Cluster-Computing and Parallelisation for the Multi-Dimensional PH-Index

Master Thesis

Bogdan Aurel Vancea
<bvancea@student.ethz.ch>

Prof. Dr. Moira C. Norrie
Tilmann Zaeschke
Christoph Zimmerli

Global Information Systems Group
Institute of Information Systems
Department of Computer Science
ETH Zurich

15th March 2015

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

globis

Abstract

Here comes the abstract.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Objectives 2

1.3 Thesis outline 3

2 Background **5**

2.1 The PhTree 5

2.2 Related work 7

2.2.1 Distributed Indexes 7

2.2.2 Concurrent data structures 7

3 Index distribution **9**

3.1 Challenges 9

3.2 Distribution strategies 10

3.2.1 Hashing 11

3.2.2 Spatial splitting 12

3.2.3 Z-Order curve splitting 13

3.3 Architecture 15

3.4 Algorithms 16

3.4.1 Key-mapping design 16

3.4.2 Point queries 17

3.4.3 Range queries 17

3.4.4 Nearest neighbour queries 18

3.4.5 Entry load balancing 20

4 Concurrency **23**

4.1 Challenges 23

4.2 Concurrency strategies 24

4.2.1 Copy-on-Write 24
4.2.2 Locking 25

5 Implementation **29**
5.1 System description 29
5.2 Technologies 30

6 Evaluation **31**

7 Conclusions **33**
7.1 Contributions 33
7.2 Future work 33

1 Introduction

This chapter briefly describes the context and the motivation of the thesis and presents the objectives that are to be achieved. Finally, section 1.3 provides an overview of this thesis.

1.1 Motivation

Multi-dimensional data is widely used today, especially in domains like database management systems, geographic information systems, computer vision and computational geometry. When all of the dimensions of the data hold numerical values, this data can be viewed as a collection of points in higher dimensional spaces. Due to this nature, multi-dimensional numerical data provides the possibly of posing more complex queries based on the distance between these points in space. For example, in the context of a geo-information system, one could query for all of the points that fall inside a specific hyper-rectangle or attempt to find the nearest neighbours of an arbitrary query point.

Several point-based multi-dimensional indexing solution have been developed in the latest years, the most prominent being kD-trees [1] and quadtrees [4]. This type of data structures store the multi-dimensional data such that more complex operations, like range and nearest neighbour queries are executed efficiently. The PhTree [10] is a new multi-dimensional data structure based on the quadtree. In addition to providing support for complex queries, the PhTree is also space-efficient, as its space requirements are sometimes even lower than those of multi-dimensional arrays.

As technology advances and the world becomes more connected, multi-dimensional data becomes easier to acquire and store. Because of this, it is necessary that multi-dimensional data structure need to store and manage more data than would fit a single machine. However, traditional multi-dimensional indexes like the kD-tree and quad-tree do not cover this use case as they are designed to run on a single machine.

Additionally, in the last few years the processor speed has reached the power wall and processor designers cannot increase the CPU frequency by increasing the number of transistors. Recent advances in processor design have been made by adding more cores on CPU's rather than increasing the processing frequency. Therefore, it is important that contemporary data structures be adapted to multi-core architectures by allowing them to support concurrent accesses. As with the case of the increase storage requirements, traditional multi-dimensional data structures do not support concurrent write operations.

This thesis attempts to provide a solution to these two issues by extending the PhTree multi-dimensional index to run on distributed cluster of machines. Furthermore, the PhTree is also updated to support concurrent access.

1.2 Objectives

The previous section has highlighted two challenges currently faced by indexing systems: *high storage requirements* and *support for concurrent access.*. This work proposes the distributed PhTree, a version of the PhTree that can be run on a cluster of machines, making it able to handle data sets that cannot fit in the main memory of a single machine. Moreover, the distributed PhTree should be able to handle concurrent requests. This applies both to requests sent to different machines that are part of the cluster and concurrent requests sent by different clients to the same machine.

Specifically, the distributed PhTree has to fulfill the following requirements:

Cluster capability The system should run across a network of machines, making use of the memory and processing resources on each machine. Furthermore, the system should attempt to balance the number of multi-dimensional entries that each machine is responsible of.

Cluster concurrency The system should be able to support concurrent requests to different nodes of the cluster. Each node should be able to process queries related to the entries that it stores locally.

Node concurrency Each node should support multi-threaded read and write access to the entries that it stores.

As the thesis touches on two main subjects, distribution and concurrency, the main challenges encountered are twofold. From the distribution perspective, the challenges are the identification of suitable entry distribution and balancing strategies, devising efficient algorithms for executing queries across multiple cluster nodes, and the efficient management of a very large number of cluster nodes. For the concurrency perspective, the challenges are the identification of a suitable concurrent access strategy that can maximize the number of concurrent write operations.

1.3 Thesis outline

This chapter gave an overview of the challenges currently faced by multi-dimensional indexing structures and briefly explained how this work seeks to address them. Additionally, this chapter also presented the main objectives of this thesis. The rest of the thesis is structured as follows:

Chapter 2 provides additional information about the PhTree, its characteristics and supported operations. The second part of this chapter describes relevant previous work done in the areas of distributed multi-dimensional indexes and concurrent data structures.

The design of the distributed PhTree from the point of view of a distributed system is presented in **Chapter 3**. This chapter presents the chosen data distribution strategy, and also touches on the possible alternatives and the consequences of this choice. Additionally, this chapter provides an overview of how the queries spanning multiple nodes are executed by the system.

The addition of the multi-threaded read and write support for the PhTree is discussed in **Chapter 4**. Several concurrent access strategies are discussed, together with their advantages, disadvantages and consistency guarantees.

Chapter 5 describes the implementation-specific decisions that were taken during the development process. This chapter also presents the technologies that were used and justifies the technological choices.

The distributed PhTree is evaluated in **Chapter 6**. The performance characteristics of the implemented distributed systems as well as those of the implemented concurrency strategy are discussed.

Chapter 7 concludes the thesis by presenting the contribution of this work in the context of distributed multi-dimensional indexing systems. This second part of this chapter discusses possible future contributions.

2
Background

The first part of this chapter analyses the single-threaded PhTree, a multi-dimensional data structure and the starting point of this work. It provides an overview of its specific characteristics and describes the supported operations.

The second part of this chapter presents the relevant related work in the area of distributed multi-dimensional index and concurrent data structures.

2.1 The PhTree

The PhTree [10] is a novel multi-dimensional indexing structure that focuses on fast retrieval and highly efficient storage. It combines concepts from PATRICIA-tries, quadtrees and hypercubes to encode the multi-dimensional data using bit prefix sharing. Because of this, the storage requirements can sometimes be lower than those of the storage requirements of an array of objects.

An example PhTree storing 2-dimensional 4 bit points is shown in 2.1. The tree illustrated in this figure stores the points: (0, 4), (3, 4) and (3, 6). However, each point is stored as the *bit interleaving* of its values in all dimensions. Therefore, the bit interleaving of point (3, 6), represented in binary as (0011, 1010) is 01001110. By storing the points under the *bit interleaving* form, the PhTree maps the multi-dimensional points into a 1-dimensional space. The 1-dimensional values will be referred to as *Z-values* as they form the *Z-order space filling curve*.

The PhTree has the following important properties:

- **Z-Ordering**. The multi-dimensional entries are order according to the Z-order space filling curve. As a consequence, points which are close in the original multi-dimensional space will also be relatively close in the Z-Ordering.

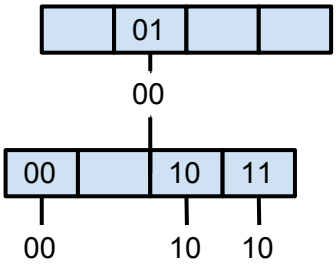


Figure 2.1: 2 dimensional 4bit PhTree containing the points: (0000, 1000), (0011, 1000), (0011, 1010).

