. The total execution time of each operation type will be measured, alongside the memory overhead produced by the structures.

Based on the existing literature, it is hypothesised that the Pyramid Tree will outperform the *kd*-tree on high-dimensional scientific datasets. The next chapter will now evaluate the performance of the chosen structures and determine if this hypothesis still holds.

[2]Online documentation for these functions is available at: `http://www.cplusplus.com/reference/`

Chapter 7

Performance Evaluation

This section evaluates the performance of the index structures described in Chapter 4 using their execution times on all the synthetic and real datasets. Execution times are illustrated with plots and bar charts. Tables containing all execution times are available in Appendix E.

Since both point insertion or deletion will have to perform a point query (`insert` has to check the point is not already stored), point query execution time will not be plotted for some tests. See Appendix E for all point query times.

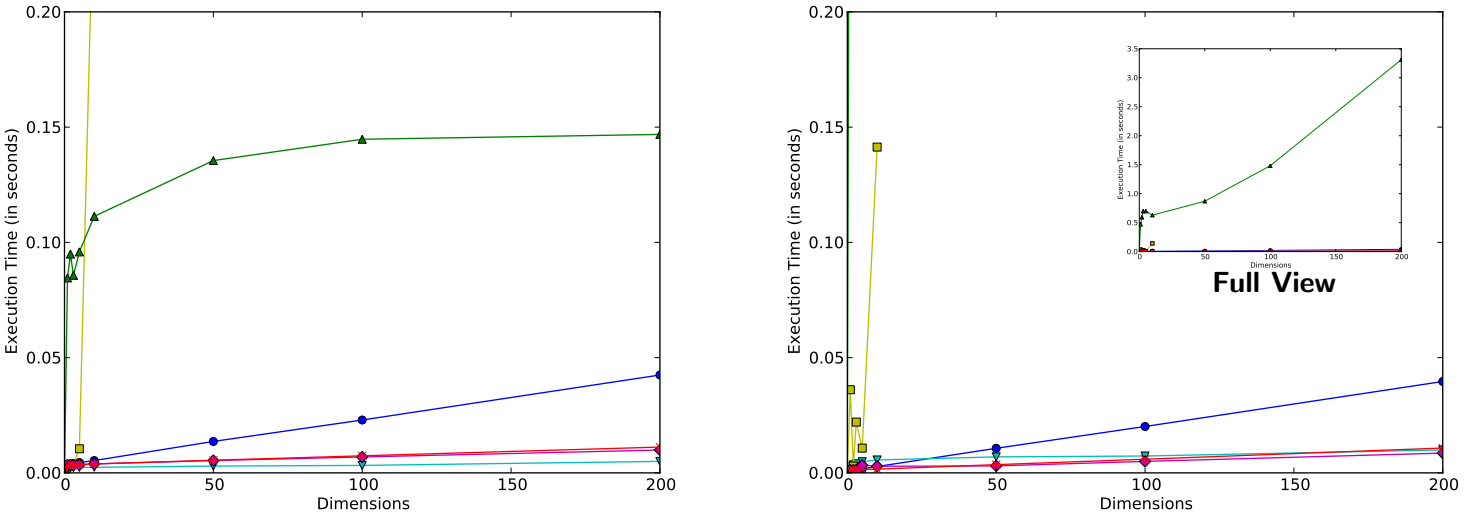
7.1 Dimensionality

Figure 7.1 plots point *dimensionality* against execution time of `insert` and `delete` operations for each structure. It shows that the performance of all structures gets worse as d increases, but the rate performance decreases varies. For insertions, sequential scan quickly becomes slower as d increases, but the decrease in speed tails off as d gets higher. For example, there is little different between `insert` execution time when the dimensions increase from 100 to 200. `delete` is sensitive to d, with its execution time increasing much faster as d increases. Higher d means more data must be moved when performing array deletion. Sequential scan pre-allocates memory for new points, so having more data per point has less of an impact on performance for insertions.

The plots show that octree execution time exponentially increases with d. When $d > 10$, the sheer amount of nodes needed when sub-dividing space causes the structure to run out of memory. From this we can conclude that this structure is not suitable for high-dimensional data.

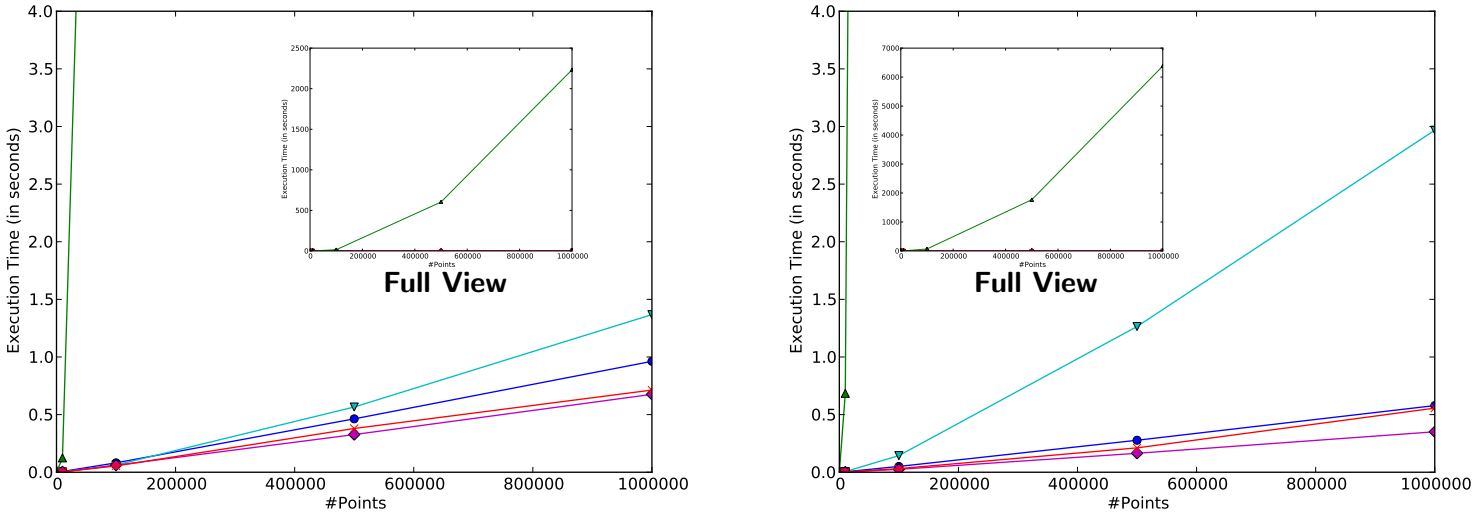
All three hash-based structures use an $O(d)$ hashing function, but the constant factors in each are different. Execution times of Bit Hash, Pseudo-Pyramid Tree and Pyramid Tree all increase linearly with d, but Bit Hash has a greater slope. Pseudo-Pyramid Tree has the cheapest hash function and as such, is the fastest of the three structures as d increases.

There have been discussions regarding the *kd*-tree's unsuitability for high-dimensional data in the literature [84, 85]. However, the *kd*-tree is affected the least by dimensionality, which was initially



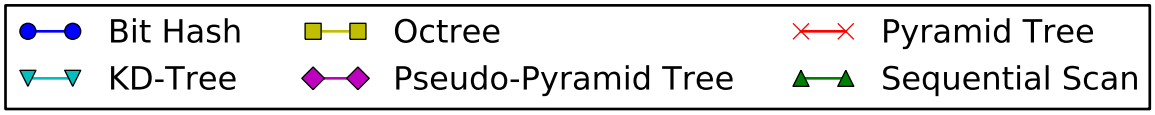
(a) `insert` (b) `delete`

Figure 7.1: Structure Performance With Respect To Dimensionality (10,000 Points, Uniform Synthetic Dataset)



(a) `insert` (b) `delete`

Figure 7.2: Structure Performance With Respect To Dataset Size (16D, Uniform Synthetic Dataset)



surprising. However, the structure does not have an explicit $O(d)$ computation; its best and worst running times are $O(\log_2 n)$ and $O(n)$ respectively. Therefore, as long as the tree remains balanced, dimensionality will have little impact on the point *kd*-tree's performance.

7.2 Dataset Size

Figure 7.2 plots dataset *size* against execution time of `insert` and `delete` operations for each structure. Sequential scan is clearly not scalable to large dynamic datasets, taking over 2000 seconds to incrementally construct a structure to store 1,000,000 unique points and over 6000 seconds to incrementally delete the points. The octree ran out of memory quickly, since 16-dimensional points are used. Again, this shows the octree is unsuitable for high-dimensional data.

Figure 7.1 shows the *kd*-tree is the fastest with 10,000 points, but it becomes the slowest (barring sequential scan) as dataset size increases. `delete` becomes even slower than `insert` does as n increases, with deletions being two times slower than insertions with 1,000,000 points. This is a problem if an application removes individual points from the structure frequently.

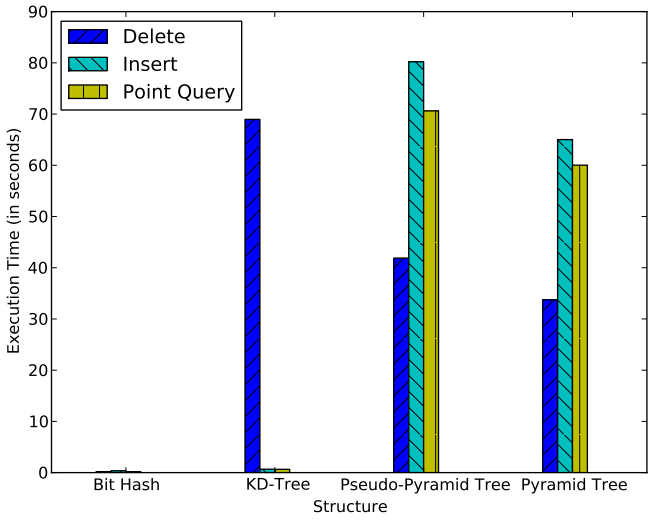
The Pseudo-Pyramid Tree and Bit Hash have the fastest and slowest hashing functions respectively. This again affects the performance of the structures, with the Pseudo-Pyramid Tree being the fastest hash-based structure and Bit Hash being the slowest. These tests show that the three hash-based structures are the most scalable structures out of six evaluated. As more synthetic data is given to the structures, execution time of the hash-based structures increases much slower than the point *kd*-tree or sequential scan.

7.3 Real Data

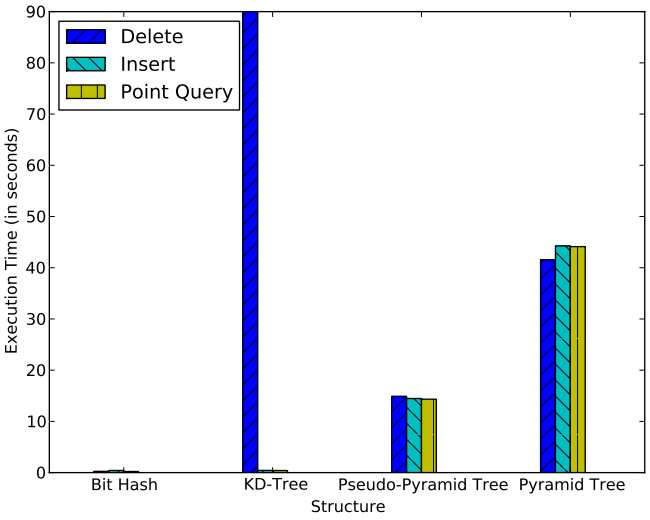
Figure 7.3 illustrates index structure performance of each operation on the three real datasets. Sequential scan is not shown because performance is similar to that shown in the dataset size plots in Figure 7.2. See Table E.5 in Appendix E for all execution times on the real datasets.

Bit Hash outperforms all structures for point queries and deletion on all three datasets. It has the fastest insertion time as well, except for the armadillo mesh where the Pseudo-Pyramid Tree and Pyramid Tree is faster. Profiling revealed that Bit Hash performs more memory allocations for insertions, which is the reason it is slower than point deletions or queries. Bit Hash is significantly faster than the other hash-based approaches on the astrophysics and hurricane Isabel dataset, with queries being over 400 and 198 times faster than the Pyramid Tree respectively.

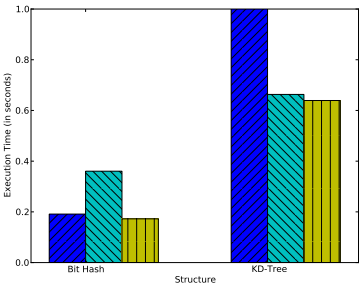
kd-trees vastly outperform Pseudo-Pyramid Tree and Pyramid Tree and almost match Bit Hash's speed. However, individual point deletion is slow. It took almost 580 seconds to delete each point from the hurricane dataset. It is interesting see that the Pyramid Tree was faster than Pseudo-Pyramid Tree for the astrophysics dataset, but over two times slower with the hurricane dataset. The hashing functions appear to be highly sensitive to data distribution. This will be explored in the next section.



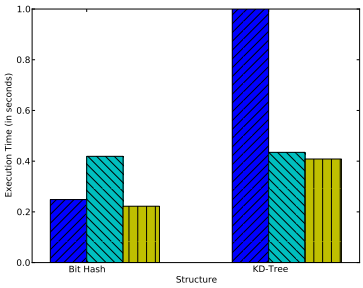
(a) Sampled Astrophysics (500,000 Points)



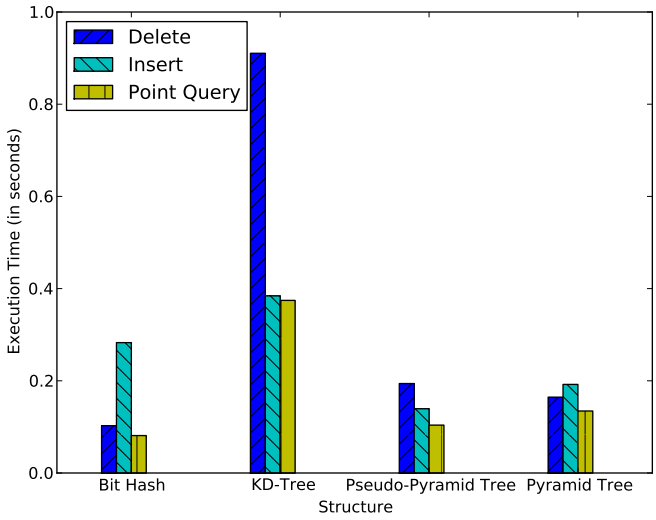
(b) Sampled Hurricane Isabel (500,000 Points)



(c) Zoomed Sampled Astrophysics



(d) Zoomed Sampled Hurricane Isabel



(e) Armadillo Mesh (435,544 Points)

Figure 7.3: Structure Performance on Real Datasets

		Bucket Statistics		
Dataset	**Time to Query (sec)**	**Average**	**Max**	**#Buckets**
Pseudo-Pyramid Tree				
500,000 16D Random Points	0.105091	1.02395	4	488304
500,000 Astrophysics Points	70.4391	62500	153471	8
500,000 Hurricane Isabel Points	14.3358	1262.63	154979	396
435,544 3D Armadillo Mesh Points	0.141219	19.1465	187	22748
Pyramid Tree				
500,000 16D Random Points	0.177153	1.16286	7	429974
500,000 Astrophysics Points	60.0216	3521.13	235260	142
500,000 Hurricane Isabel Points	44.1224	4065.04	354965	123
435,544 3D Armadillo Mesh Points	0.13448	2.45938	1173	177095
Bit Hash				
500,000 16D Random Points	0.249102	1.00975	3	495172
500,000 Astrophysics Points	0.172516	1.01051	5	494800
500,000 Hurricane Isabel Points	0.222288	1.00861	3	495733
435,544 3D Armadillo Mesh Points	0.0811833	1.00891	3	431696

Table 7.1: Statistics on Bucket Size, Based on Dataset, of All Hash-Based Structures

7.4 Impact of Bucket Size

Pyramid Tree point queries are approximately 7 times faster than sequential scan with the astrophysics dataset and 14 times faster with hurricane Isabel. With the armadillo mesh, Pyramid Tree queries are *1580* times faster. The speedup varies massively between datasets. Considering algorithms which perform queries in $O(\log n)$ time exist, this raises an important question: why is the Pyramid Tree so slow with some datasets?

