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

27th March 2015

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

globis

Abstract

Here comes the abstract.

Contents

1 Introduction

1.1 Motivation

Multi-dimensional data is widely used today, for example 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 possibility 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 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 PH-tree [11] is a new multi-dimensional data structure based on the quadtree. In addition to providing support for complex queries, the PH-tree 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 structures need to store and manage more data than would fit in the main memory of a single machine. However, traditional multi-dimensional indexes like the kD-tree and quadtree 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 per CPU 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 operations. 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 PH-tree to run on distributed cluster of machines and modifying it to support concurrent operations.

1.2 Objectives

We has highlighted two challenges currently faced by indexing systems: *high storage requirements* and *support for concurrent access*. This work proposes the distributed PH-tree, a version of the PH-tree 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 PH-tree 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 PH-tree 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 PH-tree, 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 PH-tree from the point of view of a distributed system is presented in **Chapter 3**. This chapter presents the chosen data distribution strategy and touches on the possible alternatives and the consequences of this choice. Additionally, we provide 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 PH-tree is presented in **Chapter 4**. Several concurrent access strategies are discussed, in terms of 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 PH-tree 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. We also give an outlook of the potential extensions.

2
Background

The first part of this chapter analyses the single-threaded PH-tree, 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 PH-tree

The PH-tree [11] is a novel multi-dimensional indexing structure that focuses on fast retrieval, fast update 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 PH-tree 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 PH-tree 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 PH-tree has the following important properties:

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

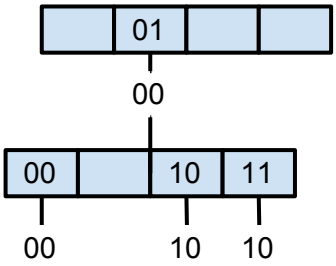


Figure 2.1: 2 dimensional 4bit PH-tree containing the points: (0000, 1000), (0011, 1000), (0011, 1010).

- **Constant maximum 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 PH-tree is 64. This property is important as *the maximum height of the PH-tree is independent of the number of points it stores*.
- **Balancing is not needed**. The PH-tree 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 PH-tree 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 operations** 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 PH-tree can store limited by the memory available on a single host. A distributed version of the PH-tree would be able to store a much larger amount of entries and would be able to make use of 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 PH-tree 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 distribution architecture and 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 [10], 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 node 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 [8] 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 [12], 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[1] and use query models like MapReduce[2] as building blocks for the implementation of range and nearest neighbour queries.

2.2.2 Concurrent data structures

Concurrent data structures are currently a very active area of research. 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 either the data structure itself, called *course-grained locking*, or to parts of the data structure, which is also called *fine grained locking*. Lock-free solutions use either atomic primitives, like *compare-*

[1] http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[2] http://en.wikipedia.org/wiki/MapReduce

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* [7], 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-operations. 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 PH-tree from the distributed point of view. It presents the challenges and the possible design choices for extending the PH-tree to run in a distributed setting, discusses the system architecture and describes the algorithms used for the distributed queries. Concurrency related issues are not tackled in this chapter as they are addressed in Chapter 4.

3.1 Challenges

While extending the PH-tree to be a distributed index increases 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 identical nodes[1], nodes which have the same hardware resources available, all nodes should store an 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 PH-tree needs to provide support for 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.

[1] Cluster machines will be referred to as nodes, or hosts, throughout this work.

- **Ensuring cluster scalability**. To support very large storage requirements, the system should handle a large number of online-indexing nodes. Furthermore, to improve 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. Online cluster re-configuration is challenging because of the data migrations that it entails in order to achieve a balanced storage load. New nodes added to the system start off as being empty and should receive some entries from the currently running active nodes. The data received by these empty nodes should preserve locality. Nodes that are being removed from the system need to first make sure that the entries currently stored are moved to different nodes in the system in a manner that preserves locality.
- **Minimizing node information**. As the PH-tree is held in-memory, it is important for the indexing nodes to be efficient in managing their memory, to maximize the number of entries they can store. Maintaining a connection between two node over a network generally requires an open socket on both participating nodes. As active sockets take up system memory and CPU cycles, it is 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 consumption.

Other challenges posed by distributed indexes, and distributed systems in general, are availability 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. Ensures 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 PH-tree containing all of the entries stored by the cluster, each node of the PH-tree 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 PH-tree depth is limited by the 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” corresponds to an additional 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 PH-tree and data points are assigned to machines according to some mapping. This mapping will be referred to as the *key-mapping*, as it maintains 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 points, 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 it 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 PH-tree’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 cluster. 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.

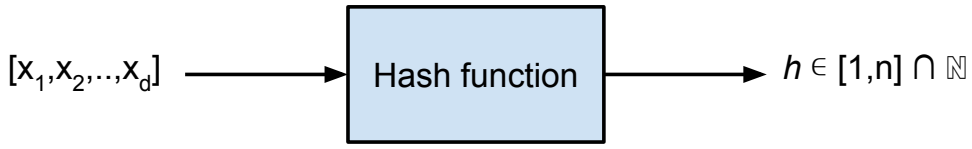


Figure 3.1: Distributing the points based on a hash function

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 partition the original space into a set of contiguous regions and assign each region to a host 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.

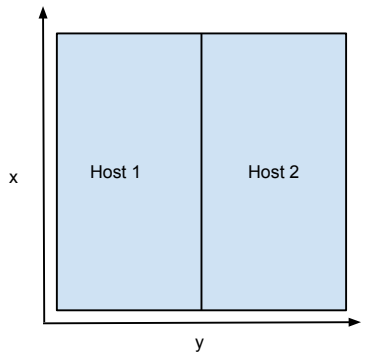


Figure 3.2: Equal geometric space splitting for 2 hosts and 2 dimensions

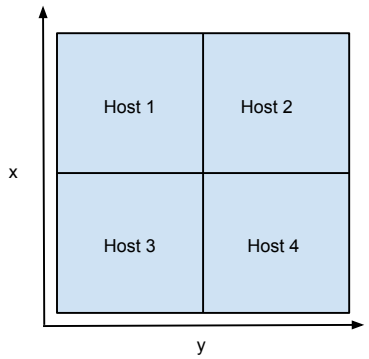


Figure 3.3: Equal geometric space splitting for 4 hosts and 2 dimensions

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 case, range queries can be dispatched to a single host. It is generally preferable to split

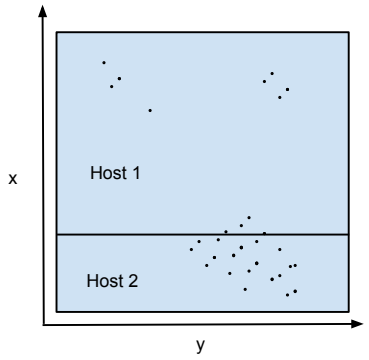


Figure 3.4: Potential partitioning of the space for a skewed input distribution.

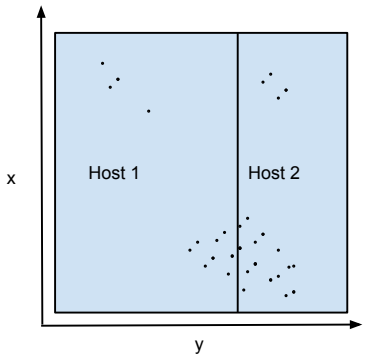


Figure 3.5: Another potential partitioning for the same skewed input distribution.