- **Constant height**. The height of the PH-tree is equal the number of bits needed to store each value of the multi-dimensional space. Therefore, when the values in each dimension are represented in 64 bits, as long or doubles, the maximum height of the PhTree is 64. This property is important as *the maximum height of the PhTree is independent of the number of points it stores*.
- **Balancing is not needed**. The PhTree does not perform balancing operation after write operation have occurred. While this fact could lead to un-balanced trees, it is usually not a problem in practice, as the height of tree is limited by a constant factor.

The PhTree is similar to a HashMap, as it stores entries consisting of key-value pairs. In this case however, the key is a multi-dimensional data point, while the value can be an arbitrary object. The following operations are supported:

- **Point queries** operate on single multi-dimensional points. The possible queries are *get()*, *put()*, *delete()* and *updateKey()*. The *get()* query retrieves the value associated with a multi-dimensional key, and the *delete()* operation removes the value associated with a multi-dimensional key. The *put()* operation adds new key-value to the index. Finally, the *updateKey()* operations updates the key associated with an existing key-value entry.
- **Range queries** select all of the data points that fall inside a multi-dimensional hyper-rectangle. This rectangle is defined by a *lower left* and a *upper right* point.
- **Nearest neighbour queries** select the closest k points in the index to an arbitrary query point.

Currently, the amount of entries the PhTree can store limited by the memory available on the host system, as the PhTree stores all data completely in-memory. A distributed version of the PhTree would be able to store a much larger amount of entries and would be able to make use of existing clusters of computing nodes. Specifically, the distributed version would be well suited for running in cloud environments, where users can easily add more computing nodes to their existing systems to handle higher storage requirements and improve performance. The challenges in creating a distributed version of the PhTree are the implementation of range and nearest neighbour queries, which might need to be dispatched to multiple nodes of the system. The goal is to design a set of algorithms that minimize the number of nodes to which a complex query is sent.

2.2 Related work

The following sections reviews the existing research concerning distributed indexes and concurrent data structures. The goal of this section is to understand to existing approaches that can be used for the design and implementation of the distributed PhTre and to identify which of these approaches are best suited for the use cases targeted by this work.

2.2.1 Distributed Indexes

Distributed Hash Tables (DHT's) are a class of decentralized distributed systems which expose an API similar to that of a hash-table. These P2P systems use a structured overlay network to assign a given key to a node in the system. Requests for a certain key can be sent to any node of the system and will be forwarded to the node responsible for it. DHT's have been designed to work on internet-scale, connecting large numbers of machines across large distances. The main concerns are scalability and availability, while the *get()* and *put()* operations have "best effort" semantics. Chord [9], one of the most popular and most efficient DHT system, manages to guarantee that any request for a certain key will arrive the responsible node in at most *O(logn)* hops, in the context in which each nod contains only *O(logn)* links to other nodes.

One important issues of distributed hash tables is that they only support "exact match" queries. The Prefix Hash Tree [7] adds support for range queries to a DHT by using a secondary data structure, to maintain a mapping between nodes and keys. This mapping is a trie containing the prefixes of the bit-strings of the keys, and thus, logically, leaf nodes of this trie correspond to interval regions of keys. All leaf-nods are linked using a linked-list, allowing range queries to traverse the query range starting for the first leaf that matches the range.

A similar approach is taken by the SkipIndex [11], a multi-dimensional indexing system that uses a SkipGraph to map geometrical regions to machine nodes. Each system node maintains only a partial view of the region tree and thus range and nearest neighbour queries are performed using selective multicasting.

A different approach for creating a scalable multi-dimensional index is to use a big-data platform. [6] propose storing a block-based multi-dimensional index structure like the R-tree directly in HDFS and use query models like MapReduce as building blocks for the implementation of range and nearest neighbour queries.

2.2.2 Concurrent data structures

Concurrent data structures are a highly studied problem in current times. Currently, there are two main ways of implementing a concurrent data structures: lock-based solution and lock-free solutions. Lock-based solutions use locks to synchronize the concurrent access to the either the data structure itself, an approach called *course-grained locking*, or to parts of the data structure, in the approach called *fine grained locking*. Lock-free solutions use either atomic primitives, like *compare-and-swap* or software transactional memory to synchronize modification attempts on the data structure.

In the area of concurrent search trees, lock-based approaches are much more common than

lock-free solutions. Fraser [5] provides a set of API's that can be used to design arbitrary concurrent data structures. While these API's can be used to for any type of data structures, they are based on software transactional memory or multi-word CAS operations, currently not implemented by contemporary hardware. Brown [3] presents a general technique for non-blocking trees implemented using multi-word equivalents of the *load-link*(LL), *store-conditional* (SC) and *validate*(VL) atomic primitives.

A naive concurrent implementation of a search tree can be achieved by protecting all access using a single lock. Concurrency can be improved by synchronizing the access at the level of nodes. *Hand-over-hand locking*, something also called *lock-coupling*, is a fine-grained technique that states that a thread can only acquire the lock of a node if it holds the lock of the node's parent node. This technique allows multiple concurrent write operations, but the degree of concurrency is limited due to the fact that all subtrees of a locked node are not accessible, even if the other threads need to perform modifications in a different part of the tree. Another approach is to perform optimistic retries by only locking the nodes that should be modified, check if locked nodes have not been removed from the tree and then perform the update. This idea is used by [2] to implement a highly concurrent AVL tree.

However, an issue with all concurrent search trees where modifications are done to a single node is that the execution of queries which have to run on the tree for longer periods of time, like iterators or range queries, overlaps with the execution of other write-operatons. Therefore, longer running queries will view the any updates that are done on the tree. In some cases, it is desirable to run the read queries on a snapshot of the tree that is isolated from other concurrent updates. This can be achieved using a strategy called *Copy-On-Write*, which allows a single writer and an arbitrary number of reads to access the same data structure concurrently. Moreover, each write operations creates a new version of the data structure that is only available to readers which have started after the update which created the new version has finished.

3
Index distribution

This chapter describes the design of the distributed PhTree from the distributed point of view. It presents the challenges and the possible design choices for extending the PhTree to run in a distributed setting, discusses the system architecture and describes the algorithms used for the distributed querries. Concurrency related issues are not tackled in this chapter as they are addressed in Chapter 4.

3.1 Challenges

While extending the PhTree to be a distributed index increase its storage capacity, there are a number of challenges to overcome in order to reach a good implementation. These challenges are:

- **Balancing the storage load**. In a cluster of n indentical nodes, nodes which have the same hardware resources available, all nodes should store and equal amount of data. Assuming that the values associated with each key are relatively equal in size, all nodes should store a relatively equal number of key-value entries.
- **Maintaining data locality**. The PhTree needs to provide support to range and nearest neighbour queries, types of queries which generally analyze contiguous portions of the space and return sets of points which are generally close together. If the indexed points manifest locality, points close in space reside on the same node, range and nearest neighbour queries could be sent to only a part of the nodes in the cluster. In the best case, range and nearest neighbour queries could be sent to only a single node. It is therefore preferable that queries are sent to as few nodes as possible, minimizing both the response time and the request load to the nodes in the cluster.
- **Ensuring cluster scalability**. To support very large storage requirements, the system

should support a large number of online-indexing nodes. Furthermore, to support scalability, adding and removing nodes to and from the system should be easy and should not require the cluster to be shut down and then re-started. Such atomic cluster reconfiguration is challenges because of the data migrations that it entails to achieve a balanced storage load. New nodes added to the system start off as being empty and should received some entries from the currently running active nodes. Moreover, the data received should preserve locality. Nodes that are being removed from the system need to first make that the entries currently stored and moved to different nodes in the system in a manner that preserves locality.

