UNIVERSITÄT DES SAARLANDES

Faculty of Natural Sciences and Technology I
Department of Computer Science

Workload-based Data Partitioning for Index Construction

Bachelor's Thesis

written by

Lukas Wilde

31st August 2022

Supervisors
Jens Dittrich

Advisor
First Advisor

1st Reviewer
Jens Dittrich

2nd Reviewer
Second Reviewer

Acknowledgement

Abstract

Contents

Chapter 1

Introduction

Here is a citation [1].

- DBMS routinely use index structures for increased performance
- Index pre-configured or chosen by user
- Mostly no utilization of underlying data or workload distribution
- Except: learned indexes -> Related Work
- Motivation: different data structures for different query workloads (hash table?)
- For this, introduce concept of hybrid index structures
- Create partitions to create singular indexes and combine them
- Optimize partitions based on one/multiple metrics
- As motivation: GENE, starting point for generic search
- Introduce what is covered in what section of this thesis

Chapter 2

Related Work

This chapter covers work related to Workload-based Data Partitioning. I also introduce the index structures used as baselines in the evaluation.

The well-known B^+-tree [2] is the first basic index structure used in the comparison.

The second index used for comparison is the Adaptive Radix Tree (ART) [3]. The authors recognize that the B^+-tree is widely used for disk-based database systems but indicate that it is unsuitable for modern-day main-memory databases. This is mainly due to the disparity of CPU cache sizes and main memory speed making main memory access times no longer uniform. Also, there is the problem of CPU stalls, which is caused by the CPU being unable to predict the result of comparisons easily. As comparisons are necessary to traverse a B^+-tree, this causes more latency for the index. To overcome these problems, the authors introduce an improvement to Radix Trees, which uses certain parts of the keys directly to guide the search in the tree. While Radix Trees get rid of the afore-mentioned CPU stalls, they often have to make a global trade-off between tree height and space efficiency. This problem is solved by introducing adaptive nodes with varying capacities to hold child pointers. Results show that ART can outperform other main-memory data structures, with the only competitor being a hash table. As these store keys in a random order, they cannot support range queries efficiently and are only useful in specific scenarios.

To cover the last index structure used in the comparison, we need to look at another class of index structures that only emerged recently. Learned index structures generally try to leverage recent progress in the field of Machine Learning to improve index performance.

The Recursive Model Index (RMI) [4] introduces the concept that indexes are models that simply map keys to positions in a sorted array. The authors state that most modern index structures do not consider the data distribution and miss out on highly relevant optimizations. While most datasets don't follow simple patterns, they argue that Machine Learning approaches can be used to incorporate these patterns. One can look at traversing a B$^+$-tree as slowly reducing the error it takes to find the final location of the key (e.g. 100M possible records are reduced to 1M). The same approach can be adapted to Machine Learning models. While it is hard to guarantee that a single model will reduce the error from 100M (possible keys) to hundreds for the final search, it is reasonable for a model to do so from 100M to 10k. With this in mind, the authors construct a hierarchy of Machine Learning models, where each model picks the next layer's model that should be used to predict the position of the key. This hierarchy does not need to follow a strict tree structure; each model can cover an individual number of keys. The benefit of this architecture lies in the ability to customize the models. For example, the bottom layer of nodes could only represent linear regression models (as there are many leaves and linear regression models are quite inexpensive), while higher up, one could use more complex neural network structures. The data segmentation happens through the given structure and training of the internal node's models. While this paper introduces the use of the underlying data distribution for index construction, it suffers from the explainability of neural networks. Therefore, there are no clear properties that could be used for my work.

FITing-Tree [5] tries to combine the flexibility of traditional index structures with learning by indexing linear data segments. The data partitioning is done by a single pass over the sorted data. The segmentation algorithm aims to determine the data segments' bounds so that the relation between keys and positions in the sorted array can be approximated by a linear function. To give reliable performance estimates, an error parameter is used to indicate how much an estimated position is allowed to deviate from the real position. A new segment is created once a point falls outside an error cone that ensures this maximum deviation. Otherwise, the cone is adjusted by tightening its bounds. Once the segments are determined, they are indexed by a B$^+$-tree to find a key's corresponding segment, and a binary search is performed inside the segment to find the actual position of the key.