With the real datasets, Bit Hash is the fastest structure and Pyramid Tree is the slowest. Both structures use the same underlying hash table implementation. Therefore, the performance differences must be caused by how the structures reduce points to a one-dimensional value.

Despite hashing a point taking $O(d)$ time, point queries will take much longer if buckets contain many points. If each bucket contains exactly one point, then operation execution time approaches $O(d)$. On the other extreme, where a single bucket contains all points, the complexity becomes $O(n)$. The number of points in a bucket, or *bucket size*, is one of the most important factors to consider when analysing the performance of hash-based index structures. A "good" hashing function tries to achieve an average running time of $O(d)$ by ensuring very few points are mapped to the same bucket.

Table 7.1 shows the average and maximum bucket size, as well as how many buckets there are, for the real datasets and a uniformly distributed synthetic dataset. The table shows there is a relationship between bucket size and speed – higher average bucket sizes are paired with longer query times.

The Pyramid Tree stores 235,260 points from the astrophysics dataset in a *single* bucket, meaning up to 235,260 points will be scanned when performing a query. Both the Pyramid and Pseudo-Pyramid Tree essentially degenerate to a *semi-sequential scan* with the scientific datasets. Bit Hash's

average bucket size is very close to 1, meaning only one point must be scanned for most operations. Larger average bucket size means queries require more computation on average. The Pyramid Tree and Pseudo-Pyramid Tree are much faster on the synthetic dataset and the armadillo mesh than the two scientific datasets. Table 7.1 shows how average bucket size is much lower for these datasets when compared to the two scientific datasets. This shows that, given certain point distributions, these structures can be fast.

Ideally, the number of buckets is equal to the number of points, meaning average bucket size is 1. This is known as *perfect hashing*. Bit Hash shows that consistently good performance can be achieved when the average bucket size tends to one. Unfortunately, Bit Hash is unreliable because it is highly susceptible floating-point errors (see Section 4.4). If a more reliable hashing function that achieves close to 1 average bucket size, then significant increases in point query performance can be achieved.

7.5 Impact of Tree Balance

While the difference in performance is less drastic than the hash-based structures, the point *kd*-tree's performance still varies between datasets. *kd*-tree point queries took 0.639265 seconds with 500,000 points sampled from astrophysics dataset, but only 0.374425 seconds on the armadillo mesh. The cause of this is most likely the *balance* of the tree. Skewed datasets are known to affect the balance of spatial decomposition trees, resulting in longer paths from the root node to the leaves.

To investigate this further, the *balance factor* of the point *kd*-tree has been measured. The balance factor is defined as the *average* length of a path from the root to a leaf [46]. A lower balance factor means the *kd*-tree has to visit less nodes on average when performing a point query, so queries will be faster. A *kd*-tree is balanced when its balance factor is $\log_2 n$. When $n = 500,000$, the best possible balance factor is $\log_2(500,000) \approx 19$.

The *exact match query cost*, used to evaluate *kd*-tree variations by Dandamudi and Sorenson [46], represents the average number of points visited in the tree for a point query. Higher exact match query cost means the tree has to do more work on average to find a point, making the structure slower. This will be measured alongside balance factor to determine the average work required per operation.

Table 7.2 contains the balance factor, exact match query cost and maximum path length from the root to a leaf when different point distributions are stored in the tree. It shows exact match query cost increases with the balance factor. Therefore, as the balance factor increases, performance decreases.

While the *kd*-tree does not degenerate to semi-sequential scan when processing datasets with skew, like the Pyramid Tree, these results show that these datasets do affect the structure's overall performance. For this reason, many variants of the *kd*-tree have been developed in an attempt to reduce the balance factor with skewed datasets. Examples of such variants include the BD-Tree [46] and fair split tree [99]. Appendix G discusses initial performance results of a *kd*-tree variant implemented in this project. This variant was not explored in more detail due to time constraints.

Dataset	Balance Factor	Max Path Length	Exact Match Query Cost
Random 10D Points (Uniform Distribution)	24.8285	47	24.4968
Random 10D Points (Skewed Distribution)	24.9092	44	24.5714
Random 10D Points (Clustered Distribution)	30.3362	57	29.9911
Astrophysics	32.405	120	29.7373
Hurricane Isabel	47.7236	123	40.5602

Table 7.2: Point *kd*-tree Balance Factor and Exact Match Query Cost with 500,000 Points from Various Datasets

	16D Synthetic Data (Uniform)		10D Astrophysics	
Structure	**Memory Usage**	**Overhead**	**Memory Usage**	**Overhead**
Sequential Scan	30.53MB	1.000327	19.074MB	1.000327
Pseudo-Pyramid Tree	51.53MB	1.688401	20.98MB	1.099984
Pyramid Tree	42.493MB	1.392300	20.99MB	1.100508
Bit Hash	54.86MB	1.797509	43.42MB	2.276516
kd-tree	38.15MB	1.25	26.7MB	1.399884

Table 7.3: Memory Overhead of Structures When Storing 500,000 Points from Different Datasets

7.6 Memory Overhead

The memory consumption of each structure has been recorded alongside execution time. If a structure's memory footprint is large, it may not scalable to large datasets. 32-bit floating point numbers are used in this project, so the minimum amount of memory required to store 500,000 points from the 16D synthetic dataset and the 10D astrophysics dataset is 30.52MB and 19.07MB respectively.

Table 7.3 shows the memory overhead of each structure when storing points from one synthetic and one real dataset. Similar results were found with the other test datasets, so for brevity they are not shown. Octree runs of memory and crashes, so no measurements are provided for it.

Sequential scan incurs almost no overhead since only three extra pointers are used by the underlying resizeable array. The point *kd*-tree produces the next lowest overhead. Each point in the tree incurs 8/16 bytes of overhead, since they are stored in a node that contains pointers to its two children.

The overhead incurred by the hash-based structures depends on the dataset. Each bucket adds overhead, so memory overhead is larger if there are many buckets. For execution speed, it is ideal to have one bucket for each point, but this consumes more memory. The Pseudo-Pyramid Tree maps all points in the astrophysics dataset to only 8 buckets and the structure's overhead factor is 1.099984. The structure uses 493,488 buckets for the synthetic data and has an overhead factor of 1.688401. Notice how the overhead factor is much higher. Bit Hash tends to use one bucket for each point, so it consumes the most memory out of all the structures. While perfect hashing is desirable for speed, it

can cause problems if large datasets must be processed on memory constrained systems. If memory is a limited resource, then the point *kd*-tree may be more suitable because it has relatively low memory overhead, which is not dependent on point distribution.

7.7 Conclusion

The performance evaluation performed in this chapter has revealed the following:

- Sequential Scan is not scalable to large datasets
- the standard octree is not usable for high-dimensional data
- Pseudo-Pyramid Tree and Pyramid Tree degenerate to semi-sequential scan with the two scientific datasets, but perform well with the armadillo mesh and synthetic datasets
- point *kd*-tree speed is affected by point distribution, but not as much as hash-based structures
- the point *kd*-tree outperforms the Pseudo-Pyramid Tree and Pyramid Tree on real scientific datasets
- individual point deletion with the point *kd*-tree is highly inefficient
- if near perfect hashing is achieved, then a structure faster than the point *kd*-tree has been found
- hash-based structures use more memory than point *kd*-trees, with overhead increasing as they get closer to perfect hashing
- hash-based structures with near perfect hashing might not be suitable for large datasets if low memory consumption is a priority

The point *kd*-tree is consistently slower than Bit Hash with the real datasets. However, the *kd*-tree is more reliable because it does not use a hashing function that is highly susceptible to floating point error. Bit Hash shows perfect hashing is a desirable property, but it will not be discussed further in the project because of its unreliability.

Chapter 8

Final Discussion

Chapter 7 showed that the point *kd*-tree greatly outperforms the Pyramid Tree and Pseudo-Pyramid Tree with the two scientific datasets. The Pseudo-Pyramid Tree has not been explored in the literature, so there were no expectations of its performance. However, it was hypothesised that the Pyramid Tree would perform better than the point *kd*-tree, which is not the case.

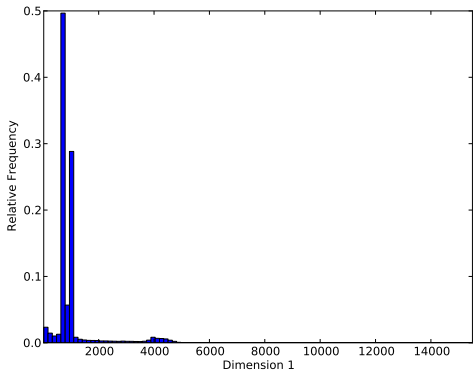
This chapter explores the reasons for this, determining which properties in the astrophysics dataset cause the performance of the Pyramid Tree to degenerate. The chapter concludes by discussing the suitability of the Pyramid Tree, *kd*-tree and different classes of index structures for high-dimensional scientific datasets.

8.1 Characteristics of Astrophysics Dataset

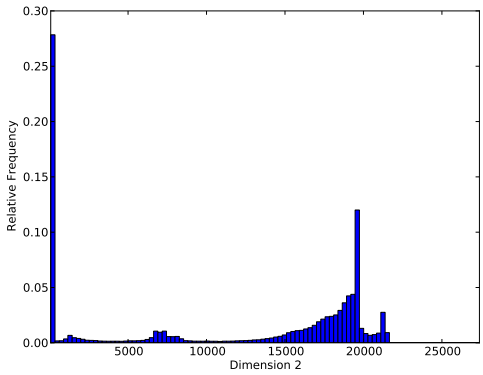
To determine why astrophysics dataset causes structures to perform worse, its point distribution will be explored. A dataset can be considered *skewed* if there are more points in some regions of the data space than others. In other words, the underlying frequency or probability distribution of point locations is non-uniform. Intuitively, the skewness of a dataset increases as the *difference* between the frequency, or probability, of points in different spatial regions increases.

Histograms are used to visualise the frequency distributions of one-dimensional data. Since this project deals with multi-dimensional data, multiple histograms, one for each dimension, can be produced to gain insight into point distribution. The astrophysics dataset was computed using a 3D sampling lattice, computing ten fields at each point. The original simulation uses interpolation to compute the ten fields at each point on the lattice [92]. Carr et al. discusses how this interpolation "implicitly applies the spatial relation between sample points" and shows that histograms are equivalent to nearest-neighbour interpolation [100]. This means histograms poorly represent datasets that use higher-order interpolants, like the astrophysics dataset, because the technique "over-emphasizes densely-sampled regions and under-emphasizes sparsely-sampled regions" [100].

Isosurface statistics have been proposed as a superior representation of such datasets [100]. How-

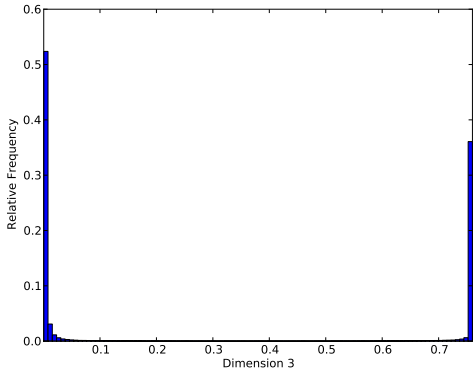


(a) Dimension 1 (total particle density)

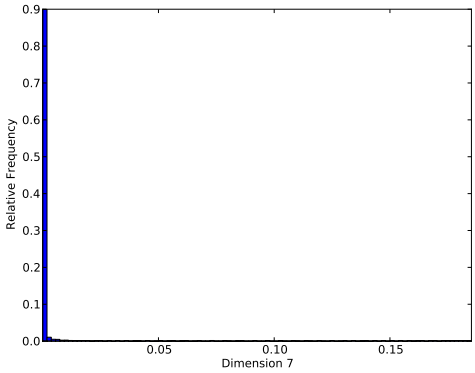


(b) Dimension 2 (gas temperature)

Figure 8.1: Frequency Distributions of Dimensions 1 and 2 of Astrophysics Dataset



(a) Dimension 3 (H mass abundance)



(b) Dimension 7 (He^{++} mass abundance)

Figure 8.2: Frequency Distributions of Dimensions 3 and 7 of Astrophysics Dataset

ever, they are conceptually and computationally more complex to compute. Due to the project's time constraints, isosurface statistics will not be used. While histograms are a poorer representation of the data, the aim of this evaluation is to determine how skewed the data space is. Histograms still allow for this even though they have poor accuracy when compared to other techniques. Therefore, histograms are still used to visualise the distribution of the astrophysics dataset.

Using the same sampled astrophysics dataset used for performance analysis in Chapter 7, a histogram has been generated for each of the ten dimensions. Figures 8.1 and 8.2 show histograms of dimensions 1, 2, 3 and 7, where bin height represents the *relative* frequency of values inside the bin. The remaining dimensions have distributions similar to the dimensions shown here, so they add little value to the discussion. They can be found in Appendix D.4 for completeness. Using these histograms, the following observations can be made:

1. the first two dimensions, total particle density and gas temperature, appear to have the greatest variance, although there are still large peaks
2. most points are clustered at the lower or upper boundaries of dimensions 3, 4, 5 and 6. The respective histograms have massive peaks at either end of the distribution
3. most points are clustered at the lower boundary of dimensions 7, 8, 9 and 10

Therefore, the astrophysics dataset is highly skewed because most points are clustered at the boundaries of a dimension, and the data space is *sparse* because the majority of it has little to no points.

8.2 Effects of Data Distribution

The effects of the astrophysics dataset's distribution on the Pyramid Tree and *kd*-Tree is explored in this section.

8.2.1 Pyramid Tree

Recall that the Pyramid Tree maps a point v to a scalar value, called the point's pyramid value. Points with the same pyramid value are stored in the same bucket. Table 8.1 shows bucket size statistics and how long it takes to query all the points for the sampled astrophysics dataset with different subsets of dimensions. When the dimensions with the least skew (dimensions 1 and 2) are used, average bucket size decreases substantially. Dimensions 3 and 7 have large clusters of points at the boundaries, so they cause the average bucket size to increase. The combination of each individually sparse dimension creates a multi-dimensional data space which even sparser. This is the reason the Pyramid Tree produces so few buckets and performance is good poor when using all dimensions.