- **Minimizing node information**. As the PhTree is held in-memory, it is important for the indexing nodes to be efficient in managing their memory, to maximize the entries stored by the tree. Maintaining a connection to node over a network generally requires an open socket on both participating nodes. However, active sockets take up system memory and CPU cycles, making it important to reduce the number of open connections that a node has to other nodes in the system. Therefore, it is important to devise a communication model in which each node has to be aware and communicate with a small number of other nodes in the system, to reduce the resource compution related to cluster membership.

Other challenges posed by distributed indexes and distributed systems in general are availabilty and security, however these issues are not tackled by this work.

By taking into account the discussed challenges, an ideal architecture of the system has the following requirements:

1. Ensure low response time by minimizing the number of network requests that have to be made.
2. Points are assigned to nodes in the cluster in a way that preserves locality.
3. New nodes can be added and removed without shutting down or stopping the cluster.
4. Each node maintains open connections to a small number of other nodes in the system.

The following sections of this chapter describe the distribution strategy chosen to properly address the challenges presented in this section, the final architecture of the system and the algorithms for the query execution.

3.2 Distribution strategies

There are two main solutions to the problem of distributing a tree data structure over a cluster of computing nodes:

- **Assigning tree nodes to machines**. In this approach, there will be only a single PhTree containing all of the entries stored by the cluster, each node of the PhTree being stored on an individual machine. This means that following a pointer from a parent tree

node to a child tree node could require the forwarding of the request to a different machine, the one containing the child tree node. As the PhTree depth is limited by a bit width of the data being stored, from the theoretical point of view, the number of such ”forwards” is limited by a constant number. Additionally, spreading the tree nodes uniformly across the machines will lead to a balanced storage load. The drawback of this approach is that even though the number of ”forwards” during a tree travels is limited by a constant number, the impact on the response time of the system is very high as each ”forward” constitutes of a network request. For example, for indexes storing 64 bit multi-dimensional points, a point query would take 64 forwards between the machines in the cluster in the worst case.

- **Assigning data points to machines**. In this approach, each machine in the system will maintain a in-memory PhTree and data points are assigned to machines according to some mapping. This mapping will be referred to as the *key-mapping*, as it mantains a mapping between the keys of the entries stored and the machines that store them. Queries for arbitrary points will always be dispatched to the machines that hold the poins, according the *key-mapping*. Therefore, if the *key-mapping* is known by the clients of the index, point queries can be resolved by a single ”network” request. However, in such an architecture, the manner in which the points are distributed according to the *key-mapping* influences the manner in which the range and nearest neighbour queries are executed.

While both of these alternatives satisfy some of the requirements of an ideal architecture, the drawbacks of the first approach, the assignment of tree nodes to machines, outweigh the advantages it provides. The number of network requests needed by this approach when storing 32 or 64 bit data, typical for most applications, make the use of this approach prohibitive in a setting where queries must have a low-latency.

Therefore, this work will focus on the second distribution approach. The following sections will present possible strategies for distributing the points to PhTree’s store on different machines in the cluster.

3.2.1 Hashing

The first approach considered is using a *hashing* function to assign the multi-dimensional points to machines in the clusters. This process is illustrated in figure 3.1. The hashing function takes the multi-dimensional point and generates a host id h based on the point values in each dimension. This approach is similar to the approach used to implement in-memory hash tables, where keys are mapped to buckets in the hash table.

The main advantage of this approach is that if the hash function is known to all of the machines in the cluster and to all of the clients, point queries can be executed in O(1) network requests. Moreover, if the hash function is uniform, the multi-dimensional points are spread uniformly across the cluster, leading to a balanced storage load on all of the machines.

However, the use of a hash function for the point distribution presents some drawbacks. First of all, a hash function works well for a fixed number of hosts n. A different cluster configuration, with a different number of machines, requires the use of a different hash function.

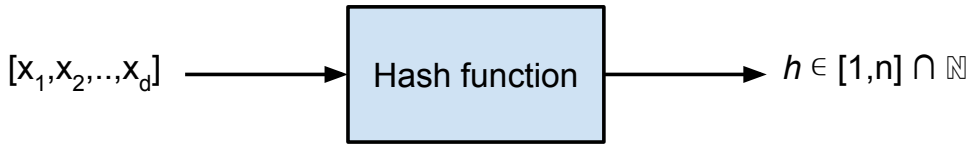


Figure 3.1: Distributing the points based on a hash function

Moreover, changing the hash function requires a re-hashing of all of the points already stored in the cluster, potentially leading to a situation where all of the stored points need to be moved to a different machine. This situation would case a lot of traffic within the cluster, making it hard for the system to answer queries until the migration process is finished.

A second issue with use of hash functions is the observation that hash functions which are perceived to be good in this situation spread the multi-dimensional points uniformly across the machines in the cluster. In such a situation, is is quite likely that the hash function does not preserve point locality, and such, complex queries like range queries and nearest neighbour queries will need to be dispatched to all of the computing nodes in the cluster.

3.2.2 Spatial splitting

Another approach that can be used for the point distribution problem is to split the partition the original space into a set of contiguous regions and assign each region to a hosts in the system. The most intuitive way to perform this split is to split the multi-dimensional space equally to obtain one region per index host. Figure 3.2 shows a potential splitting of a 2D space for 2 hosts, while Figure 3.3 presents a possible split of the same 2D space for 4 hosts.

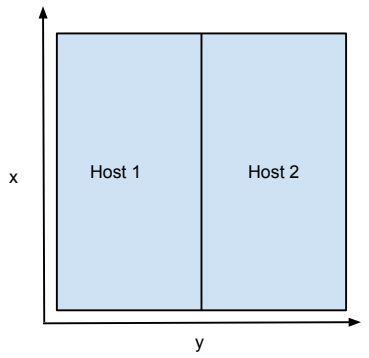


Figure 3.2: Equal geometric space splitting for 2 hosts and 2 dimensions

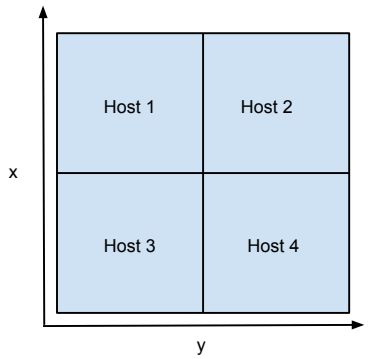


Figure 3.3: Equal geometric space splitting for 4 hosts and 2 dimensions

The spatial splitting approach is not only intuitive, but it also preserve locality, as generally, points which are close together are part of the same regions and are therefore stored on the same machine. Additionally, range queries can be resolved by dispatching the range query to all of the hosts whose assigned regions intersect with the range received as input. In the best cases, range queries can be dispatched to a single host. It is generally preferable to split

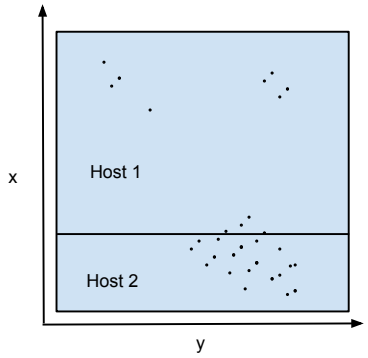


Figure 3.4

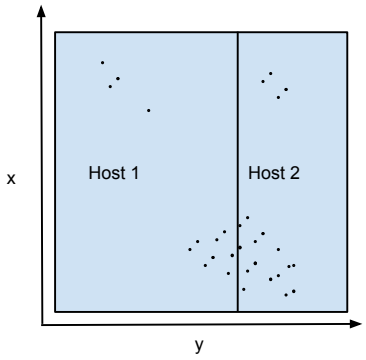


Figure 3.5: Equal geometric space splitting for 4 hosts and 2 dimensions

the space into hyper-rectangles, as these regions can be stored using only two data points, the bottom left and top right corners, independently of the number of dimensions of the original space.