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 partitioning 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 splitting 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 region 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 PH-tree stores the multi-dimensional points according to the Z-ordering, by mapping the initial space to the 1-dimensional Z-order curve. This

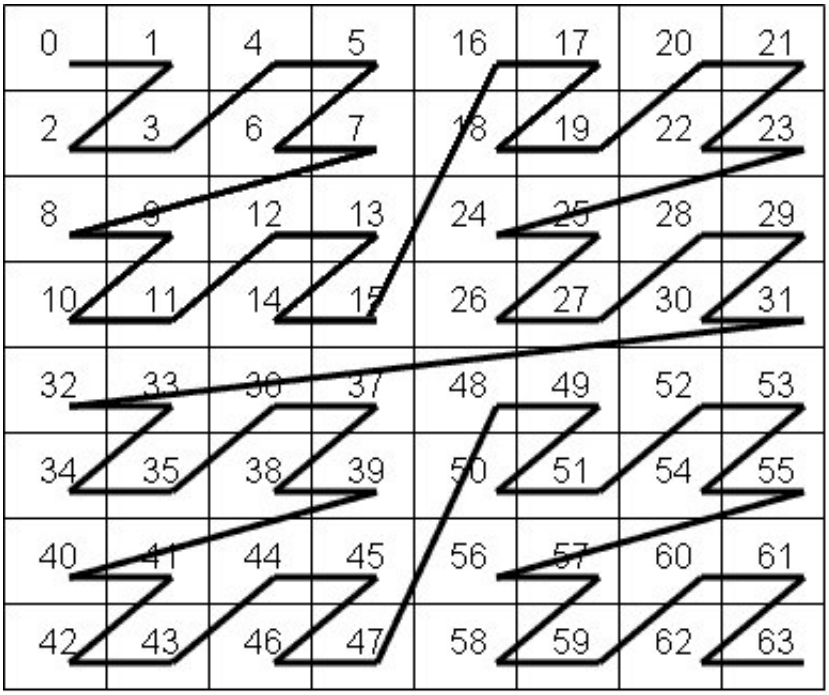


Figure 3.6: Z-order space filling curve filling the 2 dimensional space.

curve fills in the initial space in a contiguous and complete manner. Figure 3.6[2] 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 partition the z-order curve into contiguous intervals. Figure 3.7 shows the 1 dimensional Z-order curve corresponding to the same 3 bit 2-dimensional space from figure 3.6, 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. For each interval there are at most $O(d * w)$ of these hyper-rectangles, where d is the number of dimensions and k is the number of bits needed to represented the values in each dimension. [9] 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 advantages, the Z-order curve interval splitting is the method that was chosen as a point distribution strategy for the distributed PH-tree.

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

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

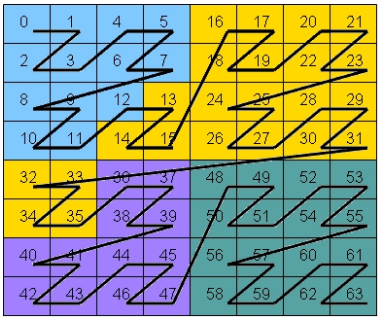


Figure 3.8: Example intervals on the 3bit Z-order curve filling the 2D space

3.3 Architecture

The network architecture of the distributed PH-tree 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 responsible 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 receive 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. Furthermore, queries concerning more indexing nodes can potentially 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 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 previously 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 curve 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.

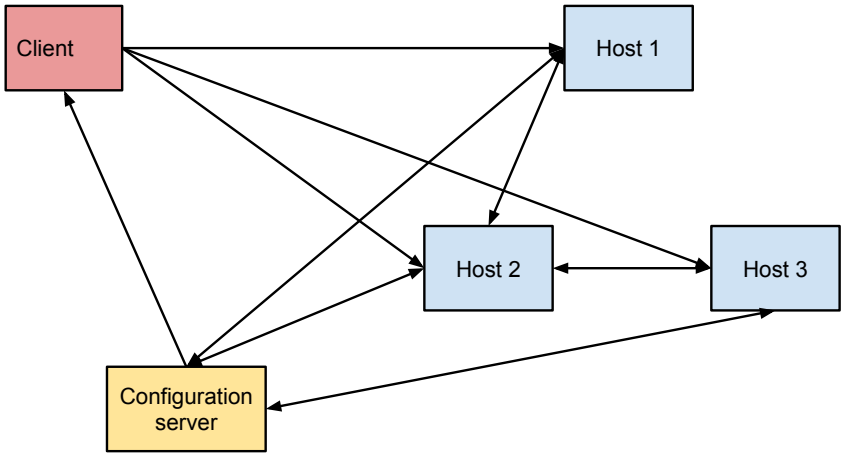


Figure 3.9: The network architecture of the distributed PH-tree

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 PH-tree 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] [3] is included in the set of d-dimensional regions corresponding to the 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. The complexity of this operation is $O(n * w * d)$, 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

[3] 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 PH-tree.

[9] 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.[4]

3.4.2 Point operations

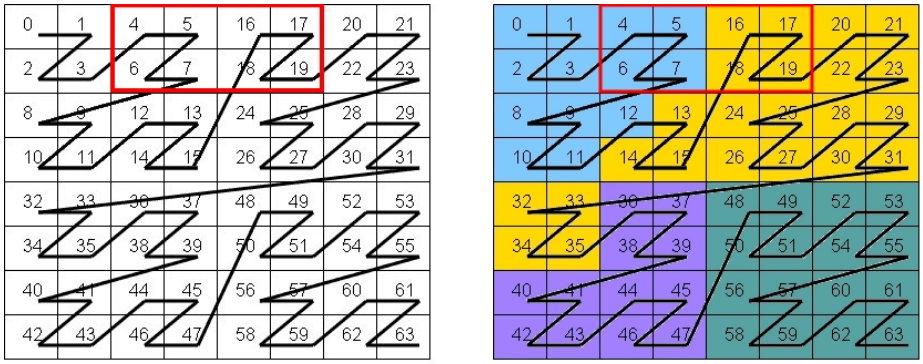
Point operations involve a single query point, like the insertion or removal of a key-value pair, obtaining the value associated with a key value pair or changing the key associated with a key-value pair. For the point queries, the client only needs to determine which host to send the request to. This 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 this operation 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 *cannot* 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, even if the regions they are responsible for intersects with the query range.

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



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

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

Figure 3.10: Illustration of a range query intersected with the host intervals

Figure 3.10a illustrates an example range query. It shows the range defined by the bottom left point $[(0, 2)]$ and top right point $[(1, 5)]$ overlaid 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.

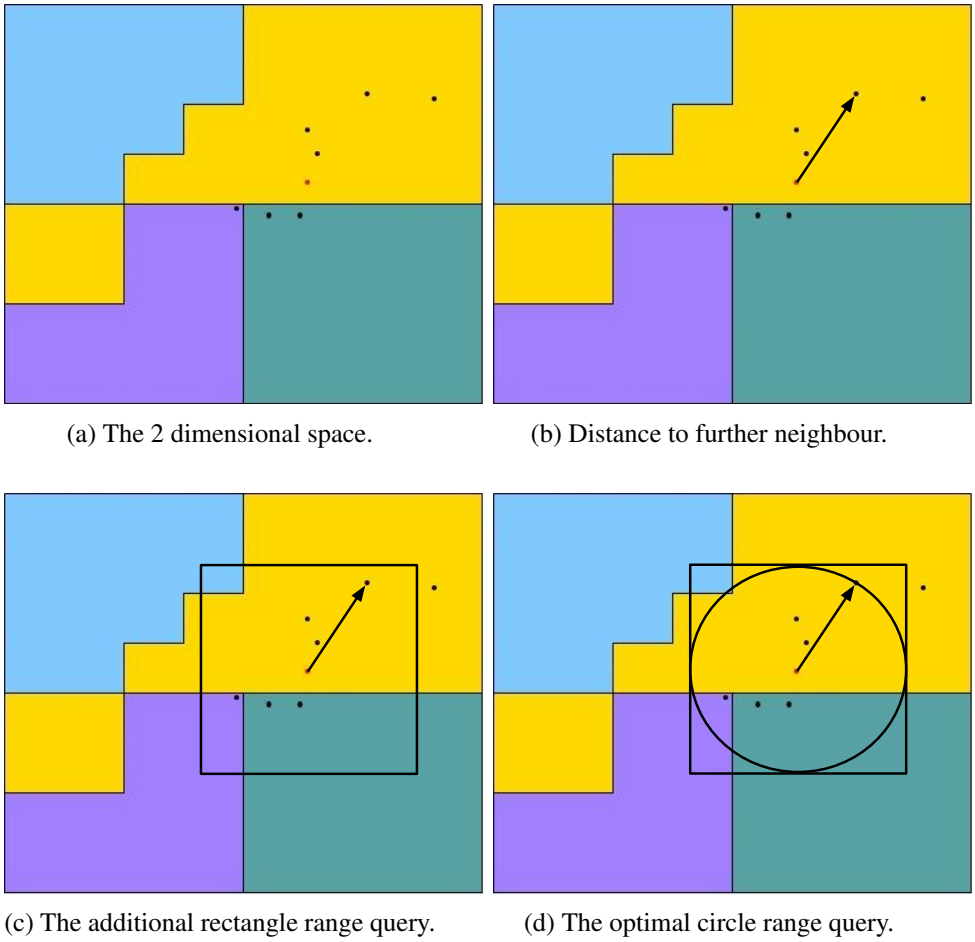
As previously mentioned, because the Z-order range associated with the query range corresponds to a larger regions than the query hyper-rectangle, especially in high dimensional spaces,

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 has the following steps:

1. The client first determines which hosts is responsible for the query point q and forwards the query to this host.
2. After receiving the result, the client determines 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 3.11d. 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 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 *left 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 entries 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.