The authors of the Piecewise Geometric Model index (PGM) [6], which I used as the third baseline in the evaluation, tried to improve upon the ideas of FITing-Tree. While FITing-Tree's approach seemed reasonable, a disadvantage was the data segmentation. The authors note that the single-pass segmentation algorithm does not produce the optimal number of data segments, leading to more data segments, a larger tree height, and increased lookup time. By reducing the segmentation to the problem of constructing a convex hull and allowing the index to be built recursively, they could increase the lookup time and ensure provably efficient time and space bounds in the worst case.

TODO: Distribution-aware PGM

While learned index structures perform so well because they can adapt to the underlying data distribution, apart from the distribution-aware PGM, they do not consider the workload that will be executed. RMI partition the data indirectly through their models, FITing-Tree and PGM explicitly use segmentation algorithms before building the index to determine the data that belongs in one segment. However, workload information might be beneficial to index construction, e.g. by indicating that certain data segments are not frequently requested. My work covers this problem: can workload information be used to improve data segmentation and thereby yield better performance?

Adaptive Hybrid Indexes [7] tackle the problem of selecting suitable encodings inside index structures to trade-off between space utilization and index performance. Compact indexes reduce the index's memory, allowing the database system to utilize that free memory to accelerate queries. However, they are naturally inferior to performance-optimized indexes. The decision of what encoding should be used on which part of the index is hard to make at build-time. Therefore, the authors propose to make these decisions at run-time. They introduce a framework that allows for monitoring the accesses across the index nodes when queries are processed, and based on these metrics, they classify whether nodes are cold or hot. Using so-called context-sensitive heuristic functions (CSHF), the framework determines, based on the classification of hot- and coldness, the memory budget, the historical classifications, and other properties, whether a node should have a compressed or performance-optimized encoding. Especially relevant to my work is the classification as hot or cold. Once the data is split into segments and inserted into a tree-like structure, it could be beneficial to modify the index based on this classification. While the authors do this at run-time, my work focuses on analyzing the workload before building the index. They use encodings for this, but one could also consider shifting leaves in the tree higher up to optimize for cache benefits.

- GENE [8] for the approach to look at indexes as logical components and combining them, generic search briefly to iterate over starting options and give our partitioning as a possible better starting point.

Distributed database systems are another field where the workload is used for partitioning. An example there is Schism [9]. The motivation behind this approach is to improve the performance and scalability of distributed databases. Each tuple is represented as a node in a graph. Two nodes are connected if the corresponding tuples occur in the same transaction. The edges are weighted with the total amount of co-occurrences in transactions. Given a number of partitions k, the algorithm will find a set of cuts of the edges that produces k distinct partitions with roughly equal weight and minimal costs along the cut edges. The intuition behind this is that tuples that are often accessed in the same transaction should also reside in the same partition/node to optimize query processing. By minimizing the cost along the cut edges, pairs of tuples that are seldom accessed together are split into different partitions, whereas often connected tuples stay in the same partition. While I don't look at transaction-based workloads in this work, there is a similarity in looking at workload properties to partition the data. Schism uses the frequency of co-occurrences to do this partitioning, which indicates that the frequency of query accesses could be a promising property to look at.

Chapter 3

Background

3.1 Hybrid Index Structures

- What are hybrid index structures?
- Advantages: optimize for subproblems, combine to one index
- challenges: correct combination of these structures (e.g. routing through data structure)

3.2 Partition and Partitioning functions

- Mathematical set theory definition of partition
- Adaption to key space/segments
- Partitioning functions for indexes
- Used in routing of through index

3.3 Numerical Differentiation

- Finite difference approximations
- Relation to true derivative (limes h -> 0)
- Consistency order of approximations
- Forward, Central, Backward finite difference approximations

Chapter 4

Approach and Algorithms

4.1 Approach

- Data generation
- Workload generation + parameters
- Partitioning (more details in section 4.2 and 4.3)
- Interface between Partitioning and Bulkloading
- (Informed) Bulkloading
- Benchmarking

4.2 Partitioning by Frequency

- Motivation: caching
- Idea from numerical approximations
- Algorithm

4.3 Partitioning by Purity

- Motivation: optimize index for different query types
- Algorithm

Chapter 5

Datasets and Workloads

This chapter deals with the used datasets and workloads

5.1 Datasets

- Generation procedure
- Used parameters for parameterized distributions

5.1.1 Synthetic Datasets

- uniform dense

5.1.2 Real-world Datasets

- SOSD datasets (osm, books, fb)