While the spatial splitting works well in some situations, sometimes it is unclear how to properly split the space between the machines. For example, the 2D space is easy to split into 2 or 4 equal regions, but it is more difficult to split for 3 or 5 hosts. It is always possible to split the space into n equal sized ”strips”, however this approach diminishes the locality of the data for large values of n. Representing regions as arbitrary polygons instead of hyper-rectangles addresses the problem to a degree, but storing these polygons and computing intersection between them is more difficult than performing the same task using hyper-rectangles.

Difficulties in splitting the space can also arise even when using hyper-rectangle regions of different size. One instance is the case skewed point distributions, presented in figures 3.4 and 3.5. In the case of the point distributions illustrated in these figures, an equal space split would not split the points equally between the hosts. However, there are many possible rectangle-based splittings of the space, and it is unclear which of these it is better. In some case, the best split could be achieved by splittng the space into more regions than machines and assign each region to a machine.

Additionally, in the case of skewed input distributions, the optimal split of the space into regions will change over time, as some regions will end up containing more and more points. In this case, the cluster needs to go through a *re-balancing* phase, when the regions are modified and some points are moved from highly populated regions to less populated neighbouring regions. An important observation is that the number of neighbouring regions for an arbitrary regions is a function of the number of dimensions of the original space. This leads to a large number of regions participating, or at least being considered, in the re-balancing of the points from a single dense region to its neighbours.

3.2.3 Z-Order curve splitting

As previously mentioned, the in-memory PhTree stores the multi-dimensional points according to the Z-ordering, by mapping the initial space to the 1-dimensional Z-order curve. This

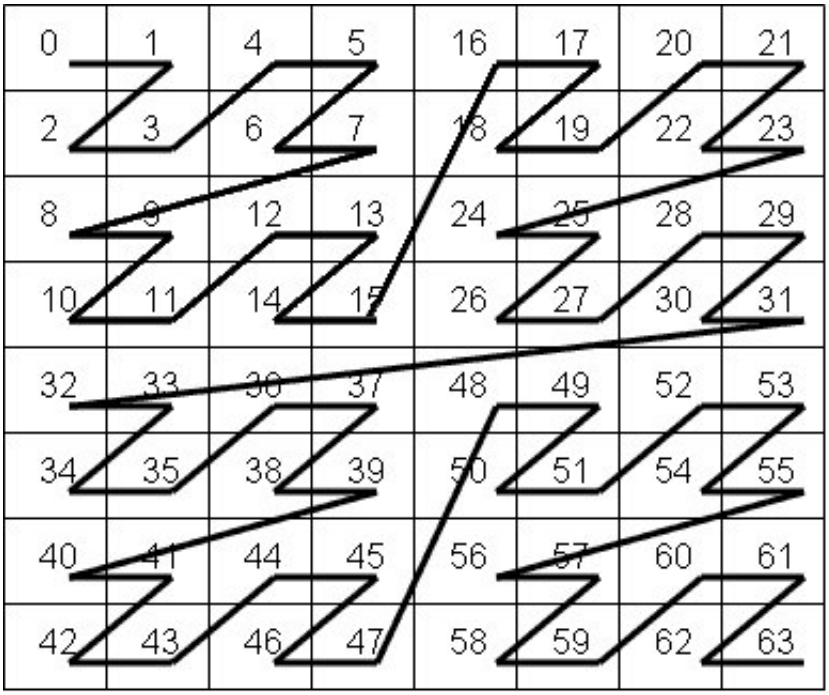


Figure 3.6: Z-order space filling curve filling the 2 dimensional space.

curve fills in the initial space in a contiguous and complete manner. Figure 3.6[1] shows the Z-order space filling curve filling the 2D space where the values in each dimension are limited by 3 bits.

Therefore, instead of splitting the original space into regions, one could attempt to parition the z-order curve into contigous intervals. Figure 3.7 shows the 1 dimensional z-order curve corresponding same 3 bit 2-dimensional space, split into 4 intervals of different colours: $[0, 12]$ with blue, $[13, 35]$ with yellow, $[36, 47]$ with violet and $[48, 63]$ with green. Figure 3.8 shows how the interval splitting is reflected in the original space. Each interval on the z-order curve corresponds to either a hyper-rectangle or a set of hyper-rectangles. Each interval in the z-order curve is reprsented by at most $O(d * w)$ hyper-rectangles, where d is the number of dimensions and k is the number of bits needed to represented the values in each dimension. [8] provides an algorithm for generating the hyper-rectangles associated with a z-order curve range.

An advantage of this splitting method is the regions corresponding to the z-order intervals manifest locality, making it well suited for a point distribution method. Moreover, even if in the original space each region has O(d) neighbours, each interval on the z-order curve has a small, constant number of neighbouring intervals (2 for the interior intervals, 1 for the two edge intervals). This makes it easier to handle the cases in which a region is densely populated and should re-balance its points to neighbouring regions.

Due to the presented advtanges, the z-order curve interval splitting is the method that was chosen as a point distibution strategy for the distributed PhTree.

[1] Image adapted from `http://www.scholarpedia.org/article/B-tree_and_UB-tree`

Figure 3.7: Example intervals on the 3bit z-order curve

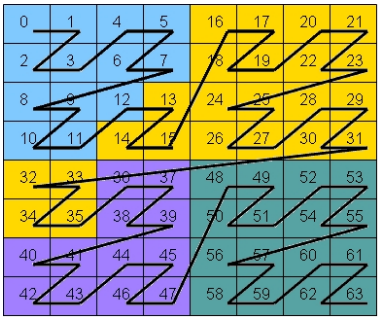


Figure 3.8: Example intervals on the 3bit z-order curve filling the 2D space

3.3 Architecture

The network architecture of the distributed PhTree is presented in Figure 3.9. The points are distributed to the hosts in the cluster using the z-order curve interval splitting method. The *key-mapping* is stored on the configuration server and is known to all of the nodes and the clients of the system. The configuration server uses a notification-based model and is responsbile for notifying the indexing hosts and clients whenever any changes occur. This model reduces the amount of network traffic within the cluster, as nodes and clients only need to read the *key-mapping* on start-up and do not need to poll the configuration server for changes.

Clients of the index first connect to the configuration server, retrieve the *key-mapping* and register to receiver notifications if the mapping changes. After receiving the mapping, clients can begin to send queries to the indexing nodes within the cluster. Clients decide which hosts need to the contacted for a query based on the type of the query and the *key-mapping*. This model is efficient as it allows the clients to decide the hosts to be queried based on local data. Futhermore, queries concerning more indexing nodes can be executed by sending network requests in parallel to all of the involved nodes. A potential drawback of this model is that the configuration server could become the bottleneck in a system with many indexing hosts or many connected clients. This issue can be addressed by using replicating the configuration data to a cluster of configuration servers. An alternative solution would be to store only a partial representation of the mapping in each indexing node and allow queries to be ”hop” from one node to the other until they reach the destination machines, as in the case of the prevously discussed distributed hash tables. However, the extra latency introduced by repeated ”hops” with the cluster makes this approach ill-suited for latency-sensitive applications like a distributed-index.

As the data is split according to z-order curver intervals, each indexing hosts only needs to maintain two connections to other hosts in the cluster (or 1 for the hosts holding the edge intervals). This means that the amount of memory used by the hosts to store cluster membership information remains relatively constant, even if the number of the nodes in the cluster increases.