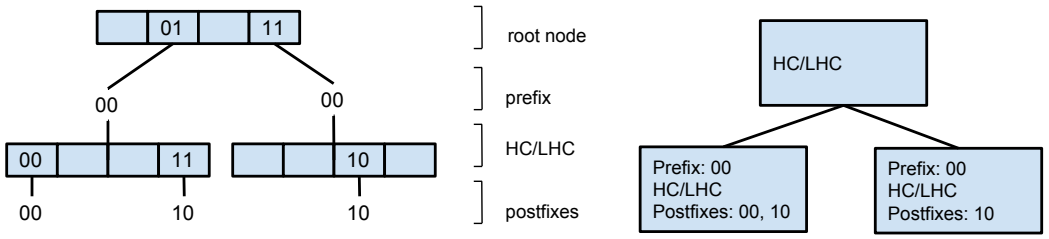
Furthermore, clients currently balancing do not accept write requests. This is done to prevent the *initiator* host to accept updates to the section of Z-order curve that is currently re-balanced to the *receiver* host.

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 PH-tree remains consistent for any number of threads that access it and for any arbitrary interleaving of the executions of these thread. Furthermore, deadlock should 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 and should be prioritized by the concurrency strategy from the response time point of view.
- 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 PH-tree also supports an iterator over all of the entries or over all of the entries contained within a specified hypercube. The concurrency strategy should allow iterators to traverse a PH-tree, even if the tree is concurrently modified by other threads.



(a) Logical structure of a PH-tree.

(b) Node structure of the same PH-tree

Figure 4.1: A 2 dimensional PH-tree containing three 4-bit entries: (0000, 1000), (0011, 1010), (1011, 1000). The values are omitted.

The concurrency model corresponding to the concurrency strategy used will specify which entries of the tree are seen by the iterator.

4.2 PH-tree structure

The PH-tree stores key-value pair entries. The keys are multi-dimensional data points and the values are optional. The entries are stored as bit strings, to exploit any prefix sharing between the bit strings of the keys and reduce the storage requirements.

Figure 4.1a illustrates a 2 dimensional PH-tree containing the entries (0000, 1000), (0011, 1010), (1011, 1000). Figure 4.1b shows the corresponding node structure for the same PH-tree. Each node contains a *prefix*, a *a hypercube* (HC) and a set of *postfixes*. The *prefix* is a portion of the bit string common to all bit string portions stored in that node. A *postfix* is an ending portion of a single bit string. The *hypercube* is a data structure that stores references to *postfixes* or subnodes. Currently, the PH-tree supports multiple representation for the *hypercube*, either as an array (HC) or a sparse linear representation LHC. For example, the root node in Figure 4.1a has a *hypercube* storing references to two subnodes. The subnodes each have an associated *prefix* and a *hypercube* storing links to postfixes.

This structural representation of the PH-tree has the following important consequences:

- All information stored in a node cannot be read or written in an atomic manner, as the prefix, the hypercube and the postfixes need to be read or written in separate instructions.
- Adding a new subnode to a node, as well as removing or replacing it can be done atomically.

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

4.3 Concurrency strategies

4.3.1 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 updated and copies of all of its ancestor 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 4.2 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.

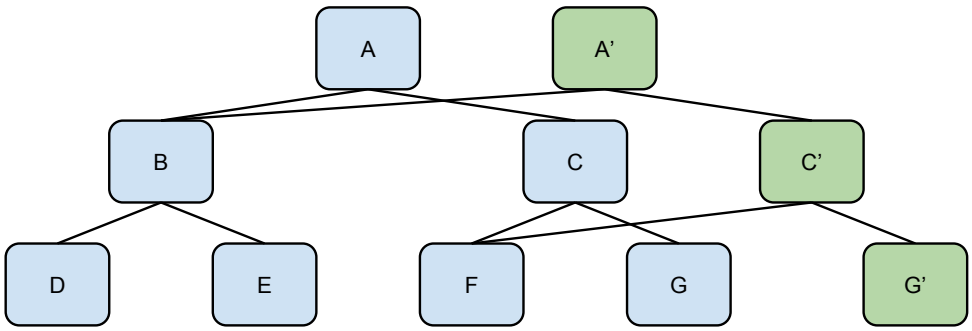
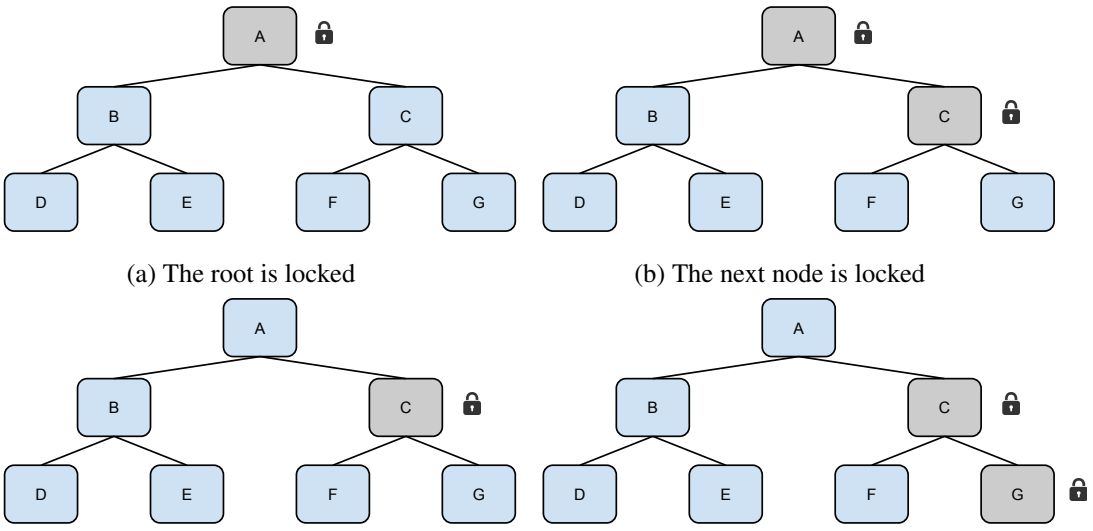


Figure 4.2: Illustration of Copy-on-Write on a binary tree where the node G has to be written

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 are made to different parts of the tree.

Iterators created on a Copy-on-Write PH-tree essentially traverse a snapshot of the tree taken



(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 4.3: Illustration of hand-over-hand locking descent to node G

on the iterator creation time t. If an iterator is create at time t, it will view all of the entries that were contained in the tree at time t, even if these entries were removed from the tree before the iterator finishes. Moreover, iterators will never see any entries that may have been added to the PH-tree after time t.

4.3.2 Locking

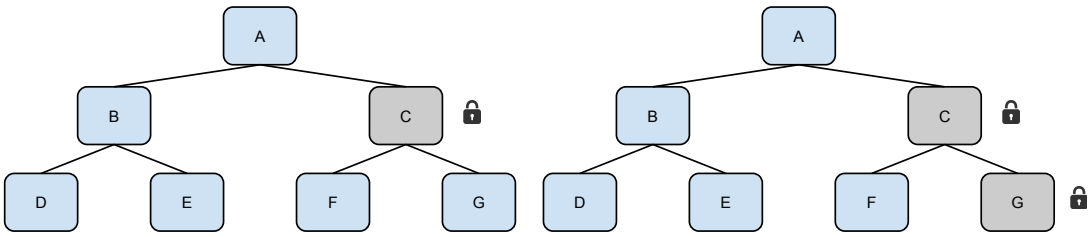
A very simple way to handle concurrent writer threads is to use a global lock that each thread must take before the execution of an operation. If the lock is a mutex, the tree can only be accessed by a single thread, and if the thread is a read-write lock, the tree can be accessed by either a single 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.

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 4.3 shows the sequence of events happening when a thread needs to update a node.



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

Figure 4.4: Illustration of optimistic locking descent to node G

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.

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 speed up checking if a node is still reachable from the root nod, 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, as the sub node swap is done in an atomic manner.

In the case of the fine-grained locking concurrency strategies, iterators running the PH-tree operate according to a relaxed consistency semantics. Therefore, an iterator created at time t, which finished traversing the tree at time e will see all of entries that were part of the tree a time t and are still part of the tree at time e. Moreover, entries inserted or removed in the time interval $[t, e]$ may also be seen by the iterator. It is however guaranteed that all of the entries seen by the iterator are visited according the Z-ordering.