The histograms from Section 8.1 illustrate how most points lie on, or close to, the boundaries of the dataset. This is a significant problem for the Pyramid Tree because it always chooses the dimension whose distance from the centre point of the space is the highest. If a large number of points are on a dimensional boundary, then it is likely they are in the same pyramid. Each point will have the same coordinate value for the chosen dimension because they are on the same boundary, meaning they will also have the same pyramid height. Therefore, it is likely many points will be in the same bucket.

Histograms are equivalent to nearest-neighbour interpolation, so they do not show if points inside the bins at the boundary of the histograms are actually boundary values or just *close* to the boundary.

		Bucket Statistics		
Dimensions	**Time to Query (sec)**	**Average**	**Max**	**#Buckets**
All	60.0216	3586.57	235260	120
1 and 2	0.0713558	6.52433	102	25232
3 and 7	8.30587	89.2853	141235	3056
No Boundary Coordinates	27.6034	25.3722	45031	16963

Table 8.1: Pyramid Tree Bucket Size Statistics with Different Dimensions of Astrophysics Dataset

Dimensions	Time to Query (sec)	Balance Factor	Max Path Length
All	0.639265	32.405	120
1 and 2	0.18445	26.5923	73
3 and 7	0.257211	28.6481	69

Table 8.2: Point *kd*-tree Statistics with Different Dimensions of Astrophysics Dataset

To determine if clusters of points at boundaries is truly the main cause of large buckets, a heuristic called **No Boundary Coordinates** was developed. Let v be a point and min_i and max_i be the minimum and maximum boundary values for dimension $i \in \{0, 1, ..., d-1\}$. Let j be the dimension v is furthest away from the centre point in. If $v_j = min_j$ or $v_j = max_j$, then a new dimension k is chosen, such that v_k is the *second* furthest coordinate from the centre point. If v_k is at the boundary, then the *third* furthest coordinate is chosen. This process is repeated until a coordinate which is not on a boundary is found. If all coordinates are on a boundary, then the first dimension is chosen.

Table 8.1 shows how this simple heuristic decreases average bucket size. However, it is still significantly slower than the *kd*-tree, so boundary points are not the only problem with the dataset. Note that creating heuristics like this requires pre-existing knowledge of the point distribution, which may not be available, especially when the data is dynamic (i.e. all the points are not known in advance).

The histograms for the three dimensions of the armadillo mesh are given in Appendix D.5. The histograms show that the mesh's points are much more uniform than the astrophysics dataset. The Pyramid Tree performs substantially better with the armadillo mesh, showing that the structure is efficient with uniform distributions.

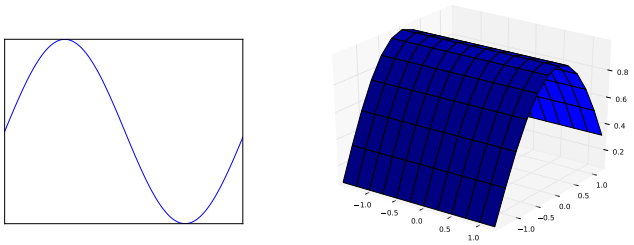
8.2.2 *kd*-Tree

Table 8.2 shows point *kd*-tree query time, balance factor and maximum path length of the tree with different dimensions of the astrophysics dataset. It can be observed that the point *kd*-tree, like the Pyramid Tree, performs worse because of the skew present in the astrophysics dataset. Using dimensions 1 and 2 gives a lower balance factor than the more skewed dimensions 3 and 7.

The performance difference between each dimension subset is much smaller than the Pyramid Tree, again showing that the point *kd*-tree is more resilient to skew than the Pyramid Tree.

8.3 Conjecture on Scientific Datasets

While an analysis of the hurricane Isabel dataset is not in this report, generating histograms of each dimension reveals distributions similar to the astrophysics dataset. Some dimensions contain large clusters of points at the boundaries, while others have smoother distributions. This raises an important question: do datasets resulting from scientific simulations generally have these properties, or are these properties just specific to the evaluation datasets? If the former is true, more general hypotheses can be made regarding which index structures are suitable for scientific computation and visualisation.



(a) $\mathbb{R}^1 \rightarrow \mathbb{R}^2$ (b) $\mathbb{R}^2 \rightarrow \mathbb{R}^3$

Figure 8.3: Manifolds Embedded in Higher-Dimensional Space

Exploring the properties of scientific multi-dimensional datasets is a project in itself, so an in-depth analysis of such properties is not performed here. Instead, a conjecture regarding point distributions of scientific datasets will be given. The reasoning behind this conjecture will then be discussed.

CONJECTURE: Highly clustered and sparse data spaces is a property held by the majority of scientific datasets computed using sampled lattices.

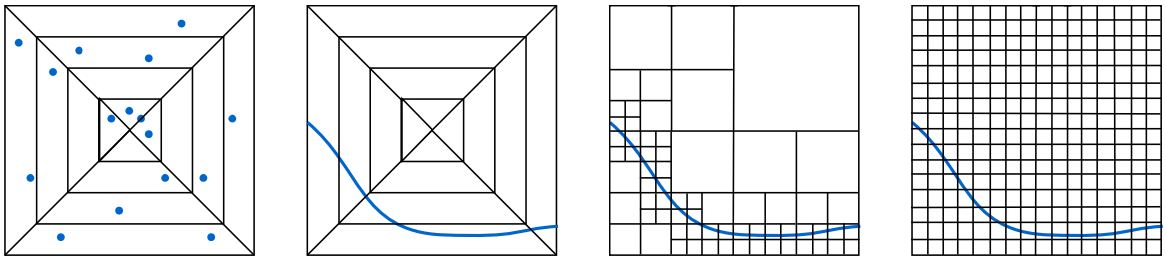
Scientific datasets are often computed using m-dimensional sampling lattices, where each point on the lattice represents a point in physical space. The astrophysics dataset was computed in this way. [92]. m is usually 2 or 3, because *physical* phenomena is typically being simulated. d continuous fields are measured at each point, which are typically physical properties such as temperature, wind velocity, chemical masses and so on. Interpolation is applied to model the *spatial* relationships between the sampling points in physical space.

With the astrophysics and hurricane Isabel datasets, d is larger than m. These physical simulations can be described as a mapping $\mathbb{R}^m \rightarrow \mathbb{R}^d$, where $d > m$. As d increases, the data space's volume increases exponentially, resulting in greater sparsity (as discussed in Section 3.2.1).

Data sparsity becomes an even greater issue when mapping continuous domains to higher dimensional continuous ranges. The resulting d-dimensional dataset is a sample of an m-manifold in d-dimensional space. Figure 8.3 illustrates examples of manifolds for $\mathbb{R}^1 \rightarrow \mathbb{R}^2$ and $\mathbb{R}^2 \rightarrow \mathbb{R}^3$. The data spaces in both examples are mostly empty. Only thin strips of the data spaces are filled by the lower-dimensional manifolds. The astrophysics dataset was the result of a $\mathbb{R}^3 \rightarrow \mathbb{R}^{10}$ simulation, so a 3 dimensional manifold is embedded in 10 dimensional space. The resultant 10D space is highly sparse, which would explain why there are huge clusters in the astrophysics dataset.

8.4 Implications to Multi-dimensional Search

This conjecture has implications for the suitability of classes of index structures for scientific datasets. If most scientific datasets are highly sparse, then they are difficult for index structures which decompose the underlying data space *without* adapting to point distribution. For example, the Pyramid Tree



(a) Pyramid Tree (Static) (b) Pyramid Tree (Static) (c) Octree (Adaptive) (d) Multigrid Tree (Static)

Figure 8.4: Various Spatial Decompositions of Non-Continuous 2D Dataset and 1-Manifold in 2D Space. Each Polygon Represents a Bucket

uses a *static* decomposition. Based on an initial boundary, exactly how the space is decomposed is already decided. This is was not a problem for the synthetic datasets used in Chapter 7. However, there are cases where the lack of ability to *adapt* the decomposition to the input data becomes problematic. This was the case with the astrophysics dataset, where the Pyramid Tree maps most points to the same few buckets.

Figure 8.4a shows a dataset computed using the mapping $f : \mathbb{R}^1 \rightarrow \mathbb{R}^2$. Unlike the astrophysics dataset, f's range is discontinuous (i.e. is not a 1-manifold). This would be the case for the synthetic data randomly generated for performance analysis in Chapter 5. In this particular example, the points are more evenly distributed across the space.

Figure 8.4b shows a 1-manifold. Here, the Pyramid Tree stores points on the manifold in just five buckets. Without the ability to adapt, the performance of the structure will rapidly degrade as more and more points are mapped to the same few buckets.

Figure 8.4c shows an Octree decomposition, which adapts the decomposition to provide greater discrimination between points in dense regions of space. The point *kd*-tree can be seen as an adaptive index structure, since decomposition of the underlying data space depends on the values of the given points. This is one of the reasons its performance is vastly superior to the Pyramid Tree with the two scientific datasets. Therefore, static decomposition may not be suitable for highly sparse distributions.

8.4.1 Case Study: Multigrid Tree

To further support the claim that static decompositions are not suitable for highly sparse data, we turn to the Multigrid Tree structure described in Appendix G. To summarise, this index structure splits the entire data space into equal sized cells and hashes points to buckets based on the cell they are contained in. Figure 8.4d shows a Multigrid Tree decomposition of the 1-manifold discussed earlier.

The Multigrid Tree essentially quantises continuous domains into discrete ranges which are used to hash points reliably (unlike Bit Hash) to a one-dimensional domain. The ultimate goal is to achieve perfect hashing so query times become closer to $O(d)$. The smaller the cells are made, the greater the discrimination between points there is and the smaller average bucket size will be. Theoretically, all

# Cells	Average Bucket Size	Total Buckets
100^{10}	1149.43	87
$1,000^{10}$	168.919	592
$10,000^{10}$	26.5252	3,770
$100,000^{10}$	10.7181	9,330
$1,000,000^{10}$	9.10581	10,997
$1,100,000^{10}$	9.06454	11,032
$10,000,000^{10}$	9.06454	11,032

Table 8.3: Number of Cells Used for Multigrid Tree Index Structure and its Effect on Bucket Size with 100,000 Astrophysics Dataset Points

one has to do to make this structure work efficiently with the astrophysics dataset, or any dataset with dense clusters in sparse data spaces, is to keep decreasing cell size until each point is in its own cell.

On real machines however, this does not work. When cells become small enough, rounding errors occur because of limited precision floating point arithmetic. Points start being mapped to the same cell even if they are conceptually in different cells because of these rounding errors. In other words, there is a limit to how small cells can be. The astrophysics has a highly clustered distribution, with very small magnitudes in some dimensions (e.g. 10^{-15}). This means extremely small cell sizes are required to achieve perfect hashing and rounding errors appear very quickly.

Table 8.3 shows the effect of cell size on average bucket size when storing 100,000 points from the astrophysics dataset. It shows that, as you increase the number of cells, the average bucket size decreases. When the number of cells exceeds a certain number, which is approximately $1,100,000^{10}$ for this dataset, no additional benefit is gained because the average bucket size does not change. This shows there is a limit to how much skew static decompositions can handle, because there is a limit on how granular the decomposition can be made.

On the other hand, adaptive decompositions can "zoom in" on dense regions of points and ignore sparse regions, instead of increasing the granularity of the decomposition across the *entire* data space. This allows greater discrimination to be achieved before running into floating point error.

8.5 Suitability of Hash-Based Approaches

The Pseudo-Pyramid Tree and Pyramid Tree both produce a static decomposition of the underlying data space through the use of dimension reduction. It was shown in the empirical results from this project that the point query performance of these two structures is inferior to the point *kd*-tree, a simple and well-established technique. These empirical results coincide with the theoretical discussion on the suitability of static decompositions for highly skewed datasets.

Adaptive variants of the Pyramid Tree exist, such as the Extended Pyramid-Technique [27] and iMinMax(θ) (see Appendix G). However, the *cost* of adapting such structures limits their usefulness. For example, the Extended Pyramid-Technique moves the centre of the data space to be the median

of all stored points. Doing so means that the entire structure must be rebuilt because each stored point may be hashed to a different value after the centre point has moved. For large n this has a massive impact on performance. An adaptable iMinMax(θ) was implemented in this project, but its performance was poor, despite it adapting to the dataset, because of how long rebuilding takes.

The empirical performance timings has lead this project to conclude that techniques which perform a static decomposition of the data space are *not* suitable for dynamic data, unless one knows about the point distributions that will be given in advance. The supplementary theoretical analysis performed in this chapter has shown that data generated by mapping continuous domains to higher-dimensional ranges are especially a challenge for structures that cannot adapt, due to the highly clustered and skewed distributions of points that result from such a mapping. Since the cost of adapting dimension reduction techniques is so high, it follows that most dimension reduction are likely to be unsuitable for dynamic or scientific datasets.

Despite the adaptive point *kd*-tree performing worse on highly skewed data, the structure's performance is superior to the others explored in this project. Index structures that adapt to point distributions *as* points are inserted, like the point *kd*-tree, do not require an expensive rebuild procedure.

The point *kd*-tree shows that these structures can provide good performance with point queries. Existing literature often discusses how *kd*-trees degenerate to Sequential Scan with higher dimensions, but this is not the case for point queries, even with real datasets that are highly skewed.

8.6 Conclusion

The original hypothesis, discussed in Section 6.5, stated that the Pyramid Tree would provide faster point queries for high-dimensional scientific datasets. The empirical evidence from Chapter 7 has shown this hypothesis is false. The discussion in this chapter has produced four new hypotheses:

HYPOTHESIS 1: Most existing index structures based on reduction to a 1D value statically decompose space, and degenerate when processing highly skewed data because there is a physical limit on how granular the decompositions can be made.

HYPOTHESIS 2: Most dimension reduction techniques cannot adapt well to skewed distributions because the cost of rebuilding the structure when adapting is too high.

HYPOTHESIS 3: Dynamic tree-based structures, which adapt the decomposition of the data space as points are inserted, provide better point query performance for high-dimensional scientific datasets than dimension reduction techniques.

HYPOTHESIS 4: The Pyramid Tree is not a suitable index structure for performing point queries on skewed datasets.

The empirical evidence verifies hypothesis 4 – the Pyramid Tree is not suitable for skewed datasets. The empirical evidence from this project, paired with the additional discussion in this chapter, strongly supports hypotheses 1, 2 and 3. However, more research is required to verify these hypotheses.

Chapter 9

Project Conclusion

This section will discuss the limitations of the project's work and propose future work to support or extend the project's findings. The project's methodology, requirements and final deliverables will then be discussed to measure the project's success.

9.1 Limitations

This project has only worked with three synthetic datasets and three real datasets. Only two of these datasets represent the kind of data this project is targeting, which is data from scientific computation. Using more data from a scientific domain and performing further analysis on their properties may have derived more insight into which index structures are suited to such data.