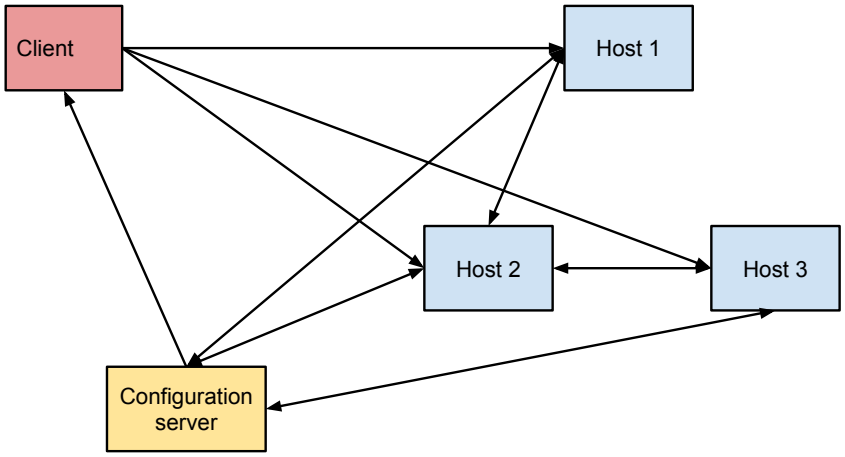


Figure 3.9: The architecture of the distributed PhTree

3.4 Algorithms

This section will discuss how the *key-mapping* is created from the Z-order curve intervals and how the queries are executed based on the key-mapping.

3.4.1 Key-mapping design

The *key-mapping* used by the distributed PhTree assigns the points based on the Z-order curve intervals. To determine the host responsible for a certain multi-dimensional point, one would have to map that point to the Z-order curve and determine the interval that contains it. There are two alternative for designing the *key-mapping* based on the Z-order curve intervals:

1. Store the Z-intervals directly as pairs of Z-values and perform 1-dimensional interval matchings to determine the hosts that have to be contact for certain queries. The operation is linear in the number of hosts, for both point and range queries, as the Z-values of the query points need to be compared with the Z-values of the intervals. The correctness of this approach stems from the fact that the region represented by a d-dimensional hyper-rectangle [X, Y] [2] is included in the set of d-dimensional regions corresponding to Z-interval $[Z_X, Z_Y]$.

2. Store the hyper-rectangles from the original space that correspond to each Z-order interval and perform intersections between the query points/rectangles from the original space and the hyper-rectangle of the interval regions. This operation takes $O(n*w*d)$, as each region contains at most $O(w*d)$ hyper-rectangles.

Even if the first solution is asymptotically faster than the second one for finding the hosts to query, the first approach attempts to query a much larger area in the original space than

[2] Where X is the top-right corner of the hyper-rectangle and Y is the bottom-left corner of the hyper rectangle

the second one. Therefore, the first method can potentially generate many more network requests than the first one, especially in a system with a large number of online indexing hosts. Moreover, the intersection operation itself is performed on the client, in-memory, and does not generate a network request. Because of these reasons, the second approach is preferable to the first approach in a system looking to minimize the number of network requests associated to each query, like the distributed PhTree.

[8] presents an algorithm to decompose a Z-order interval into a set of at most $O(w * d)$ rectangles. These rectangles are then stored in a multi-dimensional data structure providing hyper-rectangle intersection operations.[3]

3.4.2 Point queries

Point queries involve operations on a single query point, the insertion or removal of a key-value pair, obtaining the value associated with a key value pair or changing the key associated with a key-value pair. For the point queries, the client only needs to determine which host to send the request to. This is done by determining the intersections between the query point and the hyper-rectangles corresponding to the Z-order intervals. In the case of the *get()*, *put()* and *delete()* operations, the client sends the request to the hosts responsible for the query point and returns the result of the operation. In the case of the *updateKey()* operations, it is possible that a different host is currently responsible for the new key of the value associated with the query point. In that case, the *updateKey()* operation is performed in two steps: first the client removes the old key from the host responsible for it, and the inserts the new key to the proper host.

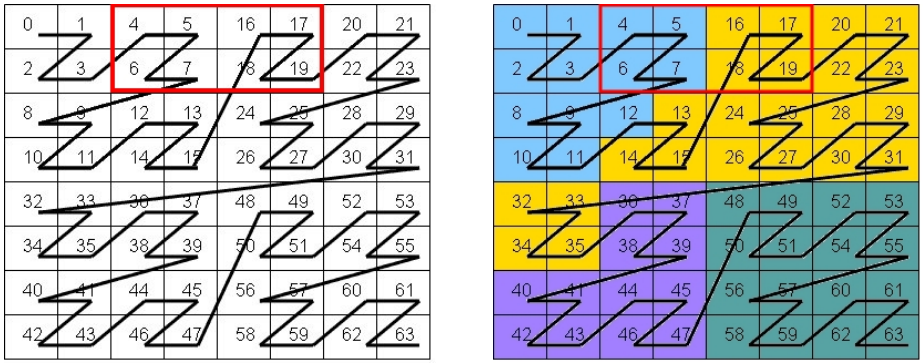
3.4.3 Range queries

Range queries have the form *range_query(bottom-left, top-right)*. Both of the received arguments are multi-dimensional data points and the operations returns all of the points in the index that fall inside the multi-dimensional rectangle defined by these two points. These operations are resolved in the following manner on the client. First, the client performs an intersection between the hyper-rectangle range and the hyper-rectangles associated with the z-order intervals contained in the *key-mapping*. It then send requests to all of the hosts whose regions were intersected, receives and combines the results. It is important to note that each host returns a list of point that are already ordered via z-ordering and because the nature of the z-order interval split, these lists do not need to be merged, but only concatenated.

A range query can generate up to $O(n)$ network requests, as in some cases, all of the hosts might need to be contacted. However, as previously also mentioned in section 3.4.1, the *key-mapping* performs the intersection between the query-rectangle and the regions corresponding to the Z-order intervals, instead of performing the intersection between the z-order interval of the query range and the z-order intervals of the machines. This guarantees that hosts which *cnnot* contain points in the query range will not be queried. It is however possible that some hosts will not contain any points falling in the query hyper-rectangle.

Figure 3.10a illustrates an example range query. It shows the range defined by the bottom

[3] This implementation actually uses a variant of the in-memory PH-tree for this function.



(a) The 2-dimensional range $[(0, 2), (1, 5)]$ overlayed on the Z-order curve.

(b) The intervals specific in the previous section overlayed over the previous picture.

Figure 3.10: Illustration of a k nearest neighbour query execution

left point $[(0, 2)]$ and top right point $[(1, 5)]$ overlayed against the 3-bit Z-order curve. The Z-order interval corresponding to this range is $[4, 19]$ and the 2 dimensional region associated to it includes the $[(0, 2), (1, 5)]$ rectangle. Figure 3.10b also displays the regions associated to each host according to the split presented in Figure 3.7. In this case, only the hosts responsible for the blue and yellow regions need to be contacted by the client.