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

5.1 System architecture

The architecture of the distributed PH-tree is shown in Figure 5.1 . The implementation consists of two artifacts: the *client library* and the *server library*. Both of these libraries have been implemented in the Java programming language and are the main artifacts of this work.

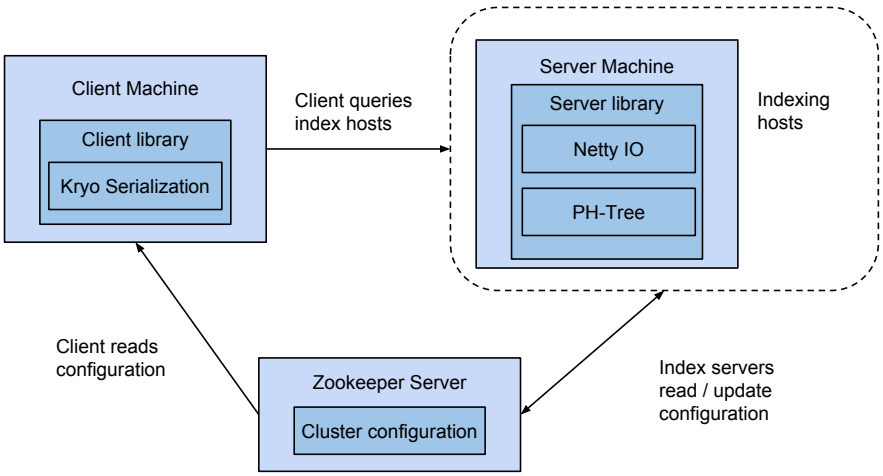


Figure 5.1: The implementation architecture of the distributed PH-tree

The *client library* is deployed on the clients and contains the code necessary to connect to the configuration server, retrieve the cluster information and perform operations on the cluster.

The *client library* also contains a dependency to the *Kryo* library, a Java serialization library that is used to encode the values stored on the PH-tree before the insert operation and decode the values retrieved from the server. Clients cannot modify the cluster information and can only subscribe to be notified by the configuration server when indexing hosts join or leave the cluster and when the Z-order curve intervals changes as the result of a rebalancing operation.

The *server library* contains the index middleware logic and needs to be deployed on the indexing hosts. The *server library* implements a multi-threaded middleware Java application running on the indexing hosts. The *Netty IO*[1] library is used to handle incoming client requests using a thread pool. When a server is started, it first connects to configuration server to retrieve the cluster information and then is assigned an interval of the Z-order curve for which it is responsible. If the cluster already contains some data the new node can mark itself as a free node and will be contacted by the next node in the cluster that will begin a balancing operation. Indexing hosts update the intervals in the cluster configuration after a balancing operation and periodically update the number of entries stored.

The *configuration server* was implemented using *ZooKeeper*[2], an open source server whose goal is to provide reliable distributed coordination. ZooKeeper provides a centralized service for maintaining configuration information and performing distributed synchronization. The *cluster configuration* is stored in a ZooKeeper specific format by the indexing hosts. ZooKeeper also allows the possibility of replicating the stored configuration by running a *qourum*[3] of machines to improve availability.

A minimal configuration of the distributed PH-tree requires a ZooKeeper server, an indexing host running the *server library* and client machines performing operation using the *cluster library*.

5.2 PH-tree Java API

The Java API of the PH-tree is presented in Figure 5.2. The interface *PhTreeV* is the main interface and corresponds to a PH-tree that stores entries with keys of type *long[]* and arbitrary objects as values. A set API is provided through the classes *PhTree* and *PhTreeD*. *PhTree* uses a *PhTreeV* instance with empty values to implement a set for multi-dimensional points with values of type *long*. *PhTreeVD* is targeted towards application which require floating point precision for the values of each dimension. This type of tree is backed by the *PhTreeV* instance and performs pre and post processing of the multiple-dimensional points to map them from *double[]* to *long[]*.

The initial PH-tree implementation also provides support for range trees, which can store *hyper-rectangles* defined by two multi-dimensional points: the bottom-left and top-right corners of the hyper-rectangle. These types of trees are implemented using special types of pre and post processors and are internally backed by point PH-trees. *PhTreeRangeV* stores hyper-rectangles defined by multi-dimensional points represented as *long[]* and allows arbitrary values. *PhTreeRangeVD* is similar but allows points to be represented as *double[]*. Both of these two implementations use an instance of *PhTreeV* to store the processed hyper-rectangles, as

[1] http://netty.io/

[2] http://zookeeper.apache.org/

[3] The name used by ZooKeeper for a replicated cluster of ZooKeeper server.

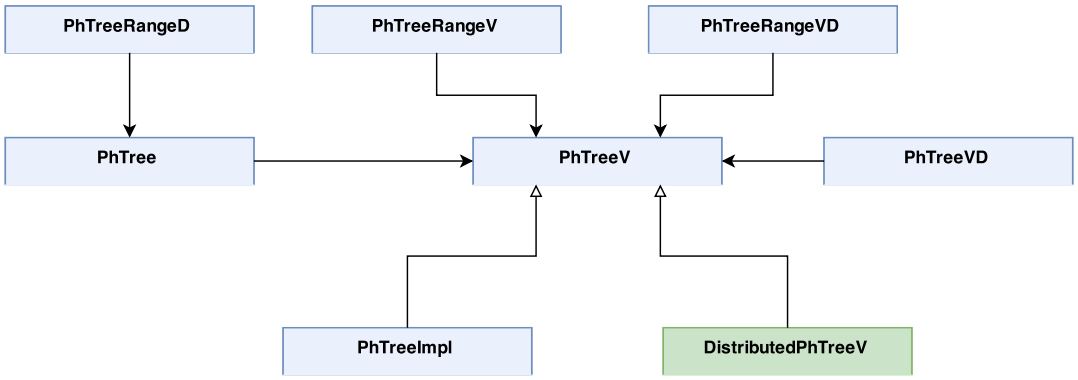


Figure 5.2: The Java API of the PH-tree. The classes coloured with blue represent the existing code, the additions in green represent the distributed implementation.

shown in Figure 5.2. Finally, *PhTreeRangeD* provides support for hype-rectangle set index is backed by a *PhTree* object.

As presented in Figure 5.2, all of the types of trees discussed depends on the interface PhTreeV. The distributed PH-tree provides another implementation for this interface. Method calls to any of the trees in the presented hierarchy are transformed into commands send by the *DistrbutedPhTreeV* class to the indexing hosts over the network. After the requests are serviced, *DistributedPhTreeV* combines the responses and provides the appropriate returning value.

One big advantage of this approach is that client is not aware if it is using a in-memory tree or a distributed one, except at the creation time. Therefore, application previously using the in-memory version of the PH-tree can easily switch to the distributed implementation, with minimal code changes.

The choice of providing a separate implementation only of the *PhTreeV* class has another important consequence: the indexing hosts only need to maintain an in-memory instance of a *PhTreeV* implementation, independent of the type of tree used by the client. This simplifies the implementation and also has the advantage of off-loading the pre and post processing operations from the servers to the clients, allowing faster request processing and better throughput on the servers.

5.3 Distribution

5.3.1 Client design

A high level diagram of the client library is presented in Figure 5.3. The *DistributedPhTreeV* class implements the *PhTreeV* interface and provides access the distributed PH-tree. Each

method call on the *PhTreeV* interface is delegated to a *PhTreeIndexProxy* object. This object converts each method call into a *Request* object, similar to the *Command design pattern*[4]. The proxy class also decides which hosts need to be contacted to resolve the operation, based on a locally cached *Key-Mapping* object. This object contains the most recent cluster configuration available to the host. The *Request* object is then passed on to a *RequestDispatcher*, where it is encoded to an array of bytes and sent to the corresponding hosts via the Java TCP Socket API. The *Transport* class keeps the most recently used connections alive, to avoid having to perform the TCP handshake for each operation when repeatedly interacting with the same host.