A large chunk of the Design and Implementation phase was spent focusing on the Pseudo-Pyramid tree and Pyramid Tree before they were found to perform poorly with point queries on the scientific datasets. By the time the limitations of these structures were discovered, there was not enough time to evaluate the other implemented structures in detail. Originally, it was intended that the Multigrid Tree and bucket *kd*-tree (see Appendix G) would be discussed and evaluated alongside the other index structures, but the remaining time did not allow for this.

Exploring these additional index structures, as well as acquiring more scientific datasets to characterise and use for evaluation, would have meant the project could explore the hypotheses made at the end of Chapter 8 in greater detail.

9.2 Future Work

This project ended with a conjecture and several hypotheses. Potential future work includes exploring these in greater detail. This section suggests ways these hypotheses can be explored further.

To further explore the conjecture that most scientific datasets have dense clusters of points in sparse data space, more scientific datasets could be examined in detail. The volume of empty space in

these datasets could be measured to derive quantitative measurements of sparsity in scientific datasets. In fact, studying the mathematical challenges of $\mathbb{R}^m \rightarrow \mathbb{R}^d$ where $d > m$, is a whole project in itself.

Exploring Hypothesis 2 would involve implementing and evaluating several adaptive dimension reduction techniques, perhaps using the same test datasets as this project. Examples of techniques to research include iDistance [101], iMinMax(θ) and the Multigrid Tree (see Appendix G).

Perhaps the most valuable finding is the superior performance of a tree-based index structure for the target data. Hypothesis 3 states dynamic tree-based structures are more suitable for scientific datasets than dimension reduction techniques for point queries. Verifying this involves implementing several dynamic tree-based structures and evaluating their performance on scientific datasets. Examples of such structures include bucket *kd*-trees and BD-trees [46], as well as the KDB-tree, quadtreap and the splay quadtree structures described in Sections 3.3.2 and 3.3.5.

Finally, embedding, distance-based or parallel structures (all briefly mentioned in Chapter 3) are avenues to explore. Based on the literature review, little research has been done on the suitability of these structure types for point queries on dynamic high-dimensional data.

9.3 Evaluation of Methodology

The project and evaluation methodology will be evaluated in this section. The majority of the project's schedule was sequential, having one major task following another. An exception to this was the Design and Implementation phase, which was iterative. There were multiple iterations of the same sub-process, which encompassed design, implementation and evaluation.

It is felt that this hybrid approach worked well. At the start of the project, the author had almost no knowledge about multi-dimensional search. It required several weeks of background research just to determine the scope and objectives of the project. Synthesising this research into a mid-project report and mid-project presentation took a significant amount of time. Each of these tasks were dependent on the tasks before. Therefore, it made sense to have the initial part of the project serial.

Implementing and testing index structures was the most exploratory part of the project, which required flexibility. Making the Design and Implementation phase iterative provided this flexibility.

The evaluation at the end of each iteration allowed one to gather insight into the implemented structures' performance. After the first two iterations, it was found that the Pyramid Tree had poor performance with the target datasets, so the next iteration turned to entirely different kind of structure. This shows planning everything that should be implemented in advance would be a waste, since new findings would have quickly made these plans irrelevant. It was also a good decision to have the supervisor meetings at the start of each iteration. It allowed the supervisor to be informed of new findings as soon as possible and offer their advice on what the next course of action should be.

One aspect of the process that could have been improved was the amount of time left for writing the final report. While the schedule left four weeks to write the report, unexpected findings were uncovered. These findings required more time to evaluate and discuss, increasing the time it took to

write the final evaluation. Perhaps the schedule should have left another week as a buffer for such situations, which are to be expected in a research project. Overall, combining waterfall and iterative methodologies worked well for this project. It is felt that the correct project methodology was chosen.

The evaluation methodology stated that skewed and clustered synthetic data was tested on the structures. However, these datasets had no negative on the performance of the structures, unlike the real skewed datasets. This was because the synthetic datasets were simply not *skewed enough*. Datasets with higher skew and smaller clusters should have been used. If they were, the Pyramid Tree's weaknesses may have been caught much earlier in the project. Perhaps more time should have been spent working on the evaluation methodology before rushing into performance analysis.

9.4 Objectives and Minimum Requirements

This section discusses if the project have met its objectives. Sections 1.2 and 1.3 list the project's objectives and minimum requirements. The minimum requirements were developed so that meeting them also meets the project's objectives. For brevity, the minimum requirements will be referred to by their numbers.

The literature review present in Chapter 3 was a comprehensive survey of the field. The challenges of multi-dimensional search were summarised and most classes of index structures were discussed in some way. A taxonomy illustrating the development of index structures was also produced (see Appendix D). Therefore, requirement 1 has been met.

Requirement 2 was met and largely exceeded; a total of nine structures were implemented in the project. Six of these implementations were discussed and analysed in the report. Due to time constraints, the remaining three structures could not be discussed and evaluated in detail. These are described in Appendix G. One of the proposed extensions was developing a new index structure, which was achieved with the Multigrid Tree described in Appendix G.

Requirement 3 was met and exceeded by performing several optimisations across all index structures, with special attention given to base implementation used by all three hash-based structures considered in this project. These optimisations considered cache coherency and SSE parallelisation.

Evaluation was present throughout the Chapters 7 and 8, meaning Requirement 4 has been met. This requirement was exceeded by performing further evaluation in Chapter 8, which targeted a specific scientific dataset. This concluded with several hypothesis related to the suitability of different classes of index structure to scientific datasets, exceeding the original requirements of just producing a performance evaluation. Therefore, it is felt requirement 4 has also been exceeded.

All the minimum requirements were met and exceeded to varying degrees. In particular, the amount of optimisations and index structures that were implemented and evaluated was much larger than intended. This was done to ensure the evaluation was as comprehensive as possible within the project time frame.

9.5 Deliverables

The deliverables described in Section 1.4 were produced in the following ways:

1. Source code of the evaluation framework and all implemented structures, along with documentation, has been submitted alongside this report
2. mdsearch, a C++ library containing implementations of some of the index structures explored in the project, has been released to the public. Where to get mdsearch, as well as documentation describing what features mdsearch provides and how to use them, is available in Appendix H.
3. An evaluation of the index structures was performed as part of Requirement 4
4. Generated synthetic data can be re-produced using the data generators provided in the source code of Deliverable 1

9.6 Conclusion

To conclude, this project implemented several index structures and evaluated their performance on dynamic high-dimensional data, specifically focusing on scientific datasets. It was found that the Pyramid Tree and a similar structure, the Pseudo-Pyramid Tree, consistently gave poor point query performance with scientific datasets. A well-established structure, the point *kd*-tree, outperformed these two structures by a large margin. An evaluation of these structures' behaviour with the astrophysics dataset was performed, in which some conjectures and hypotheses were derived regarding the suitability of different classes of index structures for dynamic scientific datasets. Future work has been proposed to verify these hypotheses and explore the discussed issues in greater detail.

The overall aim of the project was to implement one or more index structures and evaluate their performance with respect to high-dimensional datasets. Using the astrophysics dataset as a test case, the evaluation went into much greater detail than initially expected. A greater number of index structures and optimisations were implemented than planned, resulting in a more comprehensive evaluation. This evaluation resulted in a conjecture and several hypotheses being made, which has created a foundation to start future work in this area. Therefore, the aim of the project has been met and the project is considered a success.

Bibliography

[1] About Python. http://www.python.org/about/. Accessed: 2014-02-24.

[2] B. W. Kernighan and D. Ritchie. *The C Programming Language (Second Edition)*. Prentice Hall, 1988.

[3] B. Stroustrup. *The C++ Programming Language*. Pearson Education, 2013.

[4] The Haskell Programming Language. http://www.haskell.org/haskellwiki/Haskell. Accessed: 2014-02-24.

[5] S. Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly, 2013.

[6] The Problem with C++. http://www.codinghorror.com/blog/2007/01/the-problem-with-c.html. Accessed: 2014-02-24.

[7] J. Østergaard. C versus C++. http://unthought.net/c++/c_vs_c++.html, 2004. Accessed 2014-04-22.

[8] When to use C over C++, and C++ over C? http://programmers.stackexchange.com/questions/113295/when-to-use-c-ov%er-c-and-c-over-c. Accessed 2014-04-22.

[9] Robert Hundt. Loop recognition in c++/java/go/scala. In *Proceedings of Scala Days 2011*, 2011.

[10] CMake - Cross Platform Make. http://www.cmake.org/. Accessed: 2014-02-24.

[11] D. Huizinga and A. Kolawa. *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, 2007.

[12] P. Hamill. *Unit Test Frameworks: Tools for High-Quality Software Development*. O'Reilly, 2004.

[13] cppunit. http://www.freedesktop.org/wiki/Software/cppunit/. Accessed: 2014-02-24.

[14] nullunit - C++ Unit Testing Framework. http://code.google.com/p/nullunit/. Accessed: 2014-02-24.

[15] googletest - Google C++ Testing Framework. https://code.google.com/p/googletest/. Accessed: 2014-02-24.

[16] M. Mason. *Pragmatic Version Control: Using Subversion*. Pragmatic Bookshelf, 2006.

[17] Git. http://git-scm.com/. Accessed: 2014-02-24.

[18] D. Bulk and D. Mayhew. *Efficient C++: Performance Programmning Techniques*. Addison-Wesley Professional, 2000.

[19] Intel VTune Aplifier XE 2013. http://software.intel.com/en-us/intel-vtune-amplifier-xe. Accessed: 2014-02-25.

[20] N. Nethercote and J. Steward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices (42.6)*, 2007.

[21] profiny - Lightweight C++ Profiler. https://code.google.com/p/profiny/. Accessed: 2014-02-25.

[22] gperftools - Fast, multi-threaded malloc() and nifty performance analysis tools. http://code.google.com/p/gperftools/. Accessed: 2014-02-25.

[23] M. Flynn. Some Computer Organizations and their Effectiveness. *Computers, IEEE Transactions on, 100(9)*, pages 948–960, 1972.

[24] Using Intel(R) Streaming SIMD Extensions and Intel(R) Integrated Performance Primitives to Accelerate Algorithms. http://software.intel.com/en-us/articles/using-intel-streaming-simd-ex%tensions-and-intel-integrated-performance-primitives-to-accelerate-algorit Accessed 2014-04-04.

[25] Y. Fu, J. Teng and S. R. Subramanya. Node Splitting Algorithms in Tree-Structured High-Dimensional Indexes for Similarity Search. *Proceedings of the 2002 ACM symposium on Applied computing*, pages 766–770, 2002.

[26] M. H. Overmars. *The Design of Dynamic Data Structures*. 1983.

[27] S. Berchtold, C. Böhm and H. Kriegal. The Pyramid-Technique: Towards Breaking the Curse of Dimensionality. *ACM SIGMOD Record 27.2*, pages 142–153, 1998.

[28] T. Sellis, N. Roussopoulous and C. Faloutos. The r+-tree: A dynamic index for multi-dimensional objects. 1987.

[29] A. Guttman. R-trees: A dynamic index structure for spatial searching. *ACM 14.2*, pages 47–57, 1984.

[30] W. Wang, J. Yang and R. Muntz. PK-tree: a spatial index structure for high dimensional point data. *Springer US*, pages 281–293, 2000.

[31] E. Park and D. M. Mount. A Self-adjusting Data Structure for Multidimensional Point Sets. *AlgorithmsESA 2012*, pages 778–789, 2012.

[32] A. Li, C. Zhang, J. Zhang and Z. Zhang. RSR-tree: A Dynamic Multi-dimensional Index Structure. *Journal of Computers 6.12*, pages 2552–2558, 2011.

[33] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[34] R. E. Bellman. *Dynamic Programming*. Rand Corporation, 1957.

[35] A. Zimek, E. Schubert and H. Kriegel. A survey on unsupervised outlier detection in highdimensional numerical data. *Statistical Analysis and Data Mining 5.5*, pages 363–387, 2012.

[36] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

[37] R. Weber, H. Schek and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. *VLDB 28*, pages 194–205, 1998.

[38] O. Procopiuc, P. K. Agarwal, L. Arge and J. S. Vitter. Bkd-tree: A Dynamic Scalable kd-tree. *Advances in Spatial and Temporal Databases*, pages 46–65, 2003.

[39] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. *Introduction to Algorithms (Third Edition)*. MIT Press, 2009.

[40] D. D. Sleator and R. E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM (JACM) 32.3*, pages 652–686, 1985.

[41] A. Li and Z. Zhang. RS-tree: A dynamic index structure. *Information Engineering and Computer Science*, pages 1–4, 2009.

[42] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys (CSUR) 16.2*, pages 187–260, 1984.

[43] R. Seidel and C. R. Aragon. Randomized Search Trees. *Algorithmica 16.4-5*, pages 464–497, 1996.

[44] R. A. Finkel and J. L. Bentley. Quad Trees A Data Structure for Retrieval on Composite Keys. *Acta Informatica 4.1*, pages 1–9, 1974.

[45] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM 18.9*, pages 509–517, 1975.

[46] S. P. Dandamudi and P. G. Sorenson. An Empirical Performance of Some Variations of the k-d Tree and BD Tree. *International Journal of Computer & Information Sciences 14.3*, pages 135–159, 1985.

[47] J. T. Robinson. The KDB-tree: a search structure for large multidimensional dynamic indexes. *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18, 1981.

[48] H. Fuchs, Z. M. Kedem and B. F. Naylor. On visible surface generation by a priori tree structures. *ACM Siggraph Computer Graphics 14.3*, pages 124–133, 1980.

[49] M. Berg and A. Khosravi. On visible surface generation by a priori tree structures. *25th European Workshop Comput. Geom (EuroCG 2008)*, pages 255–258, 2008.

[50] D. Eppstein, M. T. Goodrich and J. Z. Sun. Skip quadtrees: Dynamic data structures for multidimensional point sets. *International Journal of Computational Geometry and Applications 18.01n02*, pages 131–160, 2008.

[51] S. Aluru and F. E. Sevilgen. Dynamic compressed hyperoctrees with application to the N-body problem. *Foundations of Software Technology and Theoretical Computer Science*, pages 21–33, 1999.

[52] K. Lin, H. V. Jagadish and C. Faloutos. The TV-Tree: An Index Structure for High-Dimensional Data. *VLDB Journal 3*, pages 517–542, 1994.

[53] Y. Sakurai, M. Yoshikawa and S. Uemura and H. Kojima. The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation. *VLDB 2000*, pages 5–16, 2000.

[54] D. Comer. The Ubiquitous B-Tree. *Computing Surverys 11.2*, pages 121–137, 1979.

[55] R. Bayer. Symmetric Binary B-Trees Data Structure and Maintenance Algorithms. *Acta Informatica 1.2*, pages 290–306, 1972.

[56] M. Freeston. A General Solution of the n-Dimensional B-tree Problem. *ACM SIGMOD Record 24.2*, pages 80–91, 1995.

[57] N. Beckmann, H. Kriegel, R. Schneider and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *ACM 19.2*, pages 322–331, 1990.

[58] D. A. White and R. Jain. Similarity Indexing with the SS-tree. *Proceedings of the Twelfth International Conference*, pages 516–523, 1996.

[59] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. 1993.

[60] S. Berchtold, D. A. Keim and H. Kriegal. The X-tree: An Index Structure for High-Dimensional Data. *Readings in multimedia computing and networking*, 2001.