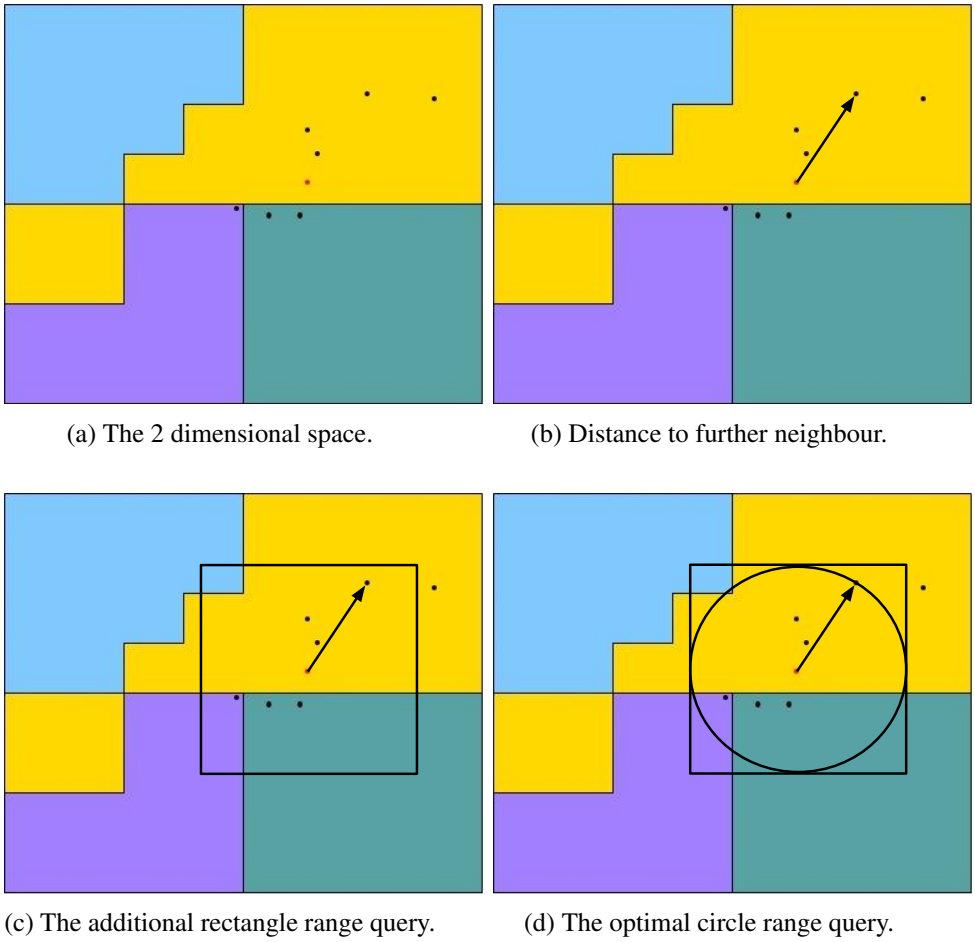
As previously mentioned, because the z-order range associated with the query range corresponds to a larger regions than the query hyper-rectangle, especially in high dimensional spaces,

3.4.4 Nearest neighbour queries

Nearest neighbour queries return the k nearest points, according to the Euclidean distance, to an arbitrary query point, thus having the form *knn_query(q, k)*. The query point does not need to be stored in the index. The execution of a nearest neighbour query follows the following steps:

1. The client first determines which hosts is responsible for the query point q and forward the query to this host.
2. After receiving the result, the client determines finds the furthest neighbour fn of the query point q according to the Euclidean distance.
3. The client computes the distance $d = dist(q, fn)$ and performs a range query on the range $[q - d, q + d]$.
4. The final result of the nearest neighbour query consists of the k points from the result of the previously executed range query that are closest to q.

The additional range query is needed because the z-order splitting, or any other geometrical splitting, does not guarantee that closest k neighbours of a point will be placed in the same



(a) The 2 dimensional space.

(b) Distance to further neighbour.

(c) The additional rectangle range query.

(d) The optimal circle range query.

Figure 3.11: Illustration of a 3 nearest neighbour query execution in a 2 dimensional space.

regions as the point itself. This range query ensures that points which are closer to q than fn, but stored in a different region, will also be returned.

It is sometime possible that the initial query sent to the host responsible for q will return fewer than k points. This will generally only happen when the cluster contains very few points. In this case, the queried area should be iteratively increased until at least k points are returned. This can be done by computing the distance d and iterative increasing it until the range query $[q - d, q + d]$ returns at least k points.

While this approach for finding the k nearest neighbours of a query point is correct, it is not the most efficient one. The main issues is that the additional range query searches for other neighbours in a hyper-cubing range. The optimal strategy would be to search for these neighbours in hyper-sphere of radius d centered at q. This circle is inscribed in the hyper-cube search by the range query, as illustrated in Figure 3.11. Therefore, the algorithm presented in this work can sometimes send unnecessary request to hosts responsible for the areas outside of the circle and still within the hypercube. This is generally not a problem in lower dimensional spaces, however the different in volume between the hyper-cube and the hyper-sphere increases gradually with dimensionality of the space.

3.4.5 Entry load balancing

As previously described, data sets which are skewed towards certain areas of the multi-dimensional space will cause some machines to store more points than the others. A load balancing algorithm is needed to balance the amount of entries stored across the cluster, such that each hosts stores roughly the same amount of entries.

Each host is assigned a threshold t corresponding to the maximum number of entries it can store. For simplicity, we assume that all the hosts in the system are identical and have the same t. The load balancing algorithm used by the distributed PH-tree assumes that values stored with each data points have roughly the same size, and such a threshold on the number of entries a hosts can store coincides with the amount of data it can store. The presented balancing algorithm can easily be modified by simply using the amount of occupied memory as a threshold.

Furthermore, the configuration server maintains the number of entries associated to each hosts, which will be referred to onwards as the *size* of the host. To improve scalability, the hosts do not update the size stored in the configuration server after each individual write operation, but only after a certain number of write operations were performed since the last update. An alternative solution would be update the size at a fixed time period. Each host maintains a cached copy of the size information, and is notified by the configuration server when a different host updates this information.

After the number of entries stored by a certain host reaches the balancing threshold t, that host will attempt to move some of the points it stores to a neighbouring host. This process is referred to as a *re-balanacing operation* and will change the Z-order intervals of both the host initiating the re-balancing and the host receiving the additional entries. Given a host h responsible for an arbitrary interval, the host responsible for the Z-order interval to the right will be called its *right neighbour*. Similarly, the host responsible for the Z-order interval to the left will be called the h's *right neighbour*.

The re-balancing algorithm proceeds as follows:

1. The *initiator* host checks the size of the hosts responsible for the intervals to the left and to the right of its Z-order interval. When the host responsible for the leftmost intervals initiates a re-balancing operation it only checks the size of its right neighbour, while the host responsible for the rightmost interval only considers re-balancing to its left neighbour.

2. The *initiator* selects the neighbouring host with the fewest entry and sends an initiate balancing message. If the neighbour is available to participate in the operation, the algorithm continues with the following step. If the neighbour is busy, the *initiator* will attempt to re-balance to the other neighbour. If that is not possible, the re-balancing operation fails and will be re-attempted at a later point.

3. The initiator sends a subset of its entries to the *receiver* host. This subset is a contiguous run of entries stored by the initiator.

4. After the entries have been moved, the *initiator* updates the *key-mapping* on the configuration server.

5. The initiator sends a commit message to the *receiver* host, notifying it that the operation was successful. After this message is sent, both hosts mark themselves as available for any other re-balancing operations.

This algorithm allows a host h to be part of only a single re-balancing operation at a moment in time. As soon as a host successfully initiates a re-balancing operation, both the *initiator* and the *receiver* mark themselves as busy and will refuse further re-balancing initiation operations until the current operation is finished. This allows up to $n/2$ balancing operations to run in parallel across the cluster, as each operation involves two neighbouring hosts.

Furthermore, clients currently balancing do not accept write requests. This is done to prevent the *initiator* host to accept updates to the section of z-order curve that is currently re-balanced to the *receiver* host.

4
Concurrency

4.1 Challenges

The first version of the PH-tree does not support concurrent access, and remains consistent only if accessed by a single thread at a time. With the current popularity of multi-core CPU's, concurrent access has the potential of providing a significant improvement in the throughput of the PH-tree. The addition of concurrent access presents the following challenges:

- Concurrent access strategies must guarantee that deadlock will not occur when the data structure is accessed by multiple threads.
- The chosen strategy should optimize the execution time for the most common operation, by minimizing for example, the amount of locks that have to be taken during for those operations. In case of an indexing data structure, read operations are more common than write operations.
- Different concurrent access strategies come with different consistency guarantees. For example, consider the situation in which two processes A and B work concurrently on a shared data structure. Process A starts a read operation at time t and process B a write operation at time $t + \epsilon$. The consistency model determines if process A might see the changes made by process B, if process B finishes the operation before A.

The following section presents the concurrent access strategies added to the PH-tree and discusses their consistency guarantees.