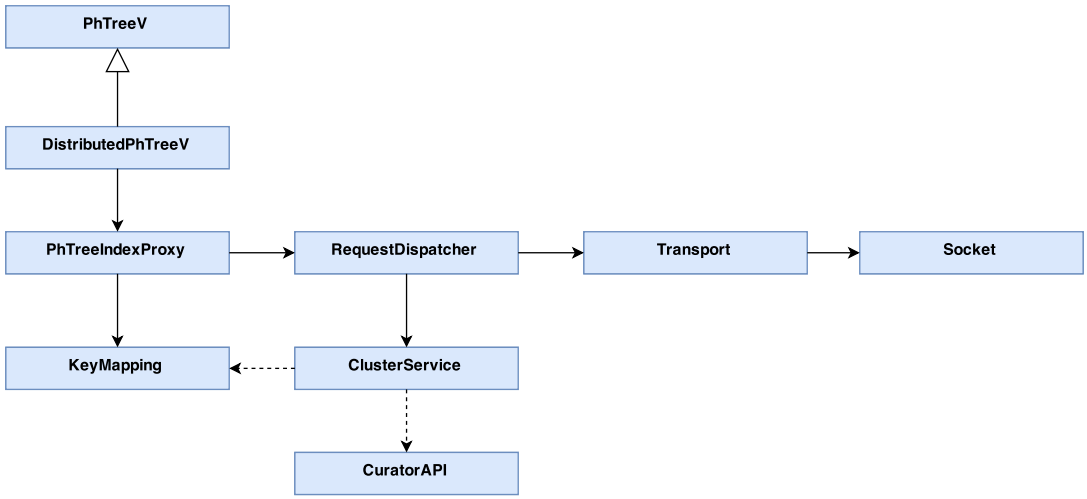


Figure 5.3: The class diagram of the client library

The calls to the *DistributedPhTreeV* methods are blocking, as the *Transport* object blocks the execution until it has received a response from the remote indexing host. The response is received as an array of bytes and is decoded into a *Response* object and returned to the *RequestDispatcher* object. If more hosts participated in the query, their corresponding *Response* objects are combined and the result of the operation is returned to the caller of the *DistributedPhTreeV* method.

5.3.2 Server design

The *server library* is implemented as a multi-threaded, asynchronous, event-driven server, according to the *reactor pattern*[5]. This model uses two types of threads: *Dispatcher* threads and *Handler* threads. A *Dispatcher* thread accepts incoming connections, allocates the necessary resources for the connection and then passes the connection resources to a *Handler* thread. The *Handler* thread accepts requests from the client using the connection resource provided by the *Dispatcher* thread. For the TCP protocol, this resource is generally a reference to a

[4] http://en.wikipedia.org/wiki/Command_pattern
[5] http://en.wikipedia.org/wiki/Reactor_pattern

Socket object. The *Handler* thread manages requests from more than one client by leveraging the event-based IO support provided by the operating system. The *Netty IO* Java library is used for the implementation, as it provides an easy manner to set up a pool of *Handler* threads. The chosen server architecture generally performs better than architectures which create a new thread for every request, as the *Handler* threads are pooled and are initialized before any client requests are serviced.

The high level diagram of the server library is presented in Figure 5.4. The main class of the *server library* is the *IndexMiddleware* class, which sets up the pool of handler threads and starts the dispatcher thread. The *MiddlewareChannelHandler* class is the first handler that is executed by one of the *Handler* threads upon receiving a request on a client connection. This class performs reads all of the bytes corresponding to a single request from a client and places them into a byte array object. This object is then passed to an instance of the *IOHandler* class, which coverts the byte array into a *Request* object. The *IOHandler* then passes the *Request* object to another handler, based on the type of the request. Client requests are passed along to a *RequestHandler* object, while balancing requests, coming from other servers are passed to the *BalancingRequestHandler* object.

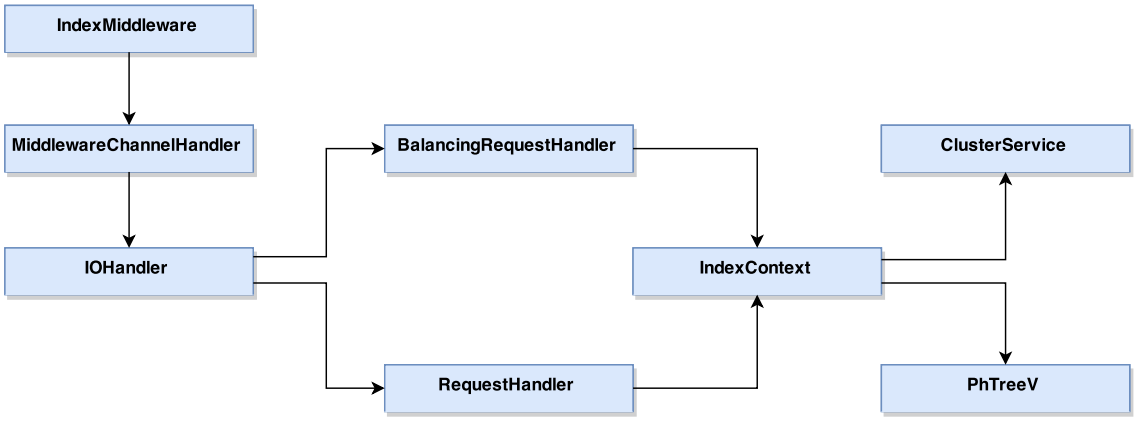


Figure 5.4: The class diagram of the server library

Information that is commonly accessed by all of the *Handler* threads is stored in the *IndexContext* object. This object contains a reference to the *ClusterService*, used to read and writer cluster configuration data, and to the in-memory PH-tree instance on that server, which is used in processing the client requests.

5.3.3 Configuration server and Key-Mapping implementation

The *Key-Mapping* was implemented using the ZooKeeper, an open-source coordination service. ZooKeeper provides access to a shared hierarchical namespace similar to a file system. This namespace is comprised of data registers, called *znodes*, which are identified by a path

and store data in the form of a byte array. The data stored within a *znode* is guaranteed to always be read and written atomically. Additionally, each *znode* can have any number of children *znodes*. ZooKeeper provides primitives to create and delete *znodes*, check if a node exists, get and set the data stored by a node and also retrieves the list of children of a node. It is also possible to set *watches* on certain znodes, and receive notifications from ZooKeeper when the data of the node was changed.

Both the client and the servers access the configuration data using the *ClusterService* object, which maintains a cached version of the configuration on a *KeyMapping* object. Handlers in the *server library* module can also update the configuration through the *ClusterService* API. The *ClusterService* receives notifications from the ZooKeeper server when any data was changed and updates the *Key-Mapping* object accordingly. The implementation of the *ClusterService* uses the *Curator*[6] framework, a set of libraries that provide a simpler and more reliable API for accessing ZooKeeper.

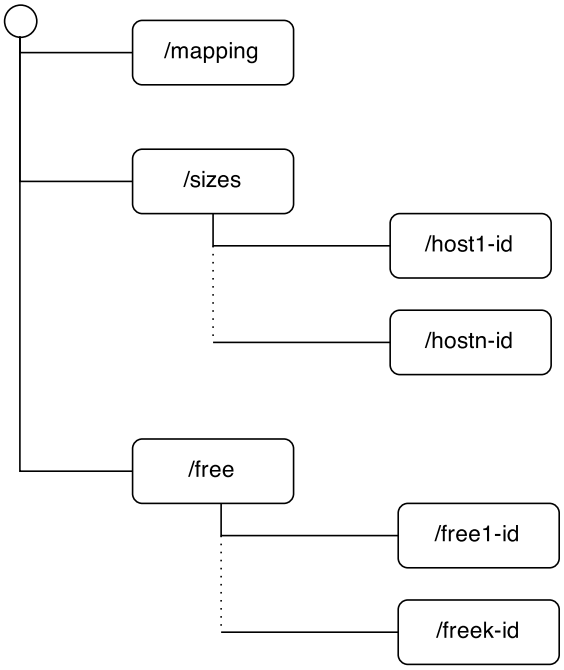


Figure 5.5: A diagram of the Key-Mapping stored in ZooKeeper

The ZooKeeper configuration for the distributed PH-tree is illustrated in Figure 5.5. This configuration was implemented using three main znodes:

- The */mapping znode* contains the set of all Z-order intervals, together the host responsible for each of these intervals. All of this information is serialized into a byte array and is always read and written atomically by the indexing hosts. This is done to ensure

[6] http://curator.apache.org/

that all Z-order intervals are read by the clients in an atomic manner, and always offer them a consistent view of the system.

- The */sizes* znode stores an approximation of the number of entries stored in each indexing host, which is needed to make decisions during the *re-balancing* operation. This znode contains one child znode for each host, the child node storing the approximate size of each host. This allows all of the hosts to update their size without clashing with updates from other hosts. Even though this means that the sizes of all of the hosts cannot be read atomically, the consistency loss is not significant, as the balancing operations do not require the precise number of entries stored on the neighbours.
- Finally, the */free* znode contains a list of hosts which have registered themselves as free on start-up and will be contacted by the balancing initiators during the next balancing operations.