[61] P. Ciaccia, M. Patella and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. *Proceedings of the 23rd International Conference on Very Large Databases (VLDB)*, pages 426–435, 1997.

[62] G. H. Dunteman. *Principal Components Analysis Vol. 69*. Sage, 1998.

[63] C. Faloutos and K. Lin. FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets. *ACM 24.2*, pages 163–174, 1995.

[64] D. M. Mount and E. Park. A Dynamic Data Structure for Approximate Range Searching. *Proceedings of the 2010 annual symposium on Computational geometry*, pages 247–256, 2010.

[65] V. V. Muniswamy. *Design and Analysis of Algorithms*. Pearson Education India, 2010.

[66] K. Zhou, Q. Hou, R. Wang and B. Gao. Real-time KD-tree Construction on Graphics Hardware. *ACM Transactions on Graphics (TOG) 27.5*, pages 126–137, 2008.

[67] J. Fix, A. Wilkes and K. Skadron. Accelerating Braided B+ Tree Searches on a GPU with CUDA. *Proceedings of the 2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC)*, 2011.

[68] K. Kaczmarski. Experimental B+-tree for GPU. *ADBIS (2)*, 2011.

[69] K. Kaczmarski. B+-tree Optimized for GPGPU. *On the Move to Meaningful Internet Systems: OTM 2012*, pages 843–854, 2012.

[70] J. Kim, S. Hong and B. Nam. A Performance Study of Traversing Spatial Indexing Structures in Parallel on GPU. *High Performance Computing and Communication and 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on. IEEE*, pages 855–860, 2012.

[71] L. Luo, M. D. F. Wong and L. Leong. Parallel Implementation of R-trees on the GPU. *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific. IEEE*, 2012.

[72] J. Kim, S. Hong and B. Nam. Parallel multi-dimensional range query processing with R-trees on GPU. *Journal of Parallel and Distributed Computing 73.8*, pages 1195–1207, 2013.

[73] H. Yokota, Y. Kanemasa and J. Miyazaki. Fat Btree: An Update Conscious Parallel Directory Structure. *Data Engineering, 1999. Proceedings., 15th International Conference on. IEEE*, pages 448–457, 1999.

[74] E. Nardelli. Distributed k-d Trees. *Proceedings 16th Conference of Chilean Computer Science Society (SCCC96)*, pages 142–154, 1996.

[75] E. Nardelli, F. Barillari and M. Pepe. Distributed Searching of Multi-dimensional Data: A Performance Evaluation Study. *Journal of Parallel and Distributed Computing 49.1*, pages 111–134, 1998.

[76] R. Vuduc and S. Baghsorkhi. *Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU)*. Morgan and Claypool Publishers, 2012.

[77] G. Coulouris, J. Dollimore and G. Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley, 2011.

[78] I. Foster. *Designing and Building Parallel Programs*. AddisonWesley, 1995.

[79] H. Nguyen. *GPU Gems 3*. Addison Wesley, 2007.

[80] B. Seeger and P. Larson. Multi-Disk B-trees. *ACM SIGMOD Record. Vol. 20.2*, pages 436–445, 1991.

[81] S. Pramanik and M. H. Kim. Parallel Processing of Large Node B-trees. *Computers, IEEE Transactions on 39.9*, pages 1208–1212, 1990.

[82] B. Schnitzer and S. T. Leutenegger. Master-client R-trees: A New Parallel R-tree Architecture. *Scientific and Statistical Database Management*, pages 68–77, 1999.

[83] I. Kamel and C. Faloutous. Parallel R-trees. *ACM SIGMOD 21.2*, pages 195–204, 1992.

[84] A. Gionis, P. Indyk and R. Motwani. Similarity search in high dimensions via hashing. 2004.

[85] R. Weber, H. Schek and S. Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. *VLDB (98)*, pages 194–205, 1998.

[86] P. Indyk. Similarity search in high dimensions via hashing. *VLDB (99)*, pages 518–529, 1999.

[87] C. Kingsford. kd-Trees. `https://www.cs.umd.edu/class/spring2008/cmsc420/L19.kd-trees.pdf`. Accessed 2014-04-16.

[88] A. F. Aviram. Interactive B+ Tree. `http://www.amittai.com/prose/bplustree.html`. Accessed 2014-04-16.

[89] P. Mattis. cpp-btree – C++ B-tree. `https://code.google.com/p/cpp-btree/`. Accessed 2014-04-16.

[90] R. E. Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods 6.2*, pages 306–318, 1985.

[91] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 8.1*, pages 3–30, 1998.

[92] D. Whalen and M. L. Norman. Competition Dataset and Description. *2008 IEEE Visualization Design Contest (`http://vis.computer.org/VisWeek2008/vis/contests.html`)*, 2008.

[93] W. Wang, C. Bruyere, B. Kuo and T. Scheitlin. IEEE Visualization 2004 Contest: Data Set. *2004 IEEE Visualization Design Contest (`http://sciviscontest.ieeevis.org/2004/data.html`)*, 2004.

[94] T. Ferguson. *A Course in Large Sample Theory*. Chapman and Hall/CRC, 1996.

[95] The Standford 3D Scanning Repository. `http://www-graphics.stanford.edu/data/3Dscanrep/`. Accessed 2014-04-17.

[96] B. Merry, J. Gain and P. Marais. Accelerating kd-tree searches for all k-nearest neighbours. *Eurographics 2013-Short Papers*, pages 37–40, 2013.

[97] The Single UNIX (R) Specification, Version 2 – clock_gettime. `http://pubs.opengroup.org/onlinepubs/007908799/xsh/clock_gettime.html`. Accessed 2014-05-12.

[98] Getting High Precision Timing on Android. `http://gamasutra.com/view/feature/171774/getting_high_precision_timing%_on_.php?print=1`. Accessed 2014-05-12.

[99] P. B. Callahan and S. R. Kosaraju. Decomposition of Multidimensional Point Sets with Applications to k-Nearest-Neighbors and n-Body Potential Fields. *Journal of the ACM (JACM) 42.1*, pages 67–90, 1995.

[100] H. Carr, B. Duffy and B. Denby. On Histograms and Isosurface Statistics. *Visualization and Computer Graphics, IEEE Transactions on 12.5* , pages 1259–1266, 2006.

[101] H. V. Jagadish, B. C. Ooi, K. Tan, C. Yui and R. Zhang. iDistance: An Adaptive B+-Tree Based Indexing Method for Nearest Neighbor Search. *ACM Transactions on Database Systems (TODS) 30.2*, pages 364–397, 2005.

[102] B. Jenkins. A Hash Function for Hash Table Lookup. `http://burtleburtle.net/bob/hash/doobs.html`. Accessed 2014-04-15.

[103] B. C. Ooi, K. Tan, Cui Yu and Stephane Bressan. Indexing the edges – a simple and yet efficient approach to high-dimensional indexing. *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 166–174, 2000.

[104] S. Maneewongvatana and D. M. Mount. Its okay to be skinny, if your friends are fat. *Center for Geometric Computing 4th Annual Workshop on Computational Geometry 2*, 1999.

Appendix A

Personal Reflection

During my time in industry last year, I worked on several projects end-to-end. Each project provided their own unique challenges, which drove me to improve both my technical and professional skills to the level required to deliver. This project was no exception.

I will admit that, after returning to university with several projects under my belt, I was not expecting the final year project to be as difficult as I found it. I had experience with gathering requirements, deciding project scope and deliverables and communicating the motivation and progress of such projects to invested parties. What separated this project from others I have done was the heavy focus on independent research and the amount of critical thinking required to make the necessary decisions. Multi-dimensional search is a huge research area, much bigger than I had imagined. I started the project with very little knowledge on the field, so even just deciding the scope of the project required a significant amount of time and thought. Every decision required me to think carefully and objectively. Is this the right data to use? Is this the right structure to implement? Is this a fair and unbiased evaluation approach?

The project had many ups and downs. There were times everything seemed to go right and times I made wrong decisions which had serious consequences on the project. A large chunk of time at the start of the project was spent planning all the work to be done throughout the three months of the project. Despite the time spent on the plan, many of this had to be rapidly adapted to react to unexpected change. During the middle of the project, the core hypothesis driving my actions was disproved by the results of an evaluation. I had to react quickly and make sure I used the knowledge gained from the evaluation to steer the project in another direction, developing a new plan. I found this very challenging to do. Nevertheless, the project as a whole was a satisfying experience and I have definitely learnt from it.

A.1 What I Have Learned

I have learnt much from this project, but the especially valuable lessons were:

- **Background Research** – I have learnt how to dive into a field I have very little knowledge about, find relevant literature in a methodical way and synthesise that literature into higher-level insights about the field.
- **Risk Management** – research projects generally involve high amounts of risks and I believe this project was no exception. This project's implementation phase in particular carried lots of risk – a structure could take longer to implement than expected, may not perform as well as expected or I might even fail to find the necessary information to implement the structure (e.g. the Splay Quadtree). I had to become better at planning for risk and mitigating its effects to ensure the project continued running smoothly.
- **Methodical Evaluation** – this is the first project that has required me to perform such an in-depth, rigorous evaluation. It required *thinking* about what I am doing and why at all times and a systematic approach to support this thinking. The project has has improved my ability to do this, by methodically evaluating the results of experiments and deriving new insights.
- **Technical Skills** – the sheer amount of algorithms reviewed, created and modified for this report has improved my algorithm design skills. Additionally, I have learnt how to write efficient code by considering cache coherency, memory management and how the underlying C/C++ code works.

A.2 What I Would Do Differently

The biggest mistake made in this project was not following a systematic and methodical approach. If a truly systematic approach was followed with the evaluation of the implementations, then the problems with the Pyramid Tree should have been caught some in the first two weeks of the Design and Implementation phase, not the fourth week. The discovery was made too late in the project, making it difficult to steer the project in a different direction in time.

Further mistakes were made after this discovery. Rather than perform an in-depth evaluation into *why* the Pyramid Tree had such bad performance straight away, other structures were explored. However, these were also hash-based structures, which as shown in the final technical evaluation in this report, are likely to always perform poorly with the target datasets. I should have had the discipline to stop and think about the reason for the poor performance of the Pyramid Tree, rather than rushing into implementing other structures without considering the wider implications of the findings. By the time I had realised this, it was late into the project.

A.3 Advice to Fellow Students

The main advice I'd give to students who are yet to start their final year projects, or anyone starting any kind of project, is to **not be afraid of making the wrong decisions**. One has to have the courage to make decisions, since that's the only way to achieve the desired outcome. Nearly everything worth doing involves some measure of risk and has the possibility of failure. I would even argue that most "right" decisions are found by making previously wrong decisions and understanding *why* they were wrong. Such is the nature of the scientific method.

I started the implementation phase of my project with a hypothesis I wanted to prove. Early test results matched my hypothesis. When this happened, I could have simply stopped the evaluation there and deemed my hypothesis to be correct. However, not satisfied my evaluation was comprehensive enough, I continued to perform more and more tests and eventually found out my hypothesis turned out to be wrong. Initially, I was frustrated that I spent so much time implementing something which ended up proving the exact opposite of what I wanted to find. However, I realised that these results, even if it was not what I wanted to find, are valuable. Losing the fear of making wrong decisions/hypotheses allows one to maintain skepticism. Never try and fit the data to a hypothesis or have a biased interpretation. In fact, try as hard as possible to break hypotheses; rigorous skepticism is the only way one can be truly confident in their work and defend it against criticism from others.

Another piece of advice I have is to **always think of the bigger picture**. Always remember what the project is actually trying to achieve, *why* this achievement is useful and where the current task fits into a greater context of the project and its domain. It is very easy to get caught up in small technical details about specific tools or implementations which do not move the project forward. Constantly re-evaluate where the project is, what work is currently being done and if that work is actually getting the project any closer to its goal. There were a couple of times where I spend days trying to implement certain functionality before taking a step back and re-evaluating the situation to determine if what I was implementing was truly worth the time investment. This is especially important since the project's timeframe is so short.

Finally, **be realistic about the project's goals**. Ambition causes people to force themselves into situations that will push them to learn more, which is good, but the project is short. Furthermore, large chunks of this short timeframe will be spent on background research and writing the mid-project/final reports. Therefore, students are not expected to develop the latest and greatest techniques/applications. It is getting experience in the *process* which is the valuable part. The process of diving into a new field, identifying where a contribution can be made, making a commitment on what to deliver and following through with that commitment, managing the inevitable changes that will occur throughout. Embracing this process will allow the project go much smoother and induce less stress.

A.4 Final Remarks

While the end results were not as efficient as I had wanted, I hope someone will find use in the project's deliverables and findings for their own work in the future. I greatly enjoyed this project, even if I probably spent far too many evenings in the lab. I found myself becoming highly interested in multi-dimensional search and I have only scratched the surface of it.

The project was challenging at times, but without those challenges it would not have been as satisfying to complete. I believe the project was repeatedly steered in the wrong directions because of poor decision making. These poor decisions were the result of not following a systematic approach rigorously. However, these mistakes allowed to me to learn a great deal from the project. Equipped with the lessons I have learnt, I would like to revisit the problem again someday!

Appendix B

Record of External Materials Used

B.1 Data

The real datasets used for evaluation in this project were retrieved from the following sources:

- **Astrophysics Turbulence Simulation** – from the IEEE Visualization 2008 Contest, which released a dataset generated by Whalen and Norman [92]
- **Hurricane Isabel Simulation** – from the IEEE Visualization 2004 Contest, which released a dataset generated by the National Center for Atmospheric Research in the United States [93]
- **Armadillo 3D Mesh** – from the Stanford 3D Scanning Repository [95]

B.2 Code

Aside from the C++ standard library and the Boost library, the only externally written source code was the original Pseudo-Pyramid tree. This was written by Zhao Geng[1], who is a research associate within the School of Computing at University of Leeds (at the time of writing).

[1] Z.Geng@leeds.ac.uk

Appendix C

Ethical Issues

There are no ethical issues involved with this project, since no private or personal data is being used in either the design, implementation or evaluation of the solution. All data used for evaluation is either synthetic data generated specifically for this project or datasets in the public domain.

Appendix D

Supplementary Material

Supplementary diagrams or tables mentioned in the main report are present here.

D.1 Schedule

This section contains diagrams which illustrate the project's schedule, milestones and overall process. See Section 2.1 for a description of the project schedule.