4.2 Concurrency strategies

4.2.1 Copy-on-Write

Copy-on-Write is an concurrent access strategy that allows multiple threads to access the same shared data structure. When one of these threads needs to modify the shared data structure, it first makes a copy of it, performs the modifications and then updates it with its modified copy. Each write operation creates a new version of the shared data structure, making it immutable. This strategy allows one writer thread and an arbitrary number of reader threads to work on the same data in the same time. If multiple writer threads were to access the data structure, each would end up with its updated copy and the last to replace the shared data structure with the copy would overwrite the changes of the other writer.

In case of the trees, writer threads do not have to copy the full tree before applying their modifications. It is sufficient to create a copy of the node that is update and to all of its parents nodes, up to and including the root of the tree. The number of nodes copied for each update is equal to the maximum height of the tree. In case of the PH-tree, this number is equal to the number of bits needed to represent the values on each dimensions, independent of the number of nodes or entries currently stored in the tree.

Figure 4.1 illustrates the execution of write operation. For simplicity, a general binary tree is pictured here, as the operation is executed in a similar manner on a PH-tree. In this example, node G needs to be updated. To achieve this, all of the ancestors of G are copied and the copy of G, G' is updated. Finally, the root pointer of the tree is changed from A to A' atomically. It is important that the root swap is performed in an atomic manner, such that any readers concurrently accessing the tree will either access the old or the new version of the tree.

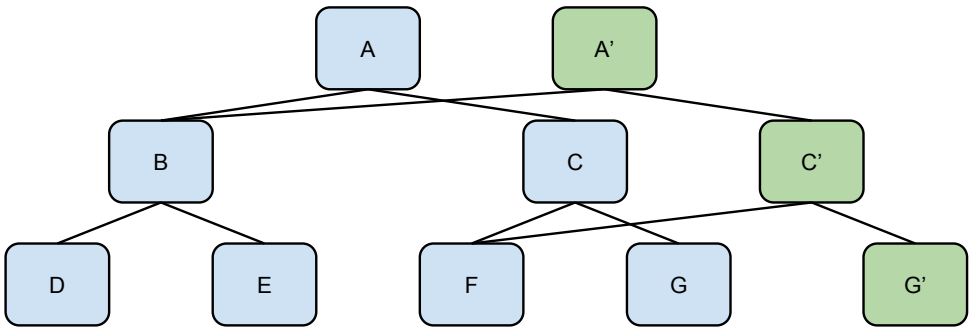
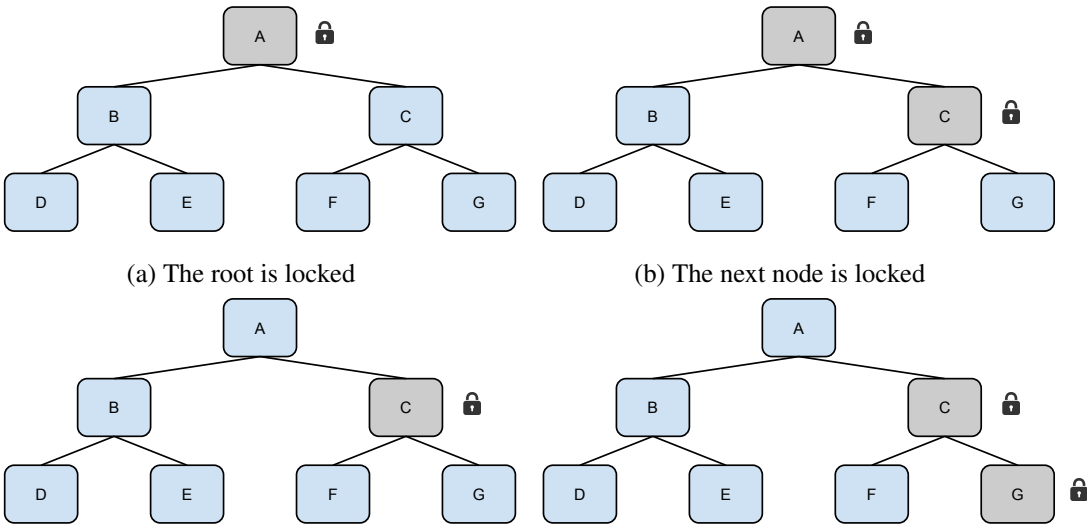


Figure 4.1: Illustration of Copy-on-Write on a binary tree where the node G has to be written

There are several possibilities to handle multiple writer threads using the Copy-on-Write strategy. One approach would be allow multiple threads to access the tree in the same time. This can be done by implementing the atomic root swap using a Test-And-Set atomic operation, essentially turning each write operation into a transaction. In case the root was changed since the thread has started the operation, the atomic Test-And-Set fails and the thread will need to restart the operation. The main drawback of this approach is that the commit fails even if the changes to the tree are made to different parts of the tree.



(a) The root is locked

(b) The next node is locked

(c) The root can now be unlocked

(d) The final node is locked

Figure 4.2: Illustration of hand-over-hand locking descent to node G

4.2.2 Locking

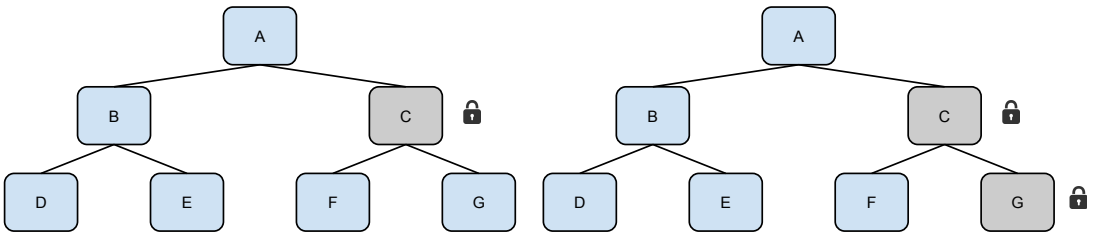
A very simple way to handle concurrent writer threads is to use global lock that each thread must take before the execution of an operation. If the lock is a mutex, the tree can only be accessed by a single thread, and if the thread is a read-write lock, the tree can be accessed by either a single writer or several reader threads. This coarse-grained locking strategy is not very efficient and therefore only useful as a baseline implementation during the performance evaluation.

Fine grained locking strategies perform locking at the level of tree nodes and can allow multiple writer threads to modify the tree in the same time. Two popular strategies are described in the following sections. It is important to note that the locking strategies provide a more relaxed consistency model. Therefore, an iterator which starts to traverse the tree at time t and finishes at time $t + d$ will potentially see all of the changes made by different threads in the time intervals $t + d$.

Hand-over-Hand locking

Hand-over-hand locking, also called *lock-coupling*, is a fine-grained locking strategy according to which a lock for a node can only be acquired if the parent lock is also held. When traversing the tree according to this strategy, at most two lock will be held by a thread at a time. An additional lock needs to be acquired before acquiring the lock of the root of the tree, to prevent conflicts on the creation of the first node of the tree.

Figure 4.2 shows the sequence of events happening when a thread needs to update a node. The node to be updated in this figure is node G. The thread first acquires the root, descends to node C and acquires that lock as well. The root can be unlocked before descending to node G and acquiring that lock. After the changes are done, the lock of node G is unlocked.



(a) The parent of the target node is locked (b) The target node is locked

Figure 4.3: Illustration of optimistic locking descent to node G

This strategy allows multiple threads to modify the different subtrees of the tree. The main drawback of this approach is that if the non-leaf nodes are modified, subtrees of those nodes cannot be modified of a different thread until the current thread finishes the update operation. Secondly, there will be a high contention for the locks on the higher levels of the tree, which could negatively affect performance for write-heavy workloads.

Optimistic locking

Optimistic locking is an alternative fine grained locking strategy which seeks to minimize the number of locks which are take during a write operation. The basic idea is to traverse the tree without acquiring any locks until the node to be updated is reached. Once at the target node, the writer thread acquires the lock of the parent and the lock of the current node. After this happens, the write operation can continue only if both of these two nodes have not yet been removed from the tree and the current node is still a child node of the parent node. If any of these checks fail, the write operation fails and the current thread begins another attempt by starting a new traversal of the tree.