As the client maintain a cached copy of the latest information in ZooKeeper, it is possible that, due to network delays, the mapping on the client be different from the actual intervals on the server. Consider the following situation: a client initiates a query and after analyzing the local *Key-Mapping*, it decides it has to contact host h to retrieve the value for a certain key k. However, before the request reaches host h, this host could have already participated in a re-balancing operation and moved a part of its entries to a neighbouring host. If the entry involved in the query was moved to a different host, host h no longer has it and would respond to the client that no such host exists in the cluster. This is wrong, as the entry was simply moved to a neighbouring host. To prevent this type of situations, the *Key-Mapping* has a version number that is incremented after each re-balancing operation. Each indexing host stores the version of the mapping that was generated during the last re-balancing operation in which that host was involved. Furthermore, the clients include the version number of their local mapping in the requests sent to the hosts. Upon receiving a request, the server checks the client mapping version against the version of its last re-balancing operation. If the client version number is higher or equal, the request can be serviced without any consistency loss, as any changes that may have increase the clients version happened to other intervals in the mapping. Instead, if the client's version number is smaller than the host's version, the client is noticed that it's mapping is outdated and should retry the query. The client keeps attempting to send queries to indexing servers until it has received a response for an up to date mapping.

In a cluster of n hosts, there can be at most $n/2$ balancing operations running in parallel on the cluster. Therefore, it is important that a host's updates to the mapping do not overwrite the changes made by other hosts. The mapping update operation is performed in a *compare-and-swap* manner, using the multi-update operation support provided by ZooKeeper. Using this type of operation, it is possible to perform an update to a znode only if its version (or alternatively, another znode's version) has not changed since the previous value was read. Following a re-balancing operation, the initiator hosts attempts to write the new version of the mapping to ZooKeeper. If the mapping was changed by another host, the initiator reads the new mapping, applies its changes on the interval bounds and retries the operation until it is successful. While this approach can lead to starvation if a host is consistently blocked by other hosts to update the mapping, the situation is not likely to happen. An alternative solution to this problem would have been to implement a distributed lock on the mapping through ZooKeeper. In that situation, the initiators would always need to perform three requests to

ZooKeeper: one request to acquire the lock, the second to perform the write and the third to release the lock. Our solution optimizes the duration of the re-balancing operation for the best case, when the write will be successful, as that situation is much more common in practice.

5.3.4 Communication protocol

The distributed PH-tree uses a custom binary protocol for both the communication between a client and an indexing host and between two indexing neighbouring indexing hosts. The main communication flow is described in Figure 5.6. The *RequestDispatcher* on the client creates a *Request* object for each method call to the PH-tree API. This request object contains the request type and the arguments received by the method and it is converted into an array of bytes by the *RequestEncoder* class, according to the communication protocol. The array of bytes is sent through the network to all of the servers that participate in the operation. On the server, the request is assigned to one of request processing threads which converts the array of bytes back to a *Request* object using a *RequestDecoder*. The request is then serviced by a *RequestHandler* object, which returns a *Response* object containing the result of the request. The *Response* object is converted into an array of bytes using a *ResponseEncoder* and sent back to the client over the wire. Back on the client, the response is decoded from a byte array. In case multiple servers were contacted, all of their responses are decoded and then combined to produce the return value of the method call.

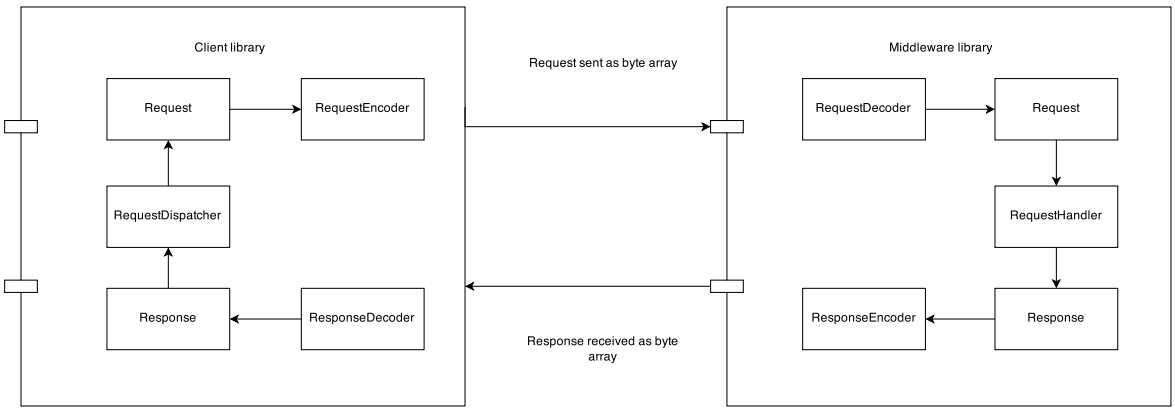


Figure 5.6: A diagram of the communication protocol between the client and the server

The main reasons for using a binary protocol, as opposed to a more high level protocol based

on JSON[7] or XML[8] are the following:

- A binary protocol produces a much smaller output than a text based protocol. In the case of the custom binary protocol, all parameters are converted into arrays of bytes of the smallest possible size and combined into a large byte array. JSON and XML formats require additional metadata to store the name of the fields and to make the representation human readable.
- JSON or XML protocols follow a two-step process: first the request is converted into a JSON or XML document, then the document is converted into an array of bytes to be sent over the network. The binary protocol skips the middle step and transforms requests directly into arrays of bytes. This makes the binary encoding faster than the text-based alternatives.
- The messages exchanged for this communication protocols are generally quite simple and thus the implementation effort for efficient custom serialization is not very high.

5.3.5 Iterators

Beside the number of queries that return a list of entries that match certain parameters, the PH-tree API also provides support for iterators. These iterators implement the Java iterator API and allow clients to iterator over all of the entries of the tree or over all of the entries that fall within a certain query hyper-rectangle.

Once an iterator is created using the client API, the client sends a request to the host holding the first key in the iterator. This host creates an in-memory iterator object that is maintained until the client has finished iterating over the entries. The client retrieves batches of entries from the iterator on the host, to avoid having to send a request each time the iterator is moved a single position. Once all entries in the iterator stored on a single host have been exhausted, the client contacts the host holding the following portion of the iterator data and repeats the procedure.

One advantage of this approach is that the client does not have to perform a network request for each entry of the iterator. Secondly, for iterators over large areas, spanning multiple hosts, it might also be unfeasible to request all of the entries in the iterator from all of the hosts and just iterator over them in-memory, on the client machine. A potential drawback of this solution is that each iterator running on a client has an associated iterator object running on an server machine. Thus, creating very large numbers of iterators could lead to large memory consumption on the servers and could potentially affect the performance in a negative manner.

5.4 Concurrency

The addition of concurrent support to the PH-tree involves the modification of the write operations to handle accesses from multiple threads. The operations which perform queries,

[7] http://json.org/
[8] http://en.wikipedia.org/wiki/XML

create or execute iterators across the PH-tree were not modified, to preserve the response time of the read operations. Therefore, only the *put()*, *delete()* and *updateKey()* methods for implementations of the *PhTreeV* interface require modifications to support concurrency.

The implementation was performed by isolating the write methods into a separate interface called *PhOperations*. Thus, the PH-tree implementation delegates the execution of the write methods to an internal *PhOperations* object. This pattern allows us to easily provide multiple implementation for concurrent write access strategies by placing each strategy into a separate implementation class.