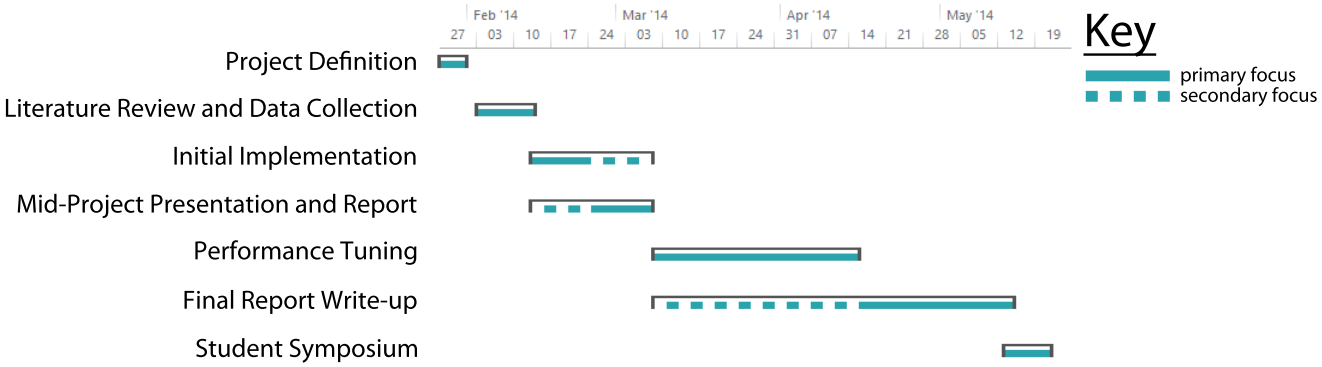


Figure D.1: Initial Project Schedule Created on 31/01/14

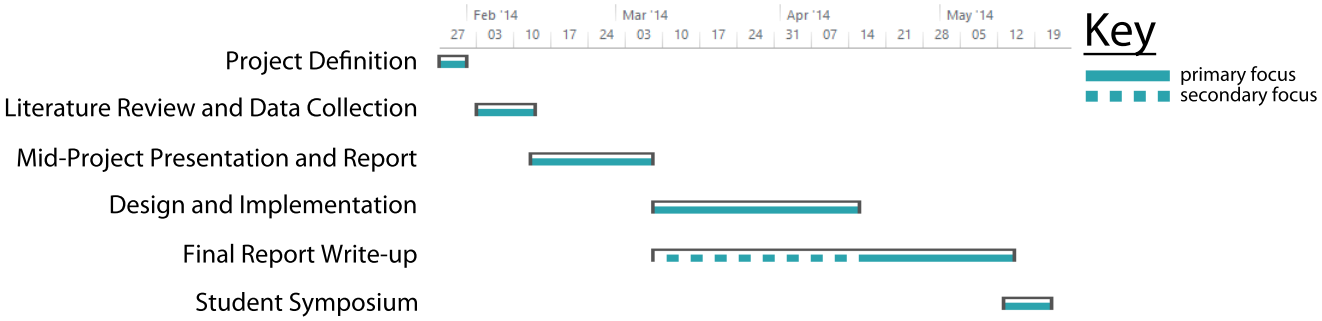


Figure D.2: Revised Project Schedule Created on 20/02/14

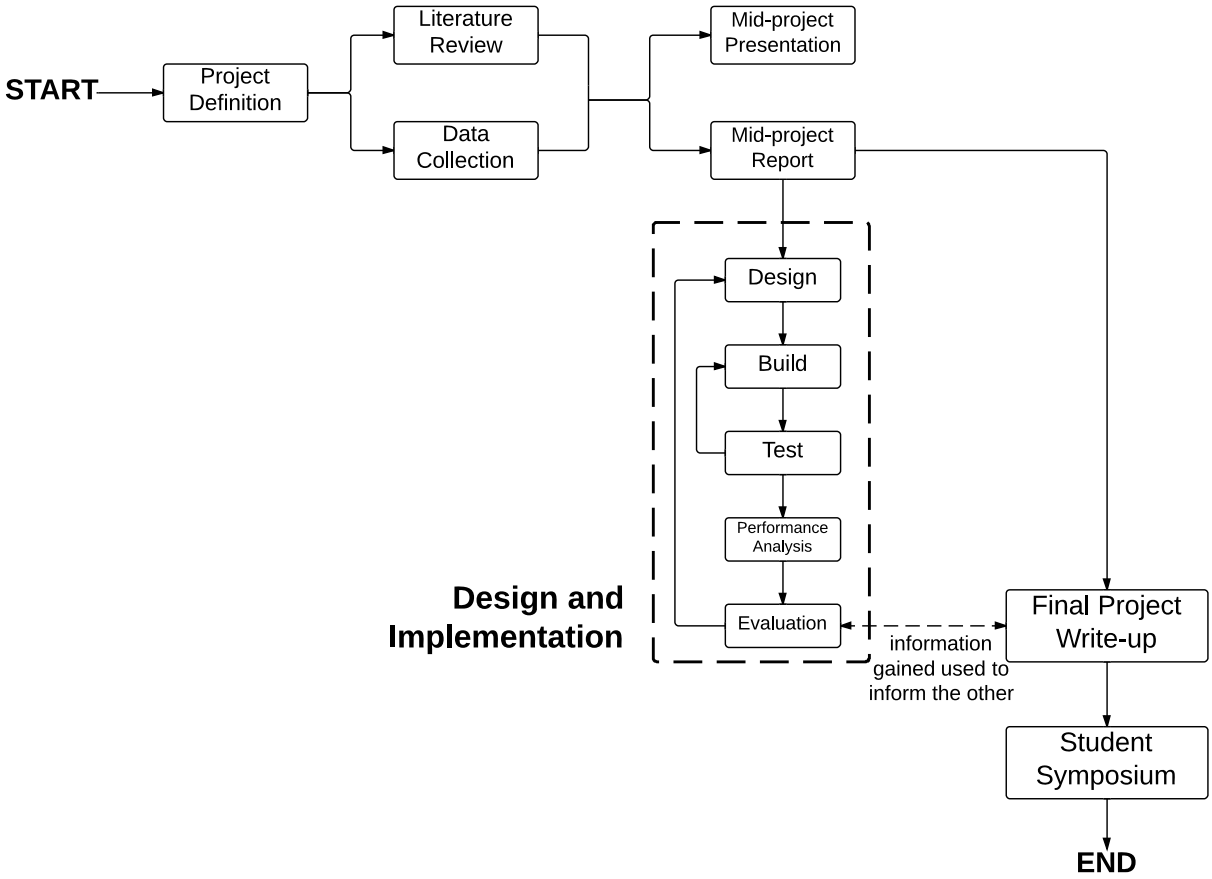


Figure D.3: Flow Chart of Full Project Schedule

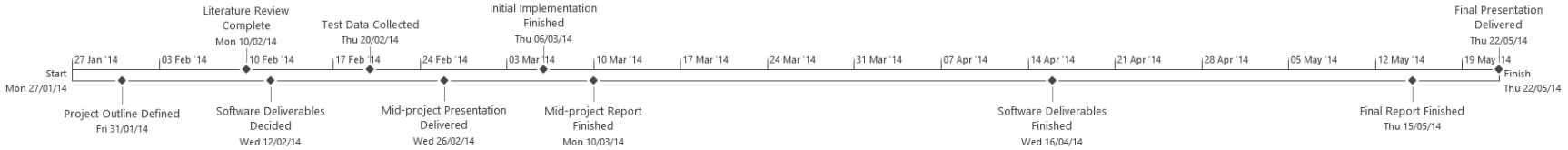


Figure D.4: Initial Milestone Timeline Created on 31/01/14

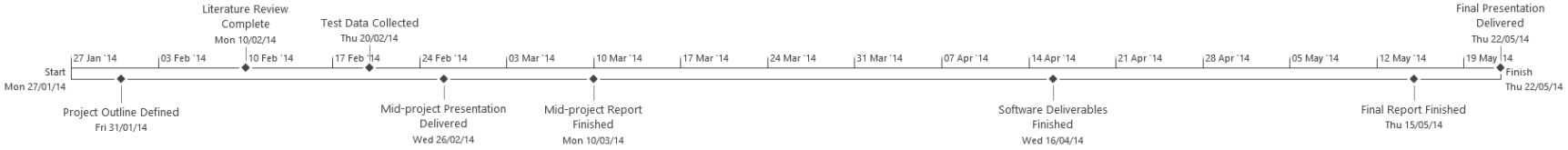


Figure D.5: Revised Milestone Timeline Created on 20/02/14

D.2 Multi-dimensional Search Structures

Supplementary information gathered from the background research in Section 3 is presented here.

Index Structure	Memory Overhead	Bucket Method?	High-Dimensional Data	Complexity
Sequential Scan	Low	No (but since data is stored contiguously, there are minimal I/O operations due to sequential access)	Often better than other structures with high d (but significantly poorer performance with low d)	Low
B^+-Tree	Low	Yes	One-dimensional	Low
kd-tree	Low	Some variants use buckets	Known to be poor with range and nearest neighbour queries with high d [86]	Low
R-Tree	Moderate	Yes	Poor for $d > 10$ [27]	Moderate
Quadtree	Low with uniformly distributed data, high for skewed data due to unnecessary nodes caused by splitting sparse regions of data space	No	Poor because it tries to use balanced splits [27]	Low
Pyramid Tree	Low	Yes (based on B^+-tree)	Good	Moderate
PK-Tree	Moderate	No (but uses similar method to reduce I/O operations)	Good	High
Skip Quadtree	Moderate	No	Untested	Moderate
Quadtreap	Low	No	Untested	High
Splay Quadtree	Moderate	No	Untested	High

Table D.1: Comparison of Dynamic Multi-Dimensional Structures

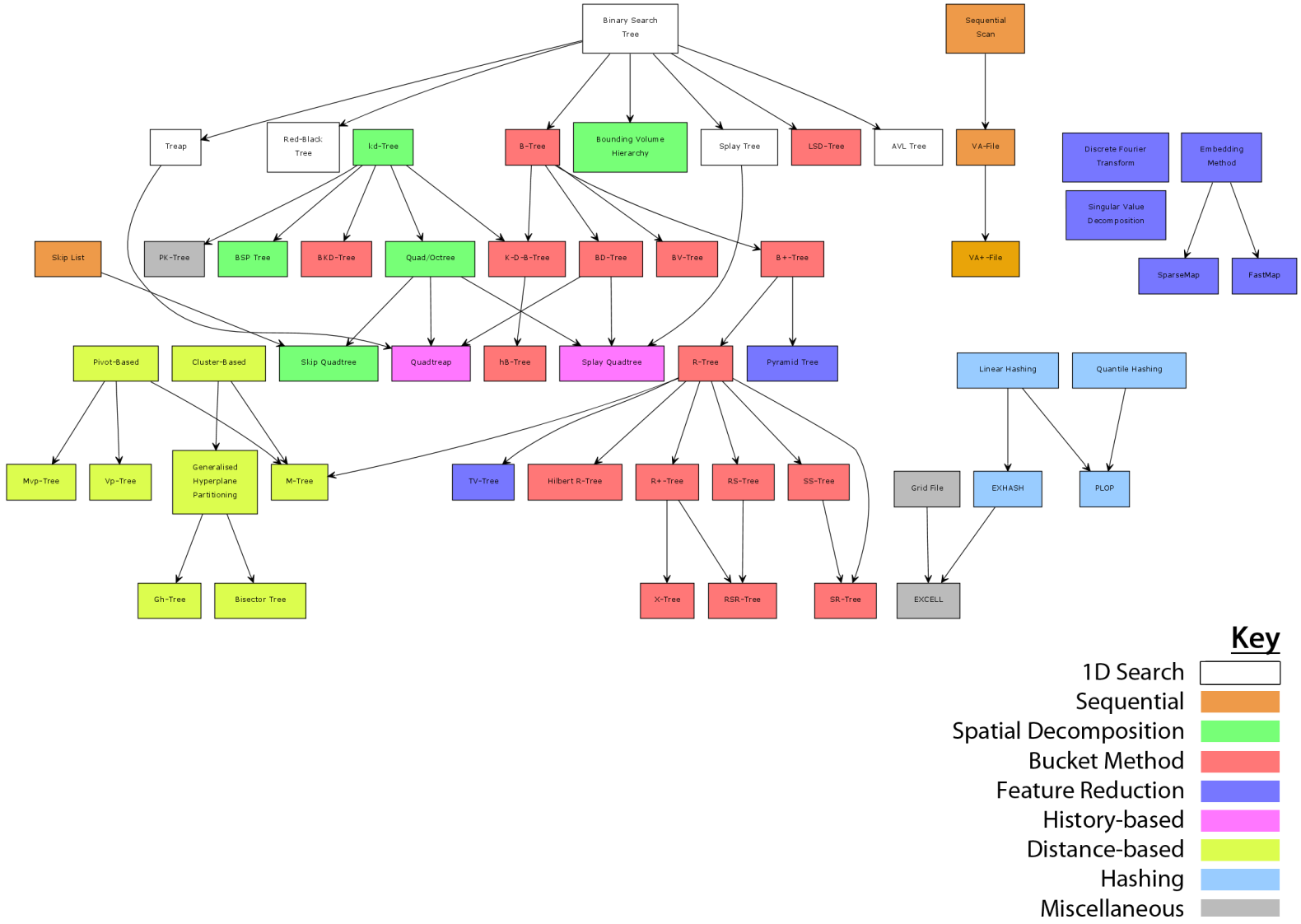
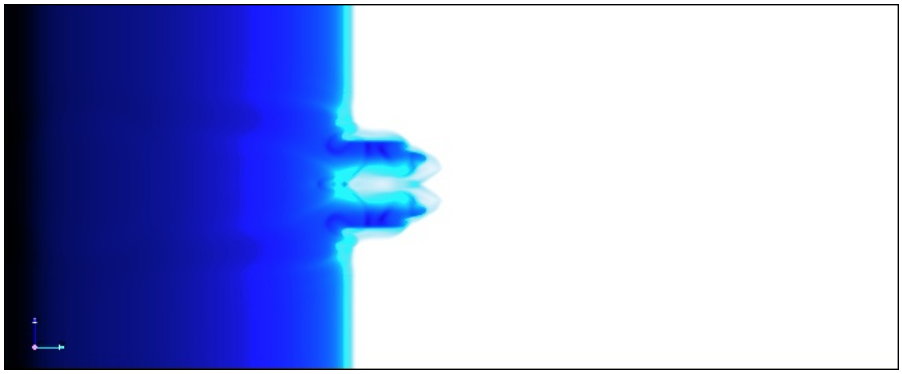


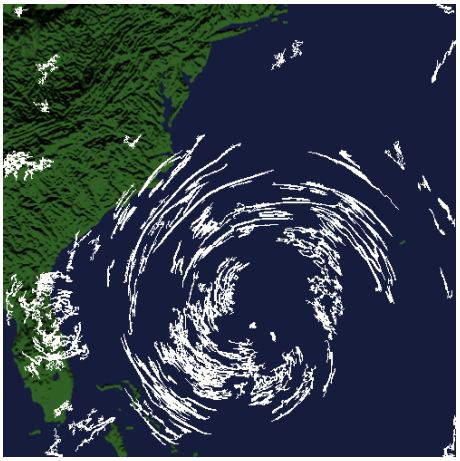
Figure D.6: Multi-dimensional Search Structure Taxonomy

D.3 Real Dataset Visualisation

Visualisations of three real datasets used for evaluation (described in Section 6.3) are shown here, to give more insight into the kind of information the datasets store and what they would be used for.