To ease the implementation, each node has an additional removed flag, which indicates if certain node has been removed from the tree. When a thread removes a node, it sets this flag to true before releasing the lock of the node.

One advantage of this method is that in a large tree, re-tries are generally rare as threads do not generally attempt to update the same node. Moreover, a thread holding the lock of a certain node does not block other threads from performing update operations on the node's subtrees, unlike the *hand-over-hand* strategy.

It is important to mention the fact the in the case of the PH-tree, all update operation to a single node are not executed in an atomic fashion, which can cause reader threads to perform reads from nodes which are in an inconsistent state. This can be avoided by protecting the nodes with a read-write lock and requiring the read operations to acquire the read lock of a node before performing a read. However, the latency of the read operation is increased if readers need to acquire locks. Another solution is to add some copy-on-write logic to the write operations. Specifically, before performing a write operation on certain node, the writer thread first creates of copy of the node, performs the update on the copy and then atomically replaces the parent node's reference to the node with the copy. As the writer thread holds the lock on the parent node, it is guaranteed that no other writer threads will overwrite this

change. Furthermore, reader threads will either access the old version of the node or the updated copy.

The PH-tree supports both a full COW concurrency strategy and the fine grained locking strategies. The optimistic locking strategy is generally faster than the hand-over-hand locking strategy, but can lead to starvation when many threads need to update the same node or nodes which are close together. The choice between the COW strategy and the fine grained strategies comes down to the choice between the consistency guarantees provided by them. For example, snapshots of the tree can only be obtained if the tree is using the COW strategy.

5 Implementation

Meta - will be removed after editing. Present the implementation architecture and the techologies used.

5.1 System description

Meta - will be removed after editing. Describe the system, include class/deployment diagrams.

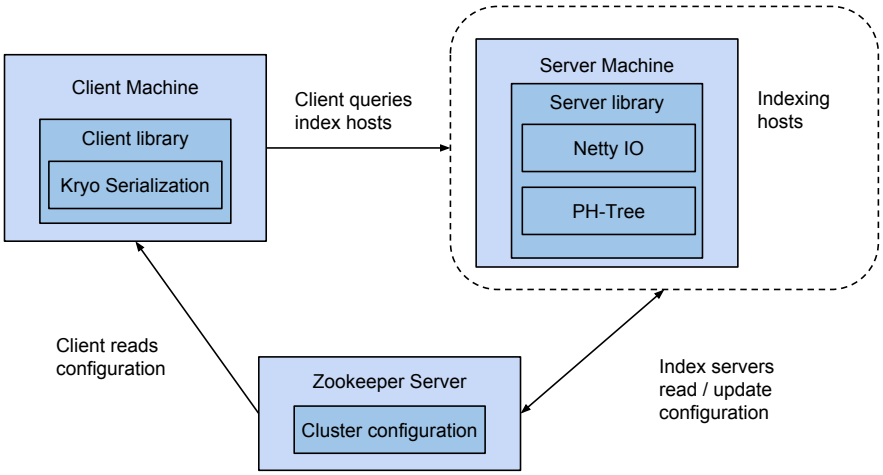


Figure 5.1: The implementation architecture of the distributed PhTree

5.2 Technologies

Meta - will be removed after editing. Describe the technologies used, the reasons for which these technologies were chosen and any alternatives.

6
Evaluation

Meta - will be removed after editing. Explain how the system should be evaluated, present and explain the benchmarks

7 Conclusions

7.1 Contributions

This work presents a complete solution from implementing a distributed and parallel multi-dimensional index, by presenting both a distribution architecture and operation parallelization solutions for individual machines. The main contributions are the following:

The distribution architecture partitions the multi-dimensional space in a manner that allows fast point operation and efficient range and nearest neighbour queries. The proposed balancing algorithm ensures that all hosts will eventually hold a comparable number of entries, even in the case of very skewed input distributions. The presented distribution solution can be easily adapted for other types of index keys, as long as the key domain can be mapped to 1 dimensional line, allowing the constructing of efficient and scalable range-queryable, distributed key-values stores.

The in-memory PH-tree was updated to support concurrent access according to two consistency paradigms. The Copy-On-Write approach offers full consistency guarantees, providing snapshot semantics to queries and isolating readers from any further modifications to the tree. For situations where full consistency is not needed, the optimistic locking approach offers relaxed consistency semantics and provides improved write throughput by allowing concurrent write operations. Furthermore, none of presented approaches require the readers to acquire any locks and thus add no overhead to the execution of the read operations.

7.2 Future work

One aspect of the distributed PH-tree that could be improved is the execution of the distributed nearest neighbour queries. As previously mentioned in section 3.4, the nearest neighbour

algorithm first obtains the k nearest neighbours from the host responsible for the query point and then procees to run a range query on the cluster to check for any closer points on different hosts. The additional check is done by checking a hyper-cubic section of the initial space. A more efficient approach would be to check a hyper-spheric region centered in the query point. This improvement could reduce the number of hosts queried on the second step of the nearest neighbour queries and could potentially improve the execution time of such queries.

From the point of view of the balancing algorithm, hosts participating in re-balancing operation do not accept write requests, to simplify the management of the mapping between the points and the hosts. The balancing algorithm could be modified to allow the hosts to accept write requests to a separate buffer and integrate the buffer to the stored PH-tree after the balancing operation has finished.

The concurrency strategies presented by this work rely on locking and copy-on-write to allow concurrent access. Further work could focus on investigating potential concurrent access strategies using on atomic operations.

List of Figures

2.1 2 dimensional 4bit PhTree containing the points: (0000, 1000), (0011, 1000), (0011, 1010). 6

3.1 Distributing the points based on a hash function 12
3.2 Equal geometric space splitting for 2 hosts and 2 dimensions 12
3.3 Equal geometric space splitting for 4 hosts and 2 dimensions 12
3.4 13
3.5 Equal geometric space splitting for 4 hosts and 2 dimensions 13
3.6 Z-order space filling curve filling the 2 dimensional space. 14
3.7 Example intervals on the 3bit z-order curve 15
3.8 Example intervals on the 3bit z-order curve filling the 2D space 15
3.9 The architecture of the distributed PhTree 16
3.10 Illustration of a k nearest neighbour query execution 18
3.11 Illustration of a 3 nearest neighbour query execution in a 2 dimensional space. 19

4.1 Illustration of Copy-on-Write on a binary tree where the node G has to be written 24
4.2 Illustration of hand-over-hand locking descent to node G 25
4.3 Illustration of optimistic locking descent to node G 26

5.1 The implementation architecture of the distributed PhTree 29

List of Tables

Acknowledgements

Bibliography

[1] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[2] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. *SIGPLAN Not.*, 45(5):257–268, January 2010.

[3] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 329–342, New York, NY, USA, 2014. ACM.

[4] Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[5] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), May 2007.

[6] Haojun Liao, Jizhong Han, and Jinyun Fang. Multi-dimensional index on hadoop distributed file system. In *Networking, Architecture and Storage (NAS), 2010 IEEE Fifth International Conference on*, pages 240–249, July 2010.

[7] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Prefix hash tree: An indexing data structure over distributed hash tables. Technical report, 2004.

[8] Toms Skopal, Michal Krtk, Jaroslav Pokorn, and Vclav Snsel. A new range query algorithm for Universal B-trees. *Information Systems*, 31:489–511, 2006.

[9] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[10] Tilmann Zäschke, Christoph Zimmerli, and Moira C. Norrie. The ph-tree: A space-efficient storage structure and multi-dimensional index. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 397–408, New York, NY, USA, 2014. ACM.

[11] Chi Zhang and Arvind Krishnamurthy. Skipindex: Towards a scalable peer-to-peer index service for high dimensional data. Technical report, Princeton Univ, 2004.