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

9th March 2015

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

globis

Abstract

Here comes the abstract.

Contents

1 Introduction

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

1.1 Motivation

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

Several point-based multi-dimensional indexing solution have been developed in the latest years, the most prominent being kD-trees [1] and quadtrees [4]. This type of data structures store the multi-dimensional data such that more complex operations, like range and nearest neighbour queries are executed efficiently. The PhTree [9] is a new multi-dimensional data structure based on the quadtree. In addition to providing support for complex queries, the PhTree is also space-efficient, as its space requirements are sometimes even lower than those of multi-dimensional arrays.

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

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

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

1.2 Objectives

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

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

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

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

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

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

1.3 Thesis outline

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

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

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

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

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

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

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

2
Background

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

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

2.1 The PhTree

The PhTree [9] is a novel multi-dimensional indexing structure that focuses on fast retrieval and highly efficient storage. It combines concepts from PATRICIA-tries, quadtrees and hypercubes to encode the multi-dimensional data using bit prefix sharing. Because of this, the storage requirements can sometimes be lower than those of the storage requirements of an array of objects.

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

The PhTree has the following important properties:

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

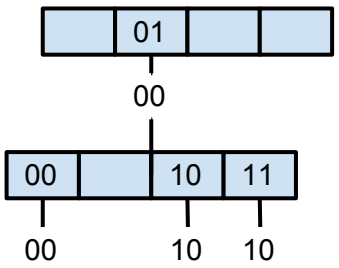

Figure 2.1: 2 dimensional 4bit PhTree containing the points: (0000, 1000), (0011, 1000), (0011, 1010).

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

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

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

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

the system. The goal is to design a set of algorithms that minimize the number of nodes to which a complex query is sent.

2.2 Related work

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

2.2.1 Distributed Indexes

Distributed Hash Tables (DHT's) are a class of decentralized distributed systems which expose an API similar to that of a hash-table. These P2P systems use a structured overlay network to assign a given key to a node in the system. Requests for a certain key can be sent to any node of the system and will be forwarded to the node responsible for it. DHT's have been designed to work on internet-scale, connecting large numbers of machines across large distances. The main concerns are scalability and availability, while the *get()* and *put()* operations have "best effort" semantics. Chord [8], one of the most popular and most efficient DHT system, manages to guarantee that any request for a certain key will arrive the responsible node in at most *O(logn)* hops, in the context in which each nod contains only *O(logn)* links to other nodes.

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

A similar approach is taken by the SkipIndex [10], a multi-dimensional indexing system that uses a SkipGraph to map geometrical regions to machine nodes. Each system node maintains only a partial view of the region tree and thus range and nearest neighbour queries are performed using selective multicasting.

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

2.2.2 Concurrent data structures

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

the data structure, in the approach called *fine grained locking*. Lock-free solutions use either atomic primitives, like *compare-and-swap* or software transactional memory to synchronize modification attempts on the data structure.

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

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

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

3
Index distribution

Meta - will be removed after editing. This chapter should focus on how the distributed index was implemented: how the data was split across the cluster nodes, the manner in which the queries are executed and how the entry load balancing is performed.

3.1 Challenges

Meta - will be removed after editing. Present the callenges of implementing a distributed system : scalability, load balancing, etc. Should not focus on issues like security, availability as those are not relevant to this report.

3.2 Distribution strategies

Meta - will be removed after editing. Present the possible ways in which the entries can be distributed across the cluster nodes. Talk about the advantages and disaadvantages of each approach. Say which approach was chosen and why.

3.2.1 Hashing

3.2.2 Spatial splitting

3.2.3 Z-Order curve splitting

3.3 Algorithms

Meta - will be removed after editing. This section should explain how the queries should be executed on the distributed system. Present the load balancing algorithm.

3.3.1 Point queries

3.3.2 Range queries

3.3.3 Nearest neighbour queries

3.3.4 Entry load balancing

3.4 Architecture

Meta - will be removed after editing. This section should explain how the queries should be executed on the distributed system. Present the load balancing algorithm.

4
Concurrency

Meta - will be removed after editing. Present the concurrency strategies that could be added to the PhTree and explain the consistency model associated with each strategy.

The PhTree does not currently support concurrent write operations. There are several strategies that could be employed to add concurrent writes.

4.1 Challenges

4.2 Concurrency strategies

4.2.1 Copy-on-Write

4.2.2 Locking

5 Implementation

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

5.1 System description

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

5.2 Technologies

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

6
Evaluation

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

7 Conclusions

7.1 Contributions

Meta - will be removed after editing. Conclude the report. This should reiterate the main points of the report and try to mirror the introduction

7.2 Future work

Meta - will be removed after editing. Present the points that were not tackled by the thesis and talk about possible future work.

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

[8] 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.

[9] 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.

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