(a) Z-Slice of Timestep 30 of Astrophysics Dataset, Displayed Using Blackbody Radiation Spectrum [92]



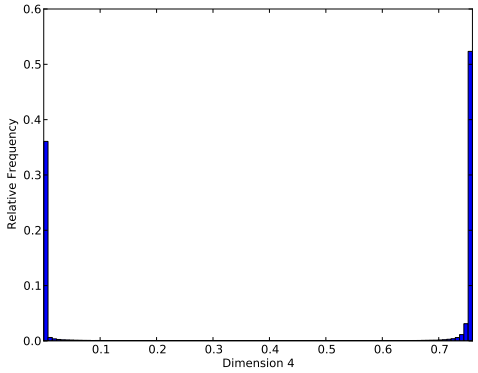
(b) Hurricane Isabel Simulation

(c) 3D Armadillo Mesh

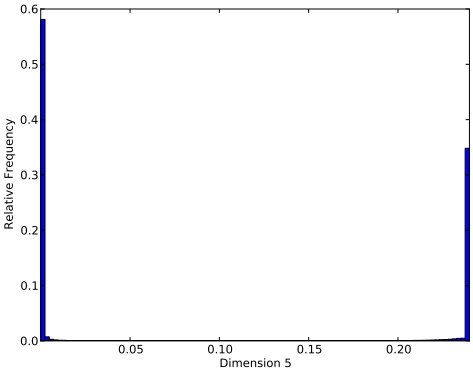
Figure D.7: Three Real Datasets Used For Evaluation

D.4 Astrophysics Histograms

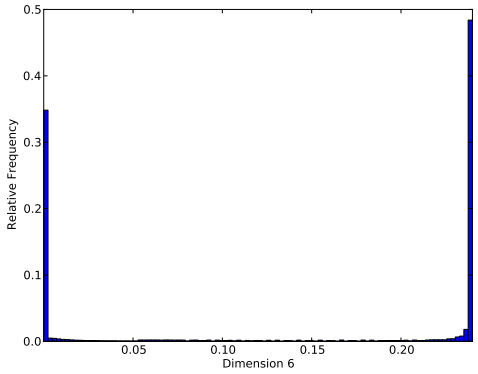
Section 8.1 analyses the skew of the astrophysics dataset using histograms. Histograms for dimensions 1, 2, 3 and 7 were shown in Section 8.1. The histograms of the remaining six dimensions are shown here for completeness.



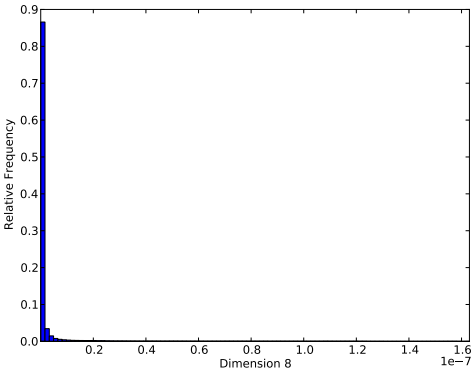
(a) Dimension 4 (H^+ mass abundance)



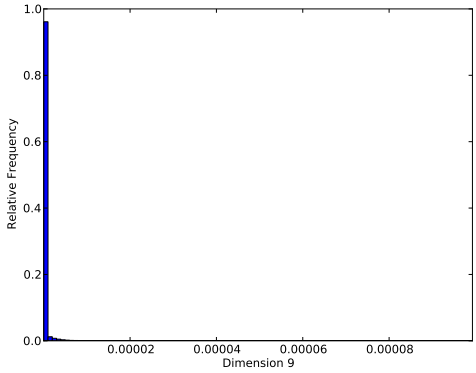
(b) Dimension 5 (He mass abundance)



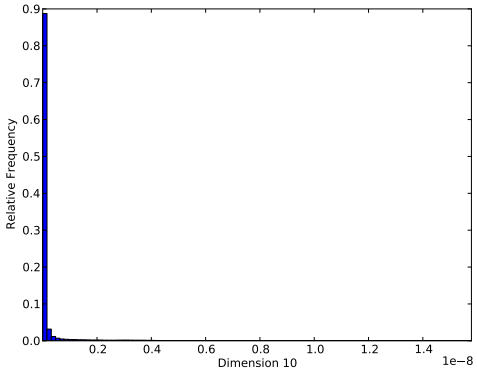
(a) Dimension 6 (He^+ mass abundance)



(b) Dimension 8 (H^- mass abundance)



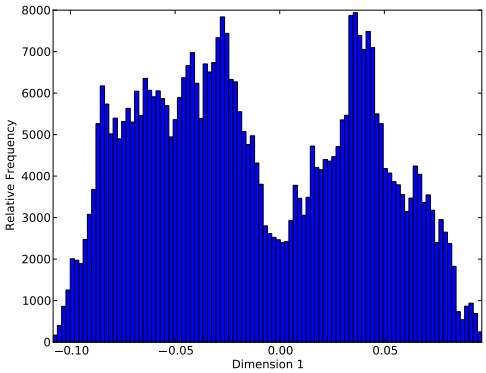
(a) Dimension 9 (H_2 mass abundance)



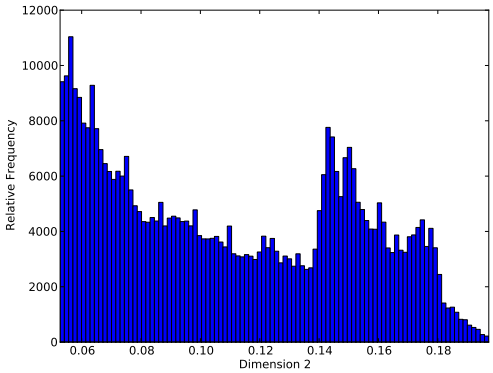
(b) Dimension 10 (H_2^+ mass abundance)

D.5 Armadillo Mesh Histograms

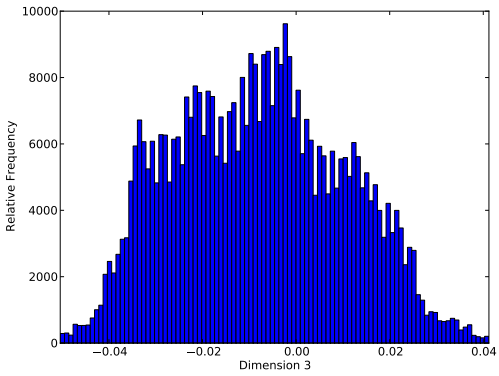
Section 8.1 mentions how the armadillo mesh's point distributions are smooth and relatively uniform. This is shown by the histograms of the mesh's dimensions in this section.



(a) Dimension 1



(b) Dimension 2



(c) Dimension 3

Appendix E

Performance Timings

Tables containing the execution times of each structure evaluated in Chapter 7 for all evaluation datasets are provided in this section. When "-" is shown instead of the number of seconds, it means that the performance test was not finish because machine running out of memory.

		Dimensions							
Structure	**Operation**	1	2	3	5	10	50	100	200
Sequential Scan	Delete	0.470472	0.592127	0.697993	0.698381	0.628591	0.869429	1.481980	3.310680
	Insert	0.084579	0.094937	0.085707	0.095868	0.111307	0.135492	0.144736	0.146832
	Point Query	0.083401	0.089991	0.084615	0.094284	0.110552	0.134050	0.143772	0.143288
Octree	Delete	0.036083	0.002966	0.022031	0.010735	0.141340	-	-	-
	Insert	0.003538	0.003241	0.003617	0.010450	0.266260	-	-	-
	Point Query	0.002353	0.002051	0.002198	0.005085	0.087902	-	-	-
Pseudo-Pyramid Tree	Delete	0.001257	0.001183	0.001355	0.003325	0.002860	0.003024	0.004981	0.008579
	Insert	0.003387	0.003402	0.003371	0.003488	0.003901	0.005354	0.006898	0.009913
	Point Query	0.000596	0.000650	0.000683	0.000741	0.000890	0.002223	0.003984	0.007371
Pyramid Tree	Delete	0.001028	0.001179	0.001287	0.001430	0.001613	0.003569	0.005926	0.010783
	Insert	0.003283	0.003442	0.003448	0.003630	0.003856	0.005487	0.007401	0.011192
	Point Query	0.000557	0.000663	0.000715	0.000881	0.001122	0.002896	0.005028	0.009483
Bit Hash	Delete	0.000962	0.001165	0.001359	0.001767	0.002753	0.010661	0.020123	0.039643
	Insert	0.003657	0.003903	0.004099	0.004499	0.005349	0.013639	0.022934	0.042463
	Point Query	0.000521	0.000813	0.000992	0.001394	0.002390	0.010282	0.019494	0.038578
kd-Tree	Delete	0.002989	0.003472	0.003969	0.004823	0.005603	0.006942	0.007316	0.009964
	Insert	0.002171	0.002204	0.002277	0.002390	0.002412	0.002957	0.003238	0.004987
	Point Query	0.002018	0.002033	0.002113	0.002286	0.002388	0.003076	0.003868	0.006119

Table E.1: Total Execution Time (in seconds) of Each Operation for Points from Uniform Distribution of Varying Dimensionality

		Dimensions							
Structure	**Operation**	1	2	3	5	10	50	100	200
Sequential Scan	Delete	0.472209	0.587695	0.698374	0.698280	0.628742	0.870596	1.470970	3.189490
	Insert	0.084389	0.095311	0.091186	0.096900	0.111276	0.135272	0.142041	0.138340
	Point Query	0.084033	0.090757	0.089855	0.095227	0.110525	0.134600	0.139539	0.131027
Octree	Delete	0.003181	0.002863	0.002899	0.004750	0.060280	-	-	-
	Insert	0.003490	0.003123	0.003417	0.006512	0.096882	-	-	-
	Point Query	0.002517	0.002092	0.002068	0.003469	0.038702	-	-	-
Pseudo-Pyramid Tree	Delete	0.001129	0.001311	0.001404	0.001567	0.001715	0.003100	0.005837	0.011133
	Insert	0.003103	0.003043	0.003121	0.002771	0.002762	0.004453	0.006723	0.010688
	Point Query	0.000625	0.000750	0.000814	0.000982	0.001199	0.002469	0.004794	0.009071
Pyramid Tree	Delete	0.001038	0.001174	0.001341	0.001462	0.001752	0.003932	0.006805	0.012752
	Insert	0.003096	0.003368	0.003466	0.003604	0.003888	0.005053	0.007492	0.012443
	Point Query	0.000555	0.000635	0.000752	0.000890	0.001249	0.003232	0.005814	0.010997
Bit Hash	Delete	0.000950	0.001153	0.001398	0.001754	0.002715	0.010399	0.020252	0.040310
	Insert	0.003520	0.003885	0.004107	0.004437	0.005318	0.013232	0.022708	0.042113
	Point Query	0.000510	0.000789	0.001000	0.001382	0.002382	0.010002	0.019695	0.039192
kd-Tree	Delete	0.002911	0.003400	0.003990	0.004584	0.005309	0.006993	0.007280	0.009264
	Insert	0.002158	0.002190	0.002325	0.002356	0.002362	0.002729	0.003344	0.005076
	Point Query	0.002072	0.002085	0.002198	0.002250	0.002320	0.002940	0.004033	0.006363

Table E.2: Total Execution Time (in seconds) of Each Operation for Points from Skewed Distribution of Varying Dimensionality

		Dimensions							
Structure	**Operation**	1	2	3	5	10	50	100	200
Sequential Scan	Delete	0.452688	0.587081	0.697896	0.698466	0.628546	0.891051	1.462300	3.001740
	Insert	0.083242	0.094424	0.085648	0.096576	0.111202	0.132845	0.128193	0.152179
	Point Query	0.082480	0.089000	0.084912	0.095301	0.110343	0.131760	0.127011	0.139105
Octree	Delete	0.002921	0.002943	0.003555	0.008215	0.196480	-	-	-
	Insert	0.003374	0.003168	0.004207	0.008885	0.200285	-	-	-
	Point Query	0.002290	0.002197	0.002620	0.006860	0.174983	-	-	-
Pseudo-Pyramid Tree	Delete	0.001069	0.001261	0.001352	0.004816	0.005001	0.006457	0.009281	0.013836
	Insert	0.003049	0.002917	0.002505	0.009361	0.009485	0.010845	0.013384	0.017897
	Point Query	0.000575	0.000690	0.000791	0.007716	0.007960	0.009414	0.011603	0.015418
Pyramid Tree	Delete	0.001016	0.001179	0.001375	0.001435	0.001683	0.003667	0.006479	0.011976
	Insert	0.003041	0.003345	0.003504	0.003575	0.003980	0.005496	0.007778	0.012194
	Point Query	0.000542	0.000637	0.000758	0.000871	0.001212	0.002997	0.005691	0.010567
Bit Hash	Delete	0.000927	0.001112	0.001354	0.001651	0.002513	0.009530	0.018395	0.036484
	Insert	0.003368	0.003819	0.004049	0.004349	0.005112	0.012236	0.020933	0.038674
	Point Query	0.000493	0.000753	0.000946	0.001285	0.002204	0.009126	0.017825	0.035381
kd-Tree	Delete	0.003257	0.003522	0.003926	0.005384	0.007460	0.009404	0.011730	0.014420
	Insert	0.002268	0.002168	0.002241	0.002390	0.002412	0.002506	0.003285	0.004643
	Point Query	0.002142	0.002025	0.002081	0.002296	0.002349	0.002713	0.003940	0.005929

Table E.3: Total Execution Time (in seconds) of Each Operation for Points from Clustered Distribution of Varying Dimensionality

		#Points			
Structure	**Operation**	10000	100000	500000	1000000
Sequential Scan	Delete	0.684460	69.043000	1767.620000	6380.810000
	Insert	0.124515	15.434500	603.073000	2233.540000
	Point Query	0.123733	14.959700	596.142000	2185.170000
Octree	Delete	-	-	-	-
	Insert	-	-	-	-
	Point Query	-	-	-	-
Pseudo-Pyramid Tree	Delete	0.001580	0.025124	0.164381	0.350934
	Insert	0.003628	0.060662	0.326802	0.675884
	Point Query	0.001059	0.016118	0.105968	0.245357
Pyramid Tree	Delete	0.002060	0.031297	0.211423	0.556193
	Insert	0.004100	0.054527	0.379671	0.712960
	Point Query	0.001554	0.025796	0.177153	0.446886
Bit Hash	Delete	0.003926	0.050295	0.277580	0.578340
	Insert	0.006568	0.081556	0.463257	0.962260
	Point Query	0.003515	0.043550	0.249102	0.516167
kd-Tree	Delete	0.005552	0.144179	1.264420	2.967910
	Insert	0.002401	0.062702	0.565952	1.367600
	Point Query	0.002408	0.061273	0.537383	1.330470

Table E.4: Total Execution Time (in seconds) of Each Operation for Varying Numbers of Points from Uniform Distribution

		Dataset		
Structure	**Operation**	**Astrophysics**	**Hurricane Isabel**	**Armadillo Mesh**
Sequential Scan	Delete	1315.81	1920.73	1336.2
	Insert	436.385	691.833	215.865
	Point Query	435.88	651.428	212.585
Octree	Delete	-	-	-
	Insert	-	-	-
	Point Query	-	-	-
Pseudo-Pyramid Tree	Delete	82.908	14.9052	0.160532
	Insert	81.231	14.4715	0.126938
	Point Query	70.4391	14.3358	0.0956872
Pyramid Tree	Delete	60.8105	41.5488	0.164531
	Insert	65.0126	44.2493	0.192165
	Point Query	60.0216	44.1224	0.134481
Bit Hash	Delete	0.191339	0.248544	0.102674
	Insert	0.360452	0.419106	0.11828
	Point Query	0.172516	0.222288	0.0811833
kd-tree	Delete	68.9612	578.798	0.91045
	Insert	0.663231	0.434781	0.384241
	Point Query	0.639265	0.408527	0.374425

Table E.5: Total Execution Time (in seconds) of Each Operation on Real Datasets

Appendix F

Algorithms and Code Listings

This section contains high-level pseudo-code and low-level code listings of algorithms and techniques whose details were not present in the main report.