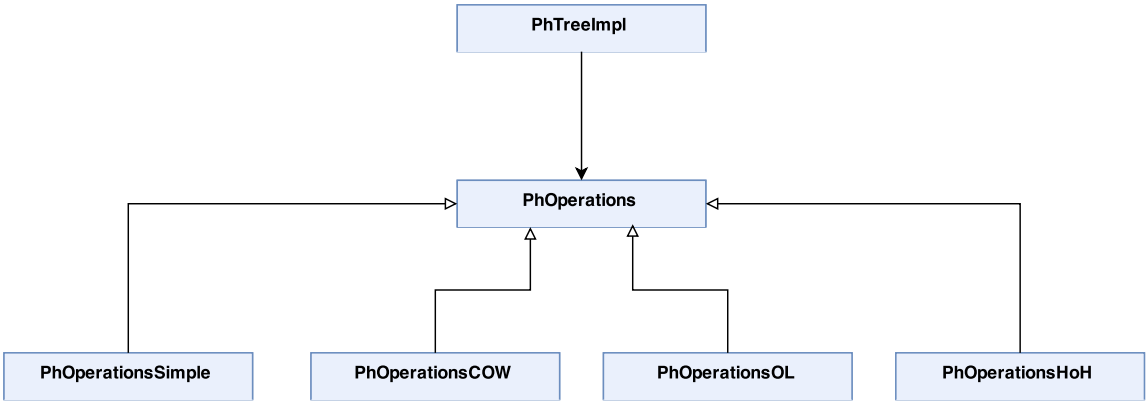


Figure 5.7: The Java API of the concurrency classes

Figure 5.7 shows the *PhOperations* interface and its implementations. First of all, *PhOperationsSimple* provides an no-concurrency implementation of the write operations, allowing applications that do not require concurrent access to by-pass the response time performance penalty added by the synchronized implementations. *PhOperationsCOW* provides a full Copy-on-Write implementation, which essentially creates a ”new” version[9] of the tree at each write operation and supports iteration on snapshots of the tree. This implementation supports a single writer thread accessing the three concurrently with an arbitrarily large number of readers. *PhOperationHoH* and *PhOperationsOL* are fine grained lock-based concurrency strategies, providing support for multiple reader and writer threads at the cost of strict consistency. In both of these implementations, the nodes modified are first copied and the modified copies are atomically replaced in the parent nodes. This removes the need for taking locks in the read methods. *PhOperationsHoH* implements the tree descent using hand-over-hand locking, while *PhOperationsOL* performs optimistic locking for this purpose. As previously discussed in Chapter 4, *PhOperationsOL* may need multiple descents on the tree until the locking attempt was successful and should generally perform better than *PhOperaionsHoH* for workloads in which the writers are not accessing the same area of the tree at the same

[9]Only a single path from the root to the update node is changed at each write operations, the rest of the tree is shared with the previous version.

time.

Another big advantage is that the *PhOperations* reference can be swapped at runtime, which changes the concurrency strategy used by the PH-tree. This allows a tree to use optimistic locking while it is ”loaded” with entries from multiple threads, and then switch to copy-on-write to allow iterators or range queries to execute on snapshots of the tree.

6 Evaluation

This chapter contains the performance evaluation of the distributed PH-tree. Section 6.1 presents the performance of the PH-tree from the distributed point of view, for indexing clusters of different sizes. The concurrent access strategies presented in Chapter 4 are evaluated in Section 6.2.

6.1 Distribution

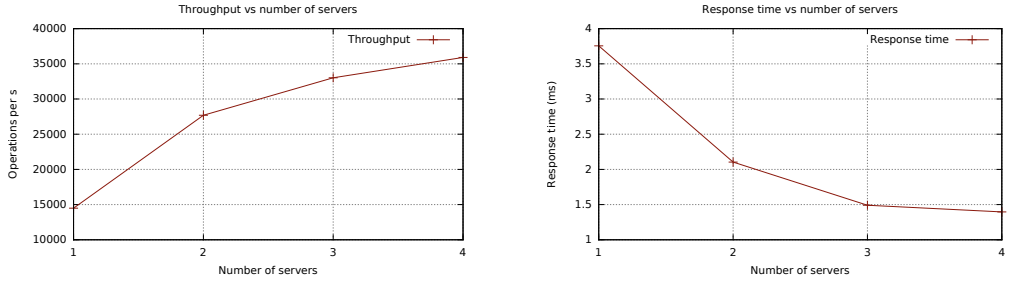
6.1.1 Experimental Setup

The experiments were executed to measure the insertion response time and throughput for distributed clusters containing different numbers of server hosts. The machines used were provided by the Google Computer Engine[1] cloud platform. Each server process was ran on a virtual instance containing 2 virtual CPU's and 7.5 GB of RAM memory. The client requests were generated several other virtual instances with the same hardware resources as the server machines.

Both the throughput and the response time were measured on the client machines. In the measurements, we consider that an operation performed on the distributed index starts in the moment when the method corresponding to this operation is called in the client code and ends when the same methods returns. Therefore, the duration of an operation includes the request encoding on the client side, the time needed by the request to travel over the network to the server machine, the processing time on the server machine, the time needed by the server response to travel over the network back to the client and the response decoding.

The experiments are run in the following manner. First, a client program is started on each client machine through a bash script and each client logs the duration of each request together

[1] https://cloud.google.com/compute/



(a) Write throughput

(b) Write response time

Figure 6.1: Evolution of the throughput and response time of the distributed PH-tree for different sizes of clusters

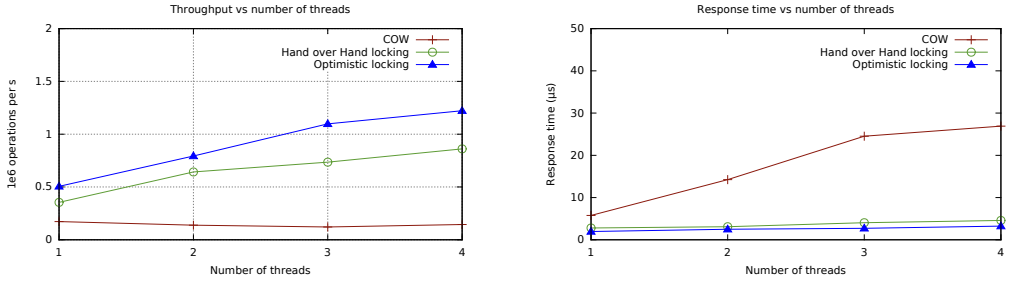
with the time stamp at which the request was completed. After all clients have finished executing, the log files are gathered from the client machines and merged into a master log. Because not all the clients start executing at precisely the same time, each experiment will be formed of three phases: *the build-up* phase, when some clients perform request and others have not started yet, the *stable request load* phase, when all clients are performing requests and the *cool down* phase, when some clients still perform request and others have finished. These phases are identified by looking at trace of the throughput of the system. The time intervals corresponding to the *build up* and *cool down* phases are removed from the master log and the stats presented are computed only on the data from the *stable request load* phase.

6.1.2 Experimental Results

The evolution of the throughput and response time of the distributed PH-tree when increasing the number of indexing servers is presented in Figure 6.1. For this experiment, requests were generated from a number of 6 client virtual machines. On each client machine, a number of 8 threads insert a number of 50000 3-dimensional points. The values for each dimension were sampled from the Gaussian distribution. At the end of each run of this experiment, the PH-tree contains 2.4 million entries. Because the number of entries inserted is the same across all runs, it is expected that setups with more servers will have a higher throughput and a lower response time.

Figure 6.1 shows that the throughput of the system increases together with the number of indexing servers. In the case of a single indexing host, the distributed index processes approximately 14500 operations per second. The addition of a second server increase the throughput to approximately 28000 operations per second. Adding the third and the fourth host increases the throughput even further, up to a 37000 operations per second for 4 servers.

The second thread which can be noticed in Figure 6.1 is that the response time for the insert operation decreases with the addition of extra indexing servers. In the case of a single server, the response time is around 3.4ms for each insert operation. As more server machines are added, the response time decrease to 2.1ms for the two server cluster, 1.5ms for the three server cluster and 1.39ms for the four server cluster.



(a) Write throughput

(b) Write response time

Figure 6.2: Evolution of write throughput and response time for 1 dimensional PH-trees

6.2 Concurrency

6.2.1 Experimental setup

The aim of the concurrency tests is the evaluate the throughput and response time of each implemented concurrency strategy. Furthermore, it is important to view how each concurrency strategy scales with respect to the number of concurrent threads accessing the PH-tree. Additionally, we identify the impact the concurrency strategies have when the PH-tree is only accessed by a single thread in the same time.

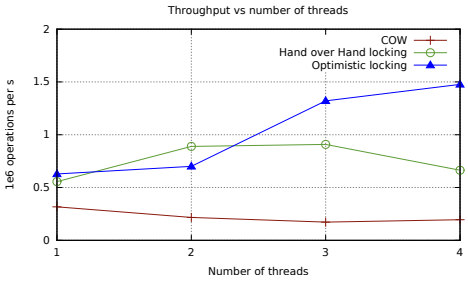
All tests were run on the following hardware: an Intel i5 processor with 4 CPU cores and 16 GB of RAM memory. Because the tree is kept completely in-memory, each individual operation is very fast and attempting to log the duration of each request to either an in-memory structure or a disk file would decrease the throughput of the PH-tree and skew the results. The measurements were performed using a background thread that is woken up at a fixed time interval, for example 1 second and count the number of operations performed since the last measurement.