5.2 Workloads

5.2.1 Synthetic Workloads

- uniform sampling
- lognormal (because used in hybrid adaptive indexing paper)
- step workloads
- Proof of concept workload

5.2.2 Real-world Workloads

- Self-generated
- Are they representative (look into dbbench/YSCB)

- workloads especially OLTP often skewed (Identifying Hot and Cold Data in Main-Memory Databases, https://www.microsoft.com/en-us/research/wp-content/uploads/2013/04/ColdDataClassification-icde2013-cr.pdf)

Chapter 6

Evaluation

This chapter will deal with the evaluation of the experiments

6.1 Setup

- hardware
- index parameters like slot size, PGM epsilon etc.

6.2 Lookup Performance

6.2.1 Frequency Algorithm

6.2.2 Purity Algorithm

6.3 Role of Partitioning Parameters

- $$window_size$$
  , delta for frequency
- $$window_size$$
  for purity (as of yet)

6.3.1 Frequency Algorithm

6.3.2 Purity Algorithm

Chapter 7

Conclusion and Future Work

- Previous results reproducable?
- What have we found?
- Does partitioning yield better lookup times?
- Is it beneficial to move leaves higher up in tree?
- Is it beneficial to use hybrid index structures (i.e. change layout/data structure in nodes)
- Best case/worst case considerations?
- Future Work: Combination of metrics
- Future Work: Look at more data structures other than BinarySearch-Leaves and Hashtables
- Future Work: What other workload metrics can be used for partitioning?

Bibliography

[1] Douglas Comer. 'Ubiquitous B-Tree'. In: *ACM Comput. Surv.* 11.2 (June 1979), pp. 121–137. ISSN: 0360-0300. DOI: 10.1145/356770.356776. URL: https://doi.org/10.1145/356770.356776.

[2] R Bayer and E McCreight. 'Organization and maintenance of large ordered indices'. In: *Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control - SIGFIDET '70.* Houston, Texas: ACM Press, 1970.

[3] Viktor Leis, Alfons Kemper and Thomas Neumann. 'The adaptive radix tree: ARTful indexing for main-memory databases'. In: *2013 IEEE 29th International Conference on Data Engineering (ICDE).* 2013, pp. 38–49. DOI: 10.1109/ICDE.2013.6544812.

[4] Tim Kraska et al. *The Case for Learned Index Structures.* 2017. DOI: 10.48550/ARXIV.1712.01208. URL: https://arxiv.org/abs/1712.01208.

[5] Alex Galakatos et al. 'FITing-Tree'. In: *Proceedings of the 2019 International Conference on Management of Data.* ACM, June 2019. DOI: 10.1145/3299869.3319860. URL: https://doi.org/10.1145/3299869.3319860.

[6] Paolo Ferragina and Giorgio Vinciguerra. 'The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds'. In: *PVLDB* 13.8 (2020), pp. 1162–1175. ISSN: 2150-8097. DOI: 10.14778/3389133.3389135. URL: https://pgm.di.unipi.it.

[7] Christoph Anneser et al. 'Adaptive Hybrid Indexes'. In: *Proceedings of the 2022 International Conference on Management of Data.* ACM, June 2022. DOI: 10.1145/3514221.3526121. URL: https://doi.org/10.1145/3514221.3526121.

[8] Jens Dittrich, Joris Nix and Christian Schön. 'The next 50 years in database indexing or'. In: *Proceedings of the VLDB Endowment* 15.3 (Nov. 2021), pp. 527–540. DOI: 10.14778/3494124.3494136. URL: https://doi.org/10.14778/3494124.3494136.

[9] Carlo Curino et al. 'Schism'. In: *Proceedings of the VLDB Endowment* 3.1-2 (Sept. 2010), pp. 48–57. DOI: 10.14778/1920841.1920853. URL: https://doi.org/10.14778/1920841.1920853.

[10] Jialin Ding et al. 'ALEX: An Updatable Adaptive Learned Index'. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, June 2020. DOI: 10.1145/3318464.3389711. URL: https://doi.org/10.1145/3318464.3389711.

Appendix A

Appendix