F.1 Bit Hash Algorithm

Bit Hash's hashing function will be described in this section. The function hashes each individual coordinate (floating point value) and combines them using exclusive-or ($\oplus$) and bitshifting operations. A magic number representing the reciprocal of the golden ratio, $\phi = \frac{1+\sqrt{5}}{2}$, is used when combining the hash values of individual coordinates. The choice to use the golden ratio was inspired by Jenkins' hash function [102], where it is used to ensure consecutive floating point values will be mapped to integers with large distances between them. This increases the likelihood of having points distributed more uniformly across buckets when points are clustered within a small numerical range.

Algorithm 1: Hashing Multi-Dimensional Point in Bucket Hash Table

Algorithm `hashPoint` (d, p)
- **input** : d = number of dimensions
- **input** : $p_0, p_1, \ldots, p_{d-1}$ = coordinates of point
- **output**: *seed* = integer representing hashed point
- **begin**
  - $seed = 0$;
  - **for** $i = 0$ *to* $d-1$ **do**
    - $seed = seed \oplus (\texttt{hashFloat}(p_i) + \texttt{0x9e3779b9} + (seed << 6) + (seed >> 2))$
  - **end**
  - **return** *seed*
- **end**

`hashFloat` is a function which hashes an individual 32-bit floating point number and corresponds to the function `float_hash_impl2` in Listing F.1. This function iterates over each digit of the number system's radix and accumulates a hash value.

F.2 Code Listings

```
1  // Copyright 2005-2009 Daniel James.
2  // Distributed under the Boost Software License, Version 1.0. (See accompanying
3  // file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
4
5  // A general purpose hash function for non-zero floating point values.
6
7  inline void hash_float_combine(std::size_t& seed, std::size_t value)
8  {
9      seed ^= value + (seed<<6) + (seed>>2);
10 }
11
12 template <class T>
13 inline std::size_t float_hash_impl2(T v)
14 {
15     boost::hash_detail::call_frexp<T> frexp;
16     boost::hash_detail::call_ldexp<T> ldexp;
17
18     int exp = 0;
19
20     v = frexp(v, &exp);
21
22     // A postive value is easier to hash, so combine the
23     // sign with the exponent and use the absolute value.
24     if(v < 0) {
25         v = -v;
26         exp += limits<T>::max_exponent -
27             limits<T>::min_exponent;
28     }
29
30     // The result of frexp is always between 0.5 and 1, so its
31     // top bit will always be 1. Subtract by 0.5 to remove that.
32     v -= T(0.5);
33     v = ldexp(v, limits<std::size_t>::digits + 1);
34     std::size_t seed = static_cast<std::size_t>(v);
35     v -= seed;
36
37     // ceiling(digits(T) * log2(radix(T))/ digits(size_t)) - 1;
38     std::size_t const length
39         = (limits<T>::digits *
40                 boost::static_log2<limits<T>::radix>::value - 1)
```

```
41          / limits<std::size_t>::digits;
42
43      for(std::size_t i = 0; i != length; ++i)
44      {
45          v = ldexp(v, limits<std::size_t>::digits);
46          std::size_t part = static_cast<std::size_t>(v);
47          v -= part;
48          hash_float_combine(seed, part);
49      }
50
51      hash_float_combine(seed, exp);
52
53      return seed;
54  }
```

Listing F.1: Code to Hash Single 32-bit Floating Point Value (Source code from file `boost/functional/hash/detail/hash_float_generic.hpp` in Boost Library 1.42.0)

Appendix G

Additional Index Structures

This section describes index structures which were implemented during the project, but were not discussed or evaluated in the main report due to time constraints. Preliminary findings on the structures' performance is discussed.

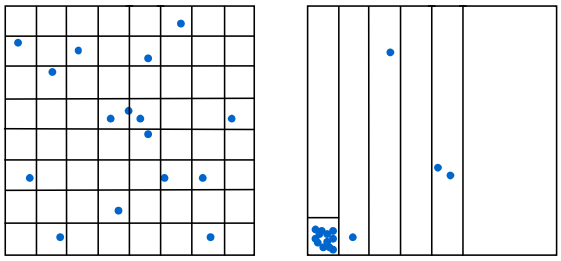
G.1 Multigrid Tree

The Multigrid Tree decomposes the data space into a uniform grid by cutting each dimension into B intervals. A cell is defined by the intervals of each dimension it is contained in. Formally, a cell is defined as a $d-1$ tuple $C = \{c_0, c_1, \ldots, c_{d-1}\}$, where c_i is the interval the cell belongs to in dimension i. There are a total of B^d cells. Figure G.1a illustrates how the Multigrid Tree decomposes 2D data space when $B = 8$, resulting in $8^2 = 64$ cells. Each cell corresponds to a bucket in the Multigrid Tree.

This structure is a hybrid of hash-based and tree-based structures. It is a tree where each leaf node is a bucket containing one or more points and non-leaves are hash tables. Each non-leaf hashes points to the interval the point is contained in, for a single dimension i. The bucket a point is hashed to is another node of the Multigrid Tree. This node is either another hash table, which hashes points based on which interval of dimension $i+1$ they are inside, or a bucket (i.e. an array of points).

The root node always hashes points using the interval of dimension 0 they are contained in. When these buckets exceed a certain capacity, say M, then the bucket is turned into a hash table which hash points to their interval in dimension 1. This is repeated until the last dimension, $d-1$, is reached. At this point, there are no more dimensions to discriminate points with, so the buckets are not transformed into hash tables when their size exceeds M.

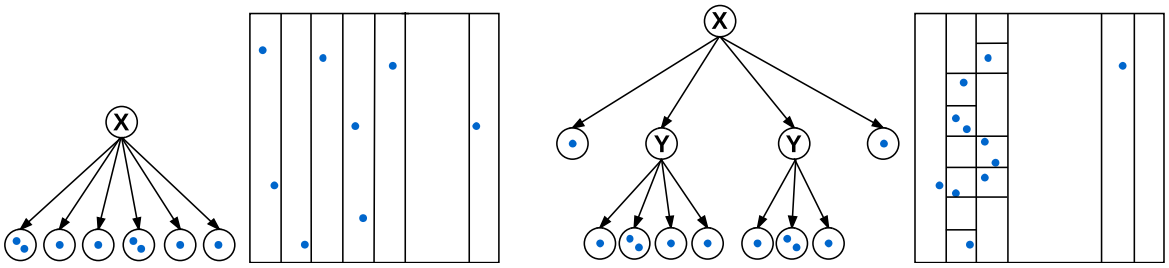
In effect, this creates a tree where the maximum height is $d+1$ and each non-leaf node has at most B children that correspond to the B intervals of the dimension it is discriminating against. All non-leaf nodes at level i of the tree hash points to the interval of dimension i they belong to. The hashing function for dimension i is given in Equation G.1.



(a) Full Multigrid Tree Decomposition

(b) Multigrid Tree Decomposition of Clustered Data

Figure G.1: Multigrid Tree Decompositions



(a) Multigrid Tree with Minimum Height

(b) Multigrid Tree with Maximum Height

Figure G.2: Multigrid Trees and their Respective Spatial Decompositions in 2D

$$h_i(p) = \left\lfloor \frac{p_i - min_i}{max_i - min_i} \times B \right\rfloor \tag{G.1}$$

where p is a point and min_i, max_i are the lower and upper boundaries of dimension i.

Figure G.2a shows a Multigrid Tree with a single non-leaf node and its accompanying spatial decomposition in 2D. Notice how points are discriminated only using the value of the x dimension. Figure G.2b shows a Multigrid Tree where some of the bucket sizes have exceeded M, which is 3 in this case. Notice how some intervals in the x dimension have been sliced in the y dimension now, allowing for greater discrimination.

Equation G.1 can clearly be computed in constant time. Since there are at most $d+1$ levels in the tree, at most d instances of Equation G.1 will be computed. It follows that retrieving the bucket a point is contained in takes $O(d)$ time. If points are uniformly distributed across the grid, then this structure will take $O(d)$ time for all operations. However, it is possible for all the points in a dataset to be mapped to the same cell, making `insert`, `delete` and point queries take $O(n)$ time.

For highly skewed datasets, most of the points will end up being mapped to the same few cells. This causes the structure to degenerate to Sequential Scan in a similar manner to the Pyramid Tree (see Section 8.4). Figure G.1b illustrates this limitation. Despite both dimensions being used to discriminate between points, they are all get hashed to the same value because they are contained in the same cell.

G.2 iMinMax(θ)

iMinMax(θ) is an index structure similar to the Pyramid Tree, whose behaviour is affected by the parameter θ [103]. θ can be tuned to optimise the structure for different data distributions. Like the Pyramid Tree, it reduces points to a single dimension and applies a one-dimensional index structure to these values. To dynamically adapt the spatial decomposition to different point distributions, the value of θ is tweaked. This changes the hashing function, meaning the structure must be rebuilt every time θ is changed.

An implementation of this was developed using the base hash structure described in Section 5.4.2. Preliminary tests revealed that the structure was only slightly faster than the Pyramid Tree for the astrophysics and hurricane Isabel datasets, regardless of which value was chosen for θ. The majority of the structure's execution time was spent rebuilding the structure, supporting the claim that existing dimension reduction techniques will require rebuilding are not suitable for dynamic data (discussed in Section 8.4).

G.3 Bucket *kd*-Tree

Instead of storing a single point in every node of the tree, the Bucket *kd*-Tree stores multiple points in each node (termed a bucket), with the constraint that only leaf nodes store points. Non-leaf nodes serve to partition the data space, each cutting the space along a single dimension by some value (the cutting value). When the number of points in a leaf exceeds a certain number, say M, then a cutting dimension and value are determined, which are used to split a leaf into two. The left and right children nodes are leaves that contain the points from the split node that lie on each side of the split.

To query or delete a point, the leaf containing the point is found, which is sequentially scanned to find the point. If the point is found, it is removed from the leaf. If the total number of points being stored in the modified leaf node and its sibling is less than $\frac{M}{4}$, then the two leaf nodes are *merged* into their parent. The parent becomes a leaf node containing all the points stored in the two leaves.

Compared to the point *kd*-tree, the bucket *kd*-tree has more knowledge about the data's distribution since multiple points are stored in a bucket. With the point *kd*-tree, only one point at a time is used to determine the cutting dimension, so less knowledge about the distribution is used to perform a split. Knowing more about the distribution allows the bucket *kd*-tree to partition the data more effectively to construct a more balanced tree.

This structure was implemented as part of the project. The *sliding midpoint* splitting rule was used to split full leaves, which is described in [104]. Preliminary results show that it reduced balance factor, but an inefficient implementation meant there were large constant factors that made the structure slower than the point *kd*-tree implementation. More work optimising the implementation should be performed to assess its suitability for dynamic scientific datasets.

Appendix H

mdsearch Documentation

---

mdsearch is a small C++ library that implements a subset of the index structures discussed in this report. The library is available to download on an online Git repository [1]. This library makes heavy use of template metaprogramming and inline functions to boost performance. As such, it is a header-only library. Each implemented index structure can be used by simply including its corresponding header file.

Currently, the following index structures have been implemented:

- **Pyramid Tree** – see Section 4.2 for more information
- ***kd*-Tree** – see Section 4.5 for more information
- **Bit Hash** – see Section 4.4 for more information
- **Multigrid Tree** – see Section G.1 for more information

H.1 Dependencies

mdsearch requires the following header-only Boost libraries

- Boost.Unordered
- Boost.Functional/Hash
- Boost.Lexical_Cast

[1] `https://github.com/DonaldWhyte/mdsearch`

H.2 API

Each index structure is represented as a class, which implements the three operations evaluated in the project. Therefore, each index structure class has the following methods:

- `bool insert(point)` – inserts unique point into the structure. Returns true if point insertion was successful and false if point is already being stored and has not been inserted.
- `bool remove(point)` – deletes point from structure. Returns true if the point was found and deleted successful and false if the point was not found.
- `bool query(point)` – returns true if point is being stored in structure and false otherwise.

Table H.1 describes the role of each class in the library. Figure H.1 illustrates the inheritance and dependencies between these classes using a UML class diagram.

Data Type	**Description**
`Real`	Type of individual coordinate of point
`RealList`	Re-sizeable array of `Reals`
`HashType`	Type of discrete hash value used by hash-based structures
`Point<D>`	Point with D dimensions
`Boundary<D>`	Boundary in D-dimensional space
`Dataset<D>`	Dataset containing D-dimension points, which can load points from files and compute the minimum bounding region containing every point
`KDTree<D>`	*kd*-tree for D-dimensional space
`Multigrid<D>`	Multigrid Tree for D-dimensional space
`HashStructure<D>`	Abstract class for hash-based structures, which follows the Bucket Pseudo-Pyramid Tree structure described in Section 5.4 closely
`PyramidTree<D>`	Pyramid Tree for D-dimensional space
`BitHash<D>`	Bit Hash for D-dimensional space

Table H.1: Descriptions of Each Type in mdsearch Library

H.3 Examples

The library comes with a program that generates random points and performs a set of correctness tests on each index structure. This program is contained within the `test_structures.cpp` file and contains examples of how to use the index structures.

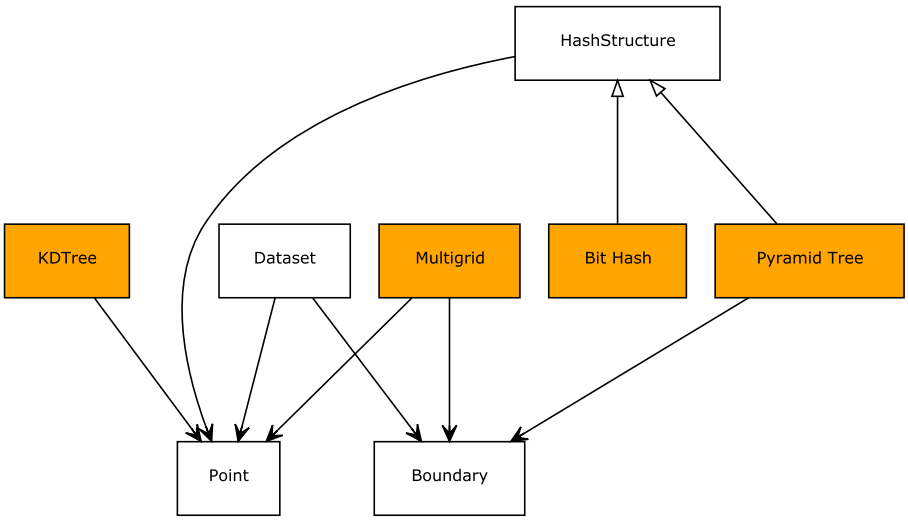


Figure H.1: UML Class Diagram of mdsearch Library. Coloured Boxes Represent Index Structures.