6.2.2 Experimental results

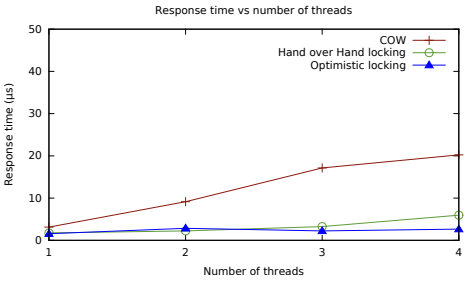
The first type of tests evaluate the throughput and response time of the concurrency strategies for an increasing number of threads. In each run, an array of 5 million multi-dimensional elements is inserted into an empty PH-tree. The values are sampled from the Gaussian distribution. The array is split equally into non-overlapping intervals between the inserting threads and each thread will insert a certain section of it. As the number of inserted entries is equal across the runs, it is expected that more inserting threads will lead to a higher throughput and a lower response time.

The experiments were performed for multi-dimensional entries with an increasing number of dimensions. Figure 6.2 shows the response time and throuput for 1 dimensional entries, 6.3 for 2 dimensional entries, 6.4 for 3 dimensional entries, 6.5 for 6 dimensional entries and 6.6 for 10 dimensional entries.

For all of the tests, the Copy-On-Write strategy maintains a relatively steady throughput, caused by the fact that this strategy only allows a single writer thread and multiple readers

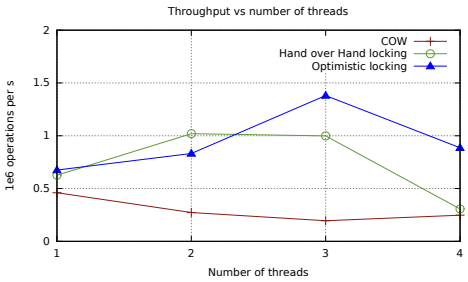


(a) Write throughput

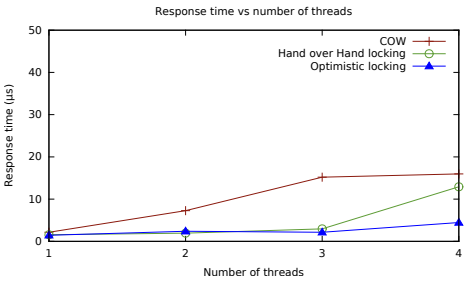


(b) Write response time

Figure 6.3: Evolution of write throughput and response time for 2 dimensional PH-trees

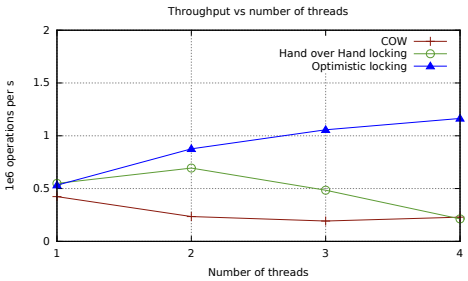


(a) Write throughput

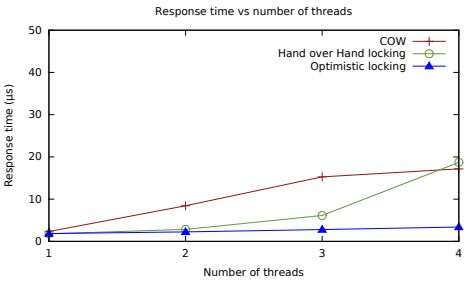


(b) Write response time

Figure 6.4: Evolution of write throughput and response time for 3 dimensional PH-trees

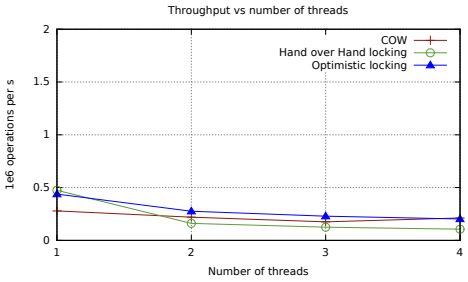


(a) Write throughput

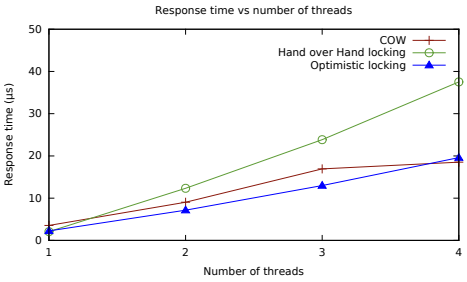


(b) Write response time

Figure 6.5: Evolution of write throughput and response time for 6 dimensional PH-trees



(a) Write throughput



(b) Write response time

Figure 6.6: Evolution of write throughput and response time for 10 dimensional PH-trees

at a time. The contention on the writer lock also causes the response time to increase as the number of writer threads increases.

For up to 6 dimensions, both the Hand-over-Hand and Optimistic locking strategies generally perform better than the Copy-on-Writer strategy, having a higher throughput and a lower response time. For both of these strategies, increasing the number of threads leads to an increase in throughput, up to a point where the lock contention becomes so high that it causes the throughput to drop. Optimistic locking generally outperforms Hand-over-Hand locking, having a better through put and a lower response time in most cases. Additionally, Optimistic locking generates less lock contention and manages to maintain the increase in throughput for a higher number of threads than Hand-over-Hand locking.

The second type of tests measures the penalty induced by the concurrency strategies when the tree is accessed by a single writer thread. For these experiments, 1 million 3-dimensional entries were inserted into an empty PH-tree by a single thread. The values for each entry were generated uniformly at random. Figure 6.7 illustrates the time needed for the insertion workload for each concurrency strategy and the no-concurrency version of the tree. For the no-concurrency version, the insertion finished in 1.1 seconds. The version will be used as a baseline for the other measurements.

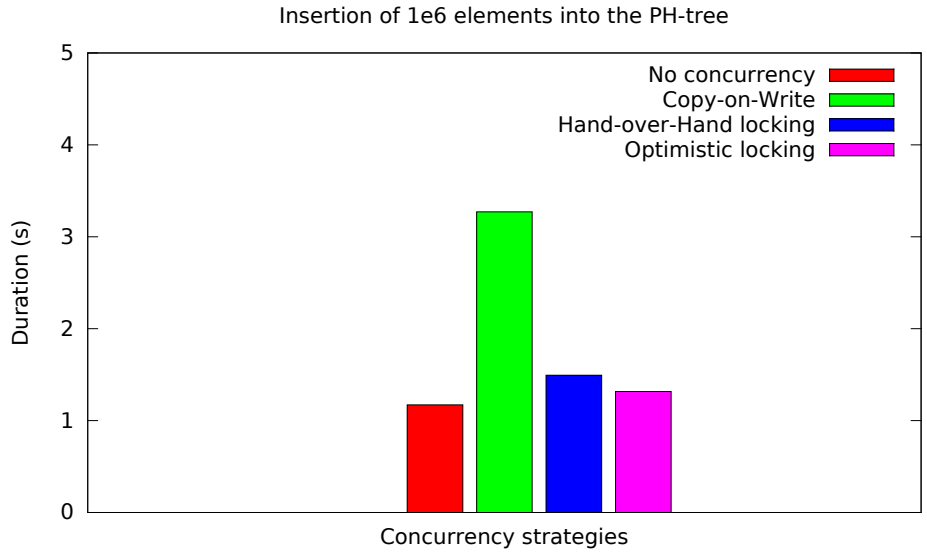


Figure 6.7: Time needed by 1 thread to insert 1e6 entries into the PH-tree

In the case of the Copy-on-Write version, the process finished in 3.2 second, taking almost three times longer than the baseline. This is due to the fact that each insert copies a full-path from the root of the tree to the modified node. The Hand-over-Hand locking finished took 1.49 seconds, and was approximately 30% slower than the baseline version. In the case of Hand-over-Hand locking, all nodes in the path from the root to the modified node are locked and copies are created only for the nodes which are updated. This makes the Hand-over-Hand strategy faster than the full Copy-on-Write strategy, but slower than the single-threaded implementation, which does not take any locks or perform any node copies. Finally, the Optimistic locking strategy performs the best from the implemented concurrency

strategies, finishing the insertion in 1.3 second. This strategy is faster than the Hand-over-Hand strategy, as only two nodes are locked, as opposed to a full path, but is still slower than the single-threaded implementation.

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 al-

gorithm first obtains the k nearest neighbours from the host responsible for the query point and then proceeds to run a range query on the cluster to check for any closer points on different hosts. The additional check is done by exploring 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

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] M. Moir and N. Shavit. Concurrent data structures. In *Handbook of Data Structures and Applications, D. Metha and S. Sahni Editors*, pages 47–14 – 47–30, 2007. Chapman and Hall/CRC Press.

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

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

[10] 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.

[11] 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